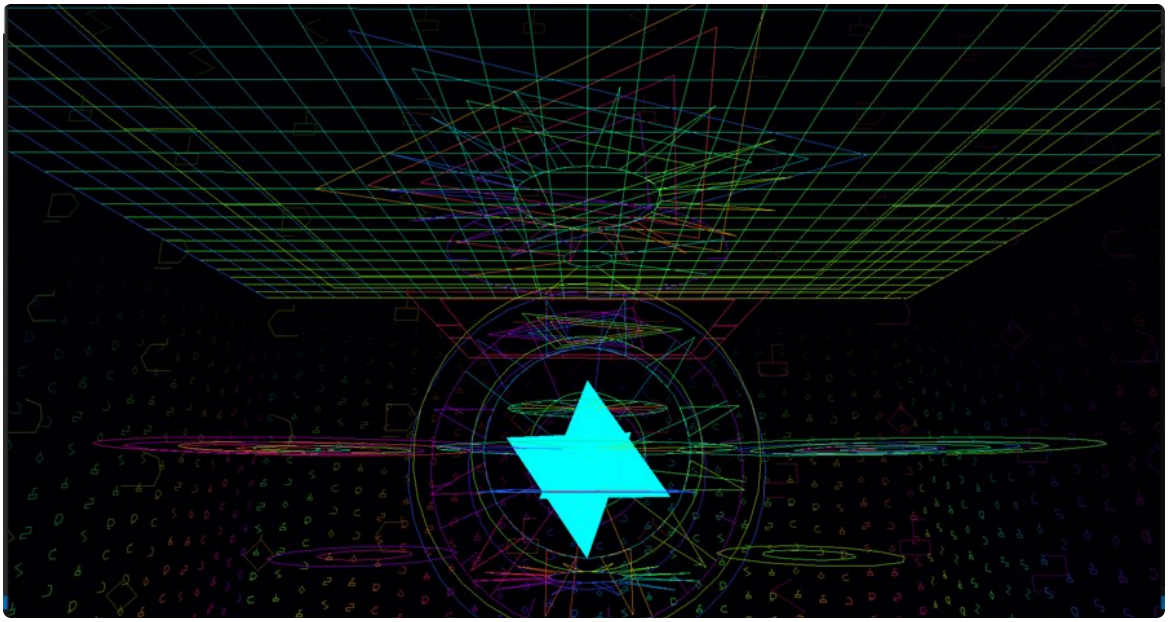
THE LING REFERENCE



灵 · 靈 · 령 · āɴ

The Ling Reference

Complete Reference

LING PRESS

The Ling Reference: Complete Reference

Copyright © 2026 Ling Press. All rights reserved.

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

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

First edition, 2026.

ISBN: TBD

Contents

WELCOME 5

How to Use This Book 6

COMPLETE REFERENCE 11

1. Keywords 12
2. Syntax, Types and Semantics 40
3. Core Builtins Reference 58
4. Command Line Tools: `ling` and `lingfu` 158
5. Window, Frame Loop, Input and Camera 174
6. Shapes, Vector Textures, Pixel Textures and Effects 210
7. Vector Fonts, the UI Toolkit and Dialogs 251
8. Animation: Anima 279
9. Audio, Music and the DAW 294
10. Physics, Game Helpers and the Body Toolkit 381
11. Input, Microphone and Speech-to-Text 445
12. The Graphics Crates 462
13. The Audio, Physics and Input Crates 480
14. Networking and the Web 503
15. Cryptography 559
16. AI, Language Models and Video 601
17. The Toolchain: Compiler Crates, Embedding and Extension 628

Known Limitations and Gotchas 646

PART

Welcome

How the reference is organised.

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

Complete Reference

Every keyword, builtin and crate API.

CHAPTER 1

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 (``) 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
```

`type`

Intended for type aliases: `type Name as Type`. The declaration is parsed and then discarded.

- **Status:** Parsed, no effect
- **Aliases:** zh: `符`; ja: `型`; ko: `타입`; th: `ชนิด` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)
- **Note:** The alias parser reads tokens until it meets `{`, `)`, `,`, `;` or `->`, so a `type` declaration swallows whatever follows it. In practice it can only appear as the very last item of a file. Avoid it.

```
bind start = do { println("types are not checked") }
type Id as number
```

```
OUTPUT
types are not checked
```

`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  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
```

`async`

Marks a function (`async fn`) or an expression (`async expr`) as asynchronous. It is accepted and then ignored: the function runs synchronously and immediately.

- **Status:** Parsed, no effect
- **Aliases:** zh: `异步`, `异`; ja: `非同期`; ko: `비동기`; th: `ไม่พร้อมกัน` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)
- **Note:** For real concurrency use the `thread_*` builtins (see the builtins chapter).

```
async fn work(n) { n * 2 }
bind start = do {
  println(work(4))
}
```

```
OUTPUT
8
```

`wait`

`wait expression` is meant as "await". It is a prefix operator that simply evaluates its operand on the interpreter.

- **Status:** Unreliable
- **Aliases:** zh: `等待`, `待`; ja: `待つ`; ko: `기다려`; th: `รอ` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

- **Note:** Under the default JIT backend `wait` of a call evaluates to `()`. Do not use it; call the function directly.

```
async fn work(n) { n * 2 }
bind start = do {
  println(wait work(4))
}
```

```
OUTPUT
()
```

Under `ling run --interp` this prints:

```
OUTPUT
8
```

`asm`

Inline-assembly marker: `asm("instruction")`. It parses to an assembly expression, which the interpreter ignores (the value is `()`); it is intended for bare-metal builds.

- **Status:** Parsed, no effect
- **Aliases:**

`post`

Documented as an idiomatic synonym of `return`/publish. The lexer recognises it but the parser gives it no meaning: it behaves as an ordinary identifier.

- **Status:** Reserved word only
- **Aliases:** zh: `发布`, `出`; ja: `投稿`; ko: `게시`; th: `ส่ง` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)
- **Note:** Because it is just an identifier, `post(1)` is a call to an unknown function and `bind post = 3` is legal.

```
bind start = do {
  bind post = 3
  println(post + 1)
}
```

```
OUTPUT
4
```

`give`

Documented as "yield a value". The lexer recognises it; the parser treats it as an ordinary identifier.

- **Status:** Reserved word only
- **Aliases:** zh: `给`, `予`; ja: `渡す`; ko: `전달`; th: `ให้` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
bind start = do {
  bind give = "a name"
  println(give)
}
```

```
OUTPUT
a name
```

`fit`

Documented as a pattern/refinement keyword. Recognised by the lexer, an ordinary identifier to the parser.

- **Status:** Reserved word only
- **Aliases:** zh: `适合`; ja: `適合`; ko: `적합`; th: `เหมาะสม` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
bind start = do {
  bind fit = 9
  println(fit)
}
```

```
OUTPUT
9
```

`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
```

`can`

Documented as a capability keyword. Recognised by the lexer; an ordinary identifier to the parser.

- **Status:** Reserved word only
- **Aliases:** zh: `能`; ja: `できる`; ko: `가능`; th: `สามารถ` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
bind start = do {
  bind can = true
  println(can)
}
```

```
OUTPUT
true
```

`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)
```

`sure`

Documented as "assert success" (unwrap). Recognised by the lexer; an ordinary identifier to the parser. To unwrap a result use `match`.

- **Status:** Reserved word only

- **Aliases:** zh: `确定`, `确`; ja: `確か`; ko: `확실`; th: `แน่นอน` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
fn unwrap(r) { match r { ok(v) => v, bad(e) => e } }
bind start = do {
  println(unwrap(ok(5)))
  println(unwrap(bad("failed")))
}
```

```
OUTPUT
5
failed
```

`maybe`

Documented as an optional-value type. Recognised by the lexer; an ordinary identifier to the parser. There is no option type; use `ok`/`bad` or `()`.

- **Status:** Reserved word only
- **Aliases:** zh: `可能`, `或`; ja: `多分`; ko: `아마도`; th: `อาจจะ` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

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

```
OUTPUT
1
```

`pure`

Documented as a "no side effects" marker. Recognised by the lexer; an ordinary identifier to the parser. Purity is not checked.

- **Status:** Reserved word only

- **Aliases:** zh: `纯`; ja: `純粋`; ko: `순수`; th: `บริสุทธิ์` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

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

```
OUTPUT
1
```

`spawn`

Documented as "start a concurrent task". Recognised by the lexer, but there is no builtin of that name and no special syntax; calling `spawn(...)` fails. Use `thread_spawn` for real threads.

- **Status:** Reserved word only
- **Aliases:** zh: `生成`, `启`; ja: `起動`; ko: `생성`; th: `สร้าง` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

`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
```

`none`

Documented as the absent value. The lexer recognises it but the parser treats it as an identifier named `none`, which is not defined: the interpreter reports `undefined: 'none'` and the JIT backend yields `()`. The real "nothing" value is `()`, obtained from a function with no result or from `()` written literally.

- **Status:** Unreliable
- **Aliases:** zh: `无`; ja: `なし`; ko: `없음`; th: `ไม่มี` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
bind start = do {
  bind nothing = ()
  println(nothing)
}
```

```
OUTPUT
()
```

`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
```

`where`

Introduces a constraint clause after a function signature: `fn f(x) where ... { }`. The tokens up to the opening brace are skipped and have no meaning.

- **Status:** Skipped
- **Aliases:** zh: `条件`; ja: `但し`; ko: `단`; th: `โดยที่` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
fn f(x) where x { x + 1 }
bind start = do {
  println(f(2))
}
```

```
OUTPUT
3
```

`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`, `শুরু`, `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 2

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 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 ` to rewrite a whole project into another language.

```
函 加倍(n) { 归 n * 2 }
令 后 = 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" 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]]
```

`format::join(list)`

Concatenates every element of a list as text, with no separator.

- **Aliases:** zh: `格式::拼接` (further languages exist)
- **Parameters:** `list` - a list of any values. If the first argument is not a list, the call behaves like `format`.
- **Returns:** Text.
- **Note:** `list_join(list, sep)` is the more general tool.

```
bind start = do {
  println(format::join(["a", 1, true, 2.5]))
}
```

```
OUTPUT
a1true2.5
```

Results and error handling

Ling has no exceptions. A function that can fail returns `ok(value)` or `bad(reason)`, and the caller inspects it with `match`. Both constructors are also keywords, so they work as match patterns.

Name	Summary
`ok(value)`	Wraps a value as a success result
`bad(reason)`	Wraps a value as a failure result
`try(function, arg, ...)`	Calls a closure and turns a runtime error inside it into a `bad` result instead of stopping the program

`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`.

```ling
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.

```ling
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)`)
`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

Name	Summary
`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.

```ling
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.

```ling
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.

```ling
bind start = do {
  println(acos(1))
  println(acos(0))
}
```

```
OUTPUT
0
1.5707963267948966
```

`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
```

`tanh(x)`

Hyperbolic tangent.

- **Aliases:** English: `tanhf`; zh: `双曲正切`, `双曲線正接`; ko: `쌍곡탄젠트` (further languages exist)
- **Parameters:** `x` - number.
- **Returns:** Number in -1..1.

```
bind start = do {
  println(tanh(0))
  println(tanh(1))
}
```

```
OUTPUT
0
0.7615941559557649
```

`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`, `対数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.

```ling
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.

```ling
bind start = do {
  println(lerp(0, 10, 0.25))
  println(lerp(10, 20, 1.5))
}
```

```
OUTPUT
2.5
25
```

`smoothstep(low, high, x)`

Smooth Hermite interpolation: 0 below `low`, 1 above `high`, an S-curve between.

- **Aliases:** zh: `平滑步进` ; ja: `スムーズステップ` ; ko: `스무스스텝` ; th: `เปลี่ยนแบบนุ่ม` (further languages exist)
- **Parameters:** `low` (default 0), `high` (default 1), `x` (default 0.5).
- **Returns:** Number in 0..1.

```
bind start = do {
  println(smoothstep(0, 1, 0.5))
  println(smoothstep(0, 10, 2))
  println(smoothstep(0, 10, 99))
}
```

```
OUTPUT
0.5
0.10400000000000002
1
```

`vnoise(x, y, seed)`

Smooth 2D value noise.

- **Aliases:** English: `noise2` ; zh: `柏林噪声2D` ; ja: `バリューノイズ2D` ; ko: `값노이즈2D` ; th: `นอยส์2ดี` (further languages exist)
- **Parameters:** `x` , `y` - coordinates; `seed` - integer (default 0).
- **Returns:** Number in 0..1.
- **Note:** Deterministic: the same arguments always give the same result.

```
bind start = do {
  println(vnoise(1.5, 2.5, 0) == vnoise(1.5, 2.5, 0))
  println(vnoise(1.5, 2.5, 0)  1000000000)
  remove_file("m.txt")
}
```

```
OUTPUT
true
```

`file_created(path)`

Creation time as Unix seconds (modification time on filesystems without one).

- **Aliases:** th: `เวลาที่สร้าง` (further languages exist)
- **Parameters:** `path`.
- **Returns:** Number; `0` if unavailable.

```
bind start = do {
  write_file("c.txt", "x")
  println(file_created("c.txt") > 1000000000)
  remove_file("c.txt")
}
```

```
OUTPUT
true
```

`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
```

`file_write_bytes(path, bytes, append)`

Writes raw bytes (the text file builtins cannot).

- **Aliases:** none (English spelling only)
- **Parameters:** `path`; `bytes` - list of numbers 0..255 (clamped); `append` - optional boolean, true to append.
- **Returns:** Boolean success.

```
bind start = do {
  println(file_write_bytes("b.bin", [72, 105, 33]))
  println(file_size("b.bin"))
  println(read_file("b.bin"))
  remove_file("b.bin")
}
```

```
OUTPUT
true
3
Hi!
```

`file_append_file(dest, source)`

Appends the contents of one file to another.

- **Aliases:** none (English spelling only)
- **Parameters:** `dest`, `source`.

- **Returns:** Bytes copied, or `-1` on failure.

```
bind start = do {
  write_file("a.txt", "AB")
  write_file("b.txt", "CD")
  println(file_append_file("a.txt", "b.txt"))
  println(read_file("a.txt"))
  remove_file("a.txt")
  remove_file("b.txt")
}
```

```
OUTPUT
2
ABCD
```

`file_magic(path)`

Classifies a file by its first bytes.

- **Aliases:** th: `มายาไฟล์` (further languages exist)
- **Parameters:** `path`.
- **Returns:** `"gzip"`, `"zip"`, `"png"`, `"other"`, or `"missing"`.

```
bind start = do {
  write_file("x.txt", "plain text")
  println(file_magic("x.txt"))
  println(file_magic("nope.bin"))
  remove_file("x.txt")
}
```

```
OUTPUT
other
missing
```

Program arguments, environment and processes

Access to the command line and the operating system.

Name	Summary
`get_args()`	Returns the whole process command line
`cli_arg(name, default)`	Reads the value of a `--name value` command-line option
`env_get(name, default)`	Reads an environment variable
`run_process(command, args, timeout_seconds, log_path)`	Runs a program (no shell involved) and waits for it to exit
`script_load(path)`	Loads a plain `.ling` file of functions while the program is running
`script_call(name, args)`	Calls a function defined by a loaded script
`script_functions()`	Names of all functions loaded from scripts

`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
```

`run_process(command, args, timeout_seconds, log_path)`

Runs a program (no shell involved) and waits for it to exit.

- **Aliases:** th: `รันโปรเซส` (further languages exist)

- **Parameters:** `command`; `args` - list of strings; `timeout_seconds` - kill the process after this long; `log_path` - file that receives standard output and error (appended).
- **Returns:** Exit code; `-1` if it could not start; `-2` if killed after the timeout.
- **Note:** Arguments are passed as an array, so there is no shell quoting or injection.

```
GUI
bind start = do {
  println(run_process("cmd", ["/c", "echo", "hi"], 10, "proc.log"))
  println(trim(read_file("proc.log")))
  remove_file("proc.log")
}
```

`script_load(path)`

Loads a plain `.ling` file of functions while the program is running.

- **Aliases:** none (English spelling only)
- **Parameters:** `path` - file named `.ling`; every function in it must be named `_...` and the file may contain nothing but `fn` definitions.
- **Returns:** `[success, message, stem]`.
- **Note:** Loading the same file again replaces its functions. Scripts cannot overwrite functions of the main program.

```
bind start = do {
  println(script_load("arp.ling"))
  println(script_functions())
  println(script_call("arp_up", [41]))
}
```

```
OUTPUT
[true, , arp]
[arp_twice, arp_up]
[true, 42]
```

`script_call(name, args)`

Calls a function defined by a loaded script.

- **Aliases:** none (English spelling only)
- **Parameters:** `name` ; `args` - list of arguments.
- **Returns:** `[true, value]` on success or `[false, message]` on failure.

```
bind start = do {
  script_load("arp.ling")
  println(script_call("arp_twice", [1, 2]))
  println(script_call("missing", []))
}
```

```
OUTPUT
[true, 5]
[false, no script function 'missing']
```

`script_functions()`

Names of all functions loaded from scripts.

- **Aliases:** none (English spelling only)
- **Parameters:** none.
- **Returns:** Sorted list of names.

```
bind start = do {
  script_load("arp.ling")
  println(script_functions())
}
```

```
OUTPUT
[arp_twice, arp_up]
```

Time and randomness

Name	Summary
`sleep(milliseconds)`	Pauses the program
`rand()`	Pseudo-random number in the range 0 up to (not including) 1

Name	Summary
`time_now()`	Seconds elapsed since the program started
`epoch_now()`	Wall-clock time as seconds since 1 January 1970 UTC
`random_hex(bytes)`	Cryptographically secure random bytes, as lower-case hex

`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 && die = 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)

Name	Summary
`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("\"Tom\" & Jerry"))
}
```

```
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
```

`base64_decode_to_file(base64, path)`

Decodes base64 and writes the bytes to a file (a `data:...base64,` prefix is tolerated).

- **Aliases:** ko: `base64파일로저장` (further languages exist)

- **Parameters:** `base64`; `path`.
- **Returns:** Boolean success.

```
bind start = do {
  println(base64_decode_to_file("aGk=", "b64.txt"))
  println(read_file("b64.txt"))
  remove_file("b64.txt")
}
```

```
OUTPUT
true
hi
```

Regular expressions

JavaScript-flavoured regular expressions (lookaround and back-references supported). The engine is stateful: compile a pattern once to get a numeric handle, then run it as often as needed.

Name	Summary
`re_compile(pattern, flags)`	Compiles a pattern
`re_exec(handle, text, start)`	Finds the first match at or after a character position
`re_error()`	Message for the most recent failed `re_compile`
`re_names(handle)`	Names of the capture groups of a compiled pattern

`re_compile(pattern, flags)`

Compiles a pattern.

- **Aliases:** none (English spelling only)
- **Parameters:** `pattern` - JavaScript-style regex text (remember to double backslashes in a Ling string); `flags` - any of `i m s u g y`.
- **Returns:** A handle (number >= 0), or `-1` if the pattern is invalid; then `re_error()` explains why.

```
bind start = do {
  bind re = re_compile("(\\d+)-(\\w+)", "")
  println(re >= 0)
  println(re_compile("(", ""))
  println(re_error())
}
```

```
OUTPUT
true
-1
Parsing error at position 1: Opening parenthesis without closing parenthesis
```

`re_exec(handle, text, start)`

Finds the first match at or after a character position.

- **Aliases:** none (English spelling only)
- **Parameters:** `handle`; `text`; `start` (default 0).
- **Returns:** `[]` when nothing matches; otherwise `[match_start, match_end, whole_match, group1, group2, ...]` with character (not byte) positions and `-1` for a group that did not take part.

```
bind start = do {
  bind re = re_compile("(\\d+)-(\\w+)", "")
  println(re_exec(re, "abc 12-xy z", 0))
  println(re_exec(re, "no digits", 0))
}
```

```
OUTPUT
[4, 9, 12-xy, 12, xy]
[]
```

`re_error()`

Message for the most recent failed `re_compile`.

- **Aliases:** none (English spelling only)

- **Parameters:** none.
- **Returns:** Text.

```
bind start = do {
  re_compile("[", "")
  println(re_error() != "")
}
```

```
OUTPUT
true
```

`re_names(handle)`

Names of the capture groups of a compiled pattern.

- **Aliases:** none (English spelling only)
- **Parameters:** `handle`.
- **Returns:** List with one entry per group; unnamed groups are empty strings.

```
bind start = do {
  bind re = re_compile("(?\\d{4})-(?\\d{2})", "")
  println(re_names(re))
}
```

```
OUTPUT
[, year, mon]
```

Archives

Name	Summary
`zip_files(paths, output)`	Packs files into a zip archive; directories are flattened to file names
`zip_dir(directory, output)`	Zips a whole directory tree, keeping relative paths
`tar_gz_list(archive)`	Lists the file entries of a `.tar.gz` archive
`tar_gz_read(archive, entry)`	Reads one text entry from a `.tar.gz` archive (first 256 KiB)

Name	Summary
`tar_gz_extract_to(archive, entry, output)`	Extracts one entry byte-for-byte to a file
`tar_gz_extract_all(archive, directory)`	Unpacks an entire `.tar.gz`; entries that would escape the target directory are rejected

---

`zip_files(paths, output)`

Packs files into a zip archive; directories are flattened to file names.

- **Aliases:** ko: `압축파일` (further languages exist)
- **Parameters:** `paths` - list of file paths; `output` - zip path (default `out.zip`).
- **Returns:** Boolean success.

```
bind start = do {
  write_file("a.txt", "A")
  println(zip_files(["a.txt"], "pack.zip"))
  println(file_magic("pack.zip"))
  remove_file("a.txt")
  remove_file("pack.zip")
}
```

```
OUTPUT
true
zip
```

`zip_dir(directory, output)`

Zips a whole directory tree, keeping relative paths.

- **Aliases:** th: `บีบอัดโฟลเดอร์` (further languages exist)
- **Parameters:** `directory`; `output`.
- **Returns:** Boolean success.

```
bind start = do {
  make_dir("pkg/sub")
  write_file("pkg/sub/f.txt", "x")
  println(zip_dir("pkg", "pkg.zip"))
  println(file_magic("pkg.zip"))
  remove_dir_all("pkg")
  remove_file("pkg.zip")
}
```

```
OUTPUT
true
zip
```

`tar_gz_list(archive)`

Lists the file entries of a `.tar.gz` archive.

- **Aliases:** th: `รายการทาร์` (further languages exist)
- **Parameters:** `archive` - path.
- **Returns:** Sorted list of entry names; empty list if unreadable.

```
bind start = do {
  println(tar_gz_list("missing.tgz"))
}
```

```
OUTPUT
[]
```

`tar_gz_read(archive, entry)`

Reads one text entry from a `.tar.gz` archive (first 256 KiB).

- **Aliases:** th: `อ่านทาร์` (further languages exist)
- **Parameters:** `archive`; `entry` - name exactly as listed.
- **Returns:** Text; empty if not found.

```
GUI
bind start = do {
  println(tar_gz_read("pack.tgz", "readme.txt"))
}
```

`tar_gz_extract_to(archive, entry, output)`

Extracts one entry byte-for-byte to a file.

- **Aliases:** th: `แตกไฟล์ทาร์` (further languages exist)
- **Parameters:** `archive`; `entry`; `output` path.
- **Returns:** Boolean (true if the entry was found and written).

```
GUI
bind start = do {
  println(tar_gz_extract_to("pack.tgz", "logo.png", "logo.png"))
}
```

`tar_gz_extract_all(archive, directory)`

Unpacks an entire `.tar.gz`; entries that would escape the target directory are rejected.

- **Aliases:** th: `แตกทาร์ทั้งหมด` (further languages exist)
- **Parameters:** `archive`; `directory`.
- **Returns:** Boolean success.

```
GUI
bind start = do {
  println(tar_gz_extract_all("pack.tgz", "unpacked"))
}
```

SQLite database

A single built-in SQLite connection. Parameters bind positionally as `?1`, `?2`, … : numbers as numbers, booleans as 0/1, everything else as text.

Name	Summary
`db_open(path)`	Opens (or creates) a database file and makes it the current connection
`db_exec(sql, param, ...)`	Runs a statement that does not return rows
`db_query(sql, param, ...)`	Runs a query and returns its rows

`db_open(path)`

Opens (or creates) a database file and makes it the current connection.

- **Aliases:** th: `ฐานข้อมูลเปิด` (further languages exist)
- **Parameters:** `path` (default `app.db`).
- **Returns:** `()`. Failure is a runtime error.

```
bind start = do {
  db_open("demo.db")
  db_exec("drop table if exists t")
  db_exec("create table t(a, b)")
  println(db_exec("insert into t values (?1, ?2)", 1, "x"))
  remove_file("demo.db")
}
```

```
OUTPUT
1
```

`db_exec(sql, param, ...)`

Runs a statement that does not return rows.

- **Aliases:** th: `ฐานข้อมูลรัน` (further languages exist)
- **Parameters:** `sql`; any number of bound parameters.
- **Returns:** Number of rows affected. Error if no database is open or the SQL is invalid.
- **Note:** Call `db_open` first.

```
bind start = do {
  db_open("demo2.db")
  db_exec("drop table if exists t")
  db_exec("create table t(n)")
  db_exec("insert into t values (?1)", 5)
  println(db_exec("update t set n = n + 1"))
  remove_file("demo2.db")
}
```

```
OUTPUT
1
```

`db_query(sql, param, ...)`

Runs a query and returns its rows.

- **Aliases:** th: `ฐานข้อมูลถาม` (further languages exist)
- **Parameters:** `sql`; bound parameters.
- **Returns:** List of `Row` structs; read a column as `row.column_name`. SQL NULL becomes an empty string, blobs become base64 text.

```
bind start = do {
  db_open("demo3.db")
  db_exec("drop table if exists t")
  db_exec("create table t(name, age)")
  db_exec("insert into t values (?1, ?2)", "Ada", 36)
  db_exec("insert into t values (?1, ?2)", "Lin", 29)
  bind rows = db_query("select * from t where age > ?1 order by age", 30)
  println(len(rows))
  println(rows[0].name)
  println(rows[0].age)
  remove_file("demo3.db")
}
```

```
OUTPUT
1
Ada
36
```

Bytes, hex and big numbers

Binary data crosses the language boundary as **lower-case hex strings** (two characters per byte). Functions that can fail on malformed input return `ok(value)` / `bad(reason)`; the rest raise a runtime error. Integer arguments must be non-negative whole numbers below 2^53.

Name	Summary
`hex_concat(list)`	Concatenates a list of hex strings
`hex_len(hex)`	Length in bytes
`hex_slice(hex, start, length)`	Extracts bytes `start
`hex_eq(a, b)`	Constant-time equality of two hex strings (case-insensitive)
`u8_hex(n)`	Encodes a number as one byte
`u32le_hex(n)`	Encodes a number as 4 little-endian bytes
`u64le_hex(n)`	Encodes a number as 8 little-endian bytes
`varint_hex(n)`	Encodes a number as a minimal LEB128 variable-length integer
`hex_u8(hex, position)`	Reads one byte at a byte offset
`hex_u32le(hex, position)`	Reads a little-endian 32-bit number
`hex_u64le(hex, position)`	Reads a little-endian 64-bit number (must be below 2^53)
`hex_varint(hex, position)`	Reads a LEB128 integer
`text_hex(text)`	UTF-8 bytes of a string, as hex
`hex_text(hex)`	Decodes hex as UTF-8 text
`base32_encode(hex)`	RFC 4648 base32 (lower case, unpadded) of the bytes
`base32_decode(text)`	Decodes base32 to hex
`base64url_encode(hex)`	URL-safe base64 of the bytes
`base64url_decode(text)`	Decodes URL-safe base64 to hex
`big_add(a, b)`	Unsigned 256-bit addition on decimal strings (numbers are also accepted)

Name	Summary
`big_sub(a, b)`	Unsigned subtraction; underflow is a runtime error
`big_mul(a, b)`	Unsigned multiplication
`big_div(a, b)`	Floor division; division by zero is a runtime error
`big_cmp(a, b)`	Compares two big numbers
`file_write_hex(path, hex)`	Atomically writes bytes (given as hex) to a file via a temporary file and rename
`file_read_hex(path)`	Reads a whole file as hex
`sys_idle_secs()`	Seconds since the last keyboard or mouse input system-wide
`sys_on_battery()`	True when the machine runs on battery power

`hex_concat(list)`

Concatenates a list of hex strings.

- **Aliases:** none (English spelling only)
- **Parameters:** `list` of hex strings.
- **Returns:** Hex text.

```
bind start = do {
  println(hex_concat(["ab", "cd", "01"]))
}
```

```
OUTPUT
abcd01
```

`hex_len(hex)`

Length in bytes.

- **Aliases:** none (English spelling only)
- **Parameters:** `hex`.
- **Returns:** Number.

```
bind start = do {
  println(hex_len("abcdef"))
}
```

```
OUTPUT
3
```

`hex_slice(hex, start, length)`

Extracts bytes `start .. start+length` (byte offsets).

- **Aliases:** none (English spelling only)
- **Parameters:** `hex`; `start`; `length`.
- **Returns:** Hex text.

```
bind start = do {
  println(hex_slice("00112233", 1, 2))
}
```

```
OUTPUT
1122
```

`hex_eq(a, b)`

Constant-time equality of two hex strings (case-insensitive).

- **Aliases:** none (English spelling only)
- **Parameters:** `a`, `b`.
- **Returns:** Boolean.

```
bind start = do {
  println(hex_eq("AB", "ab"))
  println(hex_eq("ab", "ac"))
}
```

```
OUTPUT
true
false
```

`u8_hex(n)`

Encodes a number as one byte.

- **Aliases:** none (English spelling only)
- **Parameters:** `n` 0..255.
- **Returns:** Hex text.

```
bind start = do {
  println(u8_hex(255))
}
```

```
OUTPUT
ff
```

`u32le_hex(n)`

Encodes a number as 4 little-endian bytes.

- **Aliases:** none (English spelling only)
- **Parameters:** `n`.
- **Returns:** Hex text.

```
bind start = do {
  println(u32le_hex(1))
  println(u32le_hex(258))
}
```

```
OUTPUT
01000000
02010000
```

`u64le_hex(n)`

Encodes a number as 8 little-endian bytes.

- **Aliases:** none (English spelling only)
- **Parameters:** `n` = 0)
}
```

```
OUTPUT
true
```

`sys_on_battery()`

True when the machine runs on battery power.

- **Aliases:** none (English spelling only)
- **Parameters:** none.
- **Returns:** Boolean (Windows only; `false` elsewhere).

```
bind start = do {
  bind b = sys_on_battery()
  println(b == true || b == false)
}
```

```
OUTPUT
true
```

Threads, channels and shared data

Each thread is an **isolate**: its own interpreter on its own operating-system thread. Nothing is shared except channels and a write-once store, so Ling programs cannot race. Values sent across (arguments, messages, return values) are deep-copied; functions cannot be sent. Isolates always run on the tree-walking interpreter. Handles (threads, channel ends) are unguessable strings valid only in the isolate that owns or received them. Limits: 1024 live threads, 64 MiB per message, 65 536 queued messages per channel.

The worker functions below live in a file named `worker.ling` next to the program that spawns them.

Name	Summary
`thread_spawn(file, function, args, capabilities)`	Starts a function from another source file in a new isolate
`thread_join(handle)`	Waits for a thread to finish and returns its result
`thread_join_timeout(handle, milliseconds)`	Like `thread_join` but gives up after a time limit
`thread_alive(handle)`	True while the thread is still running
`thread_cancel(handle)`	Asks a thread to stop
`thread_self()`	Handle string identifying the calling isolate
`thread_cpus()`	Number of hardware threads available
`thread_caps()`	The capabilities of the calling isolate
`chan_new(capacity)`	Creates a bounded channel
`chan_send(send_handle, value)`	Sends a value, blocking while the channel is full
`chan_recv(receive_handle)`	Receives the next value, blocking until one arrives
`chan_select(receive_handles, timeout_ms)`	Waits on several channels at once

Name	Summary
`chan_close(handle)`	Closes a channel end
`chan_len(handle)`	Number of messages queued
`shared_put(key, value)`	Stores a value in the process-wide write-once store
`shared_get(key, default)`	Reads a value from the shared store
`Producer / consumer pattern`	A worker streams values through a channel while the main program consumes them (an idiom, not a builtin)

`thread_spawn(file, function, args, capabilities)`

Starts a function from another source file in a new isolate.

- **Aliases:** none (English spelling only)
- **Parameters:** `file` - path of the `.ling` file, resolved relative to the calling file (`.ling` may be omitted); `function` - name; `args` - list of arguments (default none); `capabilities` - list from `fs db net p2p proc keys clipboard` (default none).
- **Returns:** `ok(handle)` or `bad(reason)`.
- **Note:** A child can never receive a capability its parent lacks; a builtin that needs a missing capability raises `capability "..." not granted`.
- **Note:** The returned handle is wrapped in `ok(...)`: unwrap with `match` before passing it to `thread_join`.

The examples in this group use this `worker.ling`:

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

```
bind start = do {
  bind r = thread_spawn("worker.ling", "square", [7], [])
  bind h = match r { ok(v) => v, bad(e) => e }
  println(thread_join(h))
}
```

```
OUTPUT
Ok(49)
```

`thread_join(handle)`

Waits for a thread to finish and returns its result.

- **Aliases:** none (English spelling only)
- **Parameters:** `handle` - thread handle (text).
- **Returns:** `ok(return_value)`, or `bad(error)` if the thread raised an error, panicked or was cancelled. A runtime error inside the thread never crashes the process.

```
fn unwrap(r) { match r { ok(v) => v, bad(e) => e } }
bind start = do {
  bind h = unwrap(thread_spawn("worker.ling", "square", [9], []))
  println(thread_join(h))
}
```

```
OUTPUT
Ok(81)
```

`thread_join_timeout(handle, milliseconds)`

Like `thread_join` but gives up after a time limit.

- **Aliases:** none (English spelling only)
- **Parameters:** `handle`; `milliseconds`.
- **Returns:** `ok(value)`, `bad("timeout")` or `bad(error)`.

```
fn unwrap(r) { match r { ok(v) => v, bad(e) => e } }
bind start = do {
  bind h = unwrap(thread_spawn("worker.ling", "square", [3], []))
  println(thread_join_timeout(h, 5000))
}
```

```
OUTPUT
Ok(9)
```

`thread_alive(handle)`

True while the thread is still running.

- **Aliases:** none (English spelling only)
- **Parameters:** `handle`.
- **Returns:** Boolean.

```
fn unwrap(r) { match r { ok(v) => v, bad(e) => e } }
bind start = do {
  bind h = unwrap(thread_spawn("worker.ling", "square", [3], []))
  thread_join(h)
  println(thread_alive(h))
}
```

```
OUTPUT
false
```

`thread_cancel(handle)`

Asks a thread to stop. Cancellation is cooperative: the thread aborts at its next loop iteration or call with `bad("cancelled")`.

- **Aliases:** none (English spelling only)
- **Parameters:** `handle`.
- **Returns:** True if the thread was running and not already cancelled.

```
fn unwrap(r) { match r { ok(v) => v, bad(e) => e } }
bind start = do {
  bind h = unwrap(thread_spawn("worker.ling", "square", [3], []))
  thread_join(h)
  println(thread_cancel(h))
}
```

```
OUTPUT
false
```

`thread_self()`

Handle string identifying the calling isolate.

- **Aliases:** none (English spelling only)
- **Parameters:** none.
- **Returns:** Text (32 hex characters).

```
bind start = do {
  println(len(thread_self()))
}
```

```
OUTPUT
32
```

`thread_cpus()`

Number of hardware threads available.

- **Aliases:** none (English spelling only)
- **Parameters:** none.
- **Returns:** Number >= 1.

```
bind start = do {
  println(thread_cpus() >= 1)
}
```

```
OUTPUT
true
```

`thread_caps()`

The capabilities of the calling isolate. The main program has all of them.

- **Aliases:** none (English spelling only)
- **Parameters:** none.
- **Returns:** Sorted list of capability names.

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

```
OUTPUT
[clipboard, db, fs, keys, net, p2p, proc]
```

`chan_new(capacity)`

Creates a bounded channel.

- **Aliases:** none (English spelling only)
- **Parameters:** `capacity` - whole number 1..65536.
- **Returns:** A two-element list `[send_handle, receive_handle]`.
- **Note:** `chan_new_secure(capacity)` is identical but zeroises string payloads after delivery.

```
bind start = do {
  bind ch = chan_new(8)
  println(len(ch))
  println(chan_len(ch[1]))
}
```

```
OUTPUT
2
0
```

`chan_send(send_handle, value)`

Sends a value, blocking while the channel is full.

- **Aliases:** none (English spelling only)
- **Parameters:** `send_handle`; `value` - any value except a function.
- **Returns:** `ok(())`, or `bad("closed")`.

- **Note:** `chan_try_send` never blocks and returns `bad("full")` instead.

```
bind start = do {
  bind ch = chan_new(2)
  println(chan_send(ch[0], "hello"))
  println(chan_try_send(ch[0], 2))
  println(chan_try_send(ch[0], 3))
}
```

```
OUTPUT
Ok(())
Ok(())
Err(full)
```

`chan_recv(receive_handle)`

Receives the next value, blocking until one arrives.

- **Aliases:** none (English spelling only)
- **Parameters:** `receive_handle`.
- **Returns:** `ok(value)`, or `bad("closed")` once the channel is closed and drained.
- **Note:** `chan_try_recv(h)` returns `bad("empty")` instead of blocking; `chan_recv_timeout(h, ms)` returns `bad("timeout")`.

```
fn unwrap(r) { match r { ok(v) => v, bad(e) => e } }
bind start = do {
  bind ch = chan_new(4)
  chan_send(ch[0], 42)
  println(chan_recv(ch[1]))
  println(chan_try_recv(ch[1]))
  println(chan_recv_timeout(ch[1], 20))
  chan_close(ch[0])
  println(chan_recv(ch[1]))
}
```

```
OUTPUT
Ok(42)
Err(empty)
Err(timeout)
Err(closed)
```

`chan_select(receive_handles, timeout_ms)`

Waits on several channels at once.

- **Aliases:** none (English spelling only)
- **Parameters:** `receive_handles` - list of receive handles; `timeout_ms`.
- **Returns:** `ok([index, value])` for the first channel to deliver, `bad("timeout")`, or `bad("closed")` when all are closed.

```
bind start = do {
  bind a = chan_new(2)
  bind b = chan_new(2)
  chan_send(b[0], "from b")
  println(chan_select([a[1], b[1]], 100))
}
```

```
OUTPUT
Ok([1, from b])
```

`chan_close(handle)`

Closes a channel end. Closing the send side ends the channel once its queue is drained.

- **Aliases:** none (English spelling only)
- **Parameters:** `handle` - send or receive handle.
- **Returns:** Boolean.

```
bind start = do {
  bind ch = chan_new(1)
  println(chan_close(ch[0]))
  println(chan_send(ch[0], 1))
}
```

```
OUTPUT
true
Err(closed)
```

`chan_len(handle)`

Number of messages queued.

- **Aliases:** none (English spelling only)
- **Parameters:** `handle`.
- **Returns:** Number.

```
bind start = do {
  bind ch = chan_new(4)
  chan_send(ch[0], 1)
  chan_send(ch[0], 2)
  println(chan_len(ch[1]))
}
```

```
OUTPUT
2
```

`shared_put(key, value)`

Stores a value in the process-wide write-once store.

- **Aliases:** none (English spelling only)
- **Parameters:** `key` - text; `value` - frozen deep copy.
- **Returns:** `ok(())`, or `bad("exists")` - a key can never be overwritten.

```ling
bind start = do {
  println(shared_put("config", [1, 2, 3]))
  println(shared_put("config", 0))
}
```

```
OUTPUT
Ok(())
Err(exists)
```

`shared_get(key, default)`

Reads a value from the shared store.

- **Aliases:** none (English spelling only)
- **Parameters:** `key`; `default` returned if absent.
- **Returns:** The stored value or the default.

```ling
bind start = do {
  shared_put("answer", 42)
  println(shared_get("answer", 0))
  println(shared_get("missing", "dflt"))
}
```

```
OUTPUT
42
dflt
```

`Producer / consumer pattern`

A worker streams values through a channel while the main program consumes them (an idiom, not a builtin).

- **Aliases:** not a builtin

```
fn unwrap(r) { match r { ok(v) => v, bad(e) => e } }
bind start = do {
  bind ch = chan_new(4)
  bind p = unwrap(thread_spawn("worker.ling", "producer", [ch[0], 3], []))
  println(unwrap(chan_recv(ch[1])))
  println(unwrap(chan_recv(ch[1])))
  println(unwrap(chan_recv(ch[1])))
  println(chan_recv(ch[1]))
  println(thread_join(p))
}
```

```
OUTPUT
0
10
20
Err(closed)
Ok(3)
```

Compatibility stubs

These names exist for source compatibility with older sample programs. They are **not** functional and should not be used in new code.

Name	Summary
`Timer::now()`	Stub: always returns `1000`
`Timer::hour()`	Stub: always returns `14`
`Flow::parallel(closure)`	Stub: runs the closure immediately on the calling thread (no parallelism)
`Vec::from(list)`	Returns the list unchanged; with several non-list arguments, collects them into a list

`Timer::now()`

Stub: always returns `1000`. Use `time_now()` for real time.

- **Aliases:** zh: `计时::现在` (further languages exist)
- **Parameters:** none.

- **Returns:** Number `1000`.

```
bind start = do {
  println(Timer::now())
}
```

```
OUTPUT
1000
```

`Timer::hour()`

Stub: always returns `14`.

- **Aliases:** zh: `计时::获取当前小时` (further languages exist)
- **Parameters:** none.
- **Returns:** Number `14`.

```
bind start = do {
  println(Timer::hour())
}
```

```
OUTPUT
14
```

`Flow::parallel(closure)`

Stub: runs the closure immediately on the calling thread (no parallelism). Use the `thread_*` family for real concurrency.

- **Aliases:** zh: `流水::并行` (further languages exist)
- **Parameters:** `closure`.
- **Returns:** The closure result.
- **Note:** Interpreter only.

INTERPRETER ONLY

```
bind start = do {
  println(Flow::parallel(| | 6 * 7))
}
```

OUTPUT

```
42
```

Run this example with `ling run --interp`; the default JIT backend prints:

OUTPUT

```
()
```

`Vec::from(list)`

Returns the list unchanged; with several non-list arguments, collects them into a list.

- **Aliases:** zh: `向量::从` (further languages exist)
- **Parameters:** `list` or values.
- **Returns:** List.

```
bind start = do {
  println(Vec::from([1, 2]))
  println(Vec::from(7, 8))
}
```

OUTPUT

```
[1, 2]
[7, 8]
```

CHAPTER 4

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] 
ling                     (shorthand for: ling run )
ling build  [--out dir] [--platform list] [--icon file] [--pack] [--aot]
ling ast [path] [--technical|--artwork|--ling|--all] [--out dir]
ling visualize 
ling convert  [-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: ` and status 1; no file at all prints `Usage: ling run [--wasm|--interp|--O0|--O1|--O2|--O3] ` 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  [--out dir] [--platform list] [--icon file] [--
pack] [--aot]
```

Option	Meaning
``	A single `.ling` file, or a directory. A directory is read through its manifest (`灵符.toml` , `ลิง ฟู.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 `	Output folder. Default `dist` ; each platform is written to a sub-folder such as `dist/web` or `dist/windows` .
`--platform `	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 `	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.]` 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`)	`#` `#.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 `/` 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.]`	-	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 `	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 ` 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  [-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  [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 `	`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 `	`install`, `include`	Install a package from the registry
`search `	`find`, `seek`	Search the registry
`publish`	`release`, `share`	Publish the project
`login `	`sign-in`, `auth`	Save registry credentials
`logout`	`sign-out`, `deauth`	Forget credentials
`translate `	`trans`, `convert`	Translate keywords between languages (see warning)
`manifest`	`show`, `info`	Print the manifest
`wizard`	`guide`, `create-wizard`	Interactive project creator
`normalize `	`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 ` (scaffolding language: `en`, `zh`, `ja`, `ko`, `th`, `ru`), `--type` / `-t `, `--root-lang `, `--crates`, `--path `, `--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  [--type bin|lib|game|ui|ai|crypto|web|polyglot] [--lang
en|zh|ja|ko|th|ru] [--root-lang ]
```

Creates `/` 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`, `/`, `/`, `/main.ling`, `/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  --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 ` 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 `/` 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 check`, `lingfu fmt`, `lingfu translate`

These commands exist but are **thin wrappers around `ling` subcommands that do not exist** in `ling` 2030.1.53: they run `ling check `, `ling fmt ` and `ling   `, and `ling` answers all three by printing its usage text and exiting successfully. The result is that `lingfu check` and `lingfu fmt` report success (`Formatted!`, or a count of "checked" files) without examining or changing anything, and `lingfu translate` prints the usage text. Do not rely on them. To change the language of a project use `lingfu normalize`.

`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 doc`, `bench`, `update`

- `doc` runs `mdbook build` when `docs/book.toml` or `book/book.toml` exists, otherwise `cargo doc --no-deps`.

- `bench` runs `cargo bench` and `update` runs `cargo update`, forwarding extra arguments. They are only useful in projects that also contain a Rust crate.

`lingfu normalize`

```
OUTPUT
lingfu normalize  [--dry-run|-n] [--content-only] [--files-only]
lingfu normalize --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. `` 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 `	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 `	Searches the registry. `--crates` searches crates.io through `cargo search` instead.
`add `	Downloads a package and unpacks it into `./ling_modules//`, then prints the file to import: `use "ling_modules//";`. `--crates` shells out to `cargo add` for Rust dependencies.

`lingfu seed`

Creates a reconstructable, encrypted integrity fingerprint of the project: the BLAKE3 checksum of every git-tracked file (minus `ling.ignore` patterns) rolled into a Merkle root, the manifest encrypted with a key derived from a key file, and the result written as a playable `.wav` file (`seed.wav` by default). `--check` recomputes the fingerprint and reports drift against a saved seed, and `--recover-init` additionally splits the key into three Shamir shares (any two recover it). It requires a git repository. Lose the key file without recovery shares and the seed cannot be regenerated.

`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: ` for failures such as `build failed`, `tests failed`, `run failed`).

CHAPTER 5

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.

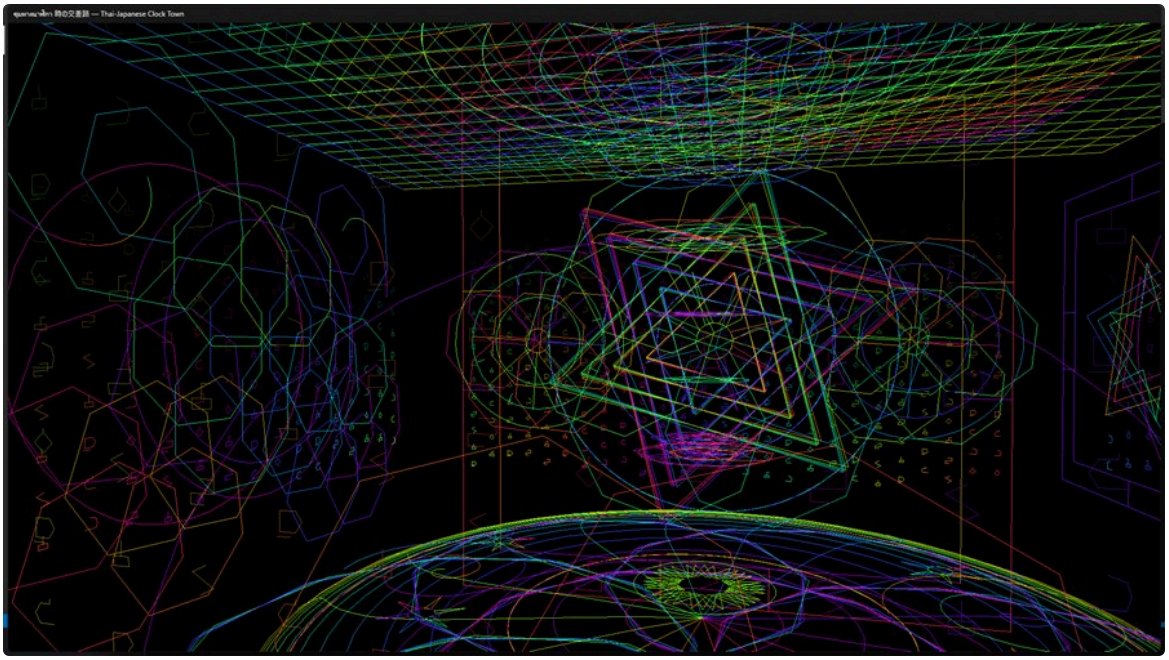


Figure 10.1. A scene built only from the builtins in this chapter and the next, drawn live in a Ling window. Source: `examples/3d-rooms/3d-thai-japanese-clocktown.ling` *in the Ling repository.*

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 Result {
    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 = (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`.
- `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 = (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` 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)` 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`.
- 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  Option`, `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 {
    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.

```rust
use ling_mic::{input_devices, MicConfig, MicInput};

// Compiled only: running it would open the real microphone.
#[allow(dead_code)]
fn listen() -> Result {
    let mic = MicInput::open(MicConfig { sample_rate: 48_000, ..MicConfig::default()
})?;
    mic.start(|samples| {
        let rms = (samples.iter().map(|s| s * s).sum::() / 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))
  }
}
```

CHAPTER 14

Networking and the Web

This chapter covers everything in Ling that talks over a network: the line-oriented TCP lobby (`net_*`), SSH (`ssh_*`), the post-quantum `lingtp://` transport, the peer-to-peer transport (`p2p_*`), the HTTP/HTTPS client and server (`http_*`), SQLite (`db_*`), request helpers, and the three Rust crates behind them (`ling-net`, `ling-http`, `ling-web`). It ends with a complete JSON-over-HTTP service, once in Ling and once in Rust.

What runs where

Capability	Ling builtins	Rust crate	Notes
Game/LAN lobby over TCP	`net_*`	(none; `src/runtime/net.rs`)	Plain text, no authentication.
SSH client and server	`ssh_*`	(none; `src/runtime/ssh.rs`, built on `russh`)	Password authentication only on the server.
Post-quantum secure fetch	`lingtp_*`	`ling-net` feature `lingtp`	Separate implementation from the Ling runtime; see the warning in that section.
Peer-to-peer transport (LWP-1)	`p2p_*`	`ling-net` feature `lwp`	Needs the `"p2p"` capability.
HTTP server and client	`http_*`	`ling-http`, `ling-net` feature `http`	Server builtins need a build with the `web` feature.
SQLite	`db_*`	`ling-http` (`rusqlite`)	One connection per interpreter.
Flask-style web apps	(Rust only)	`ling-web`	Sessions, CSRF, flash messages, templates.

Conventions and gotchas

- **Builtins are blocking or polling, never callbacks.** `net_recv()` returns `""` when nothing is queued. You write a loop with `sleep(ms)` and check again.
- **Handles and statuses are numbers.** A status of `0` means idle or failed, and failures to bind or connect are reported only through the status, not by an exception.

- **Misspelt names are silent.** An unknown function evaluates to `()`. If `net_hots(7777)` "does nothing", that is why.
- **`&&` and `||`, not `and` and `or`.** `while a && b { ... }` is the condition syntax. (This is the language rule, repeated here because network loops use compound conditions constantly.)
- **`for` loops do not update outer bindings.** A `bind` inside a `for ... in ... { }` body creates a new variable that disappears at the end of the iteration. Accumulate results with a `while` loop and an index, as the examples below do.
- **Server builtins block.** `http_serve`, `http_serve_tls`, `http_serve_dev_tls` and `lingtp_serve` run the request loop on the calling thread and return only when the server stops. Register routes first.
- **The `web` feature.** `http_*`, `db_*`, `oauth_google_*`, `totp_secret` and the other web helpers are compiled into `ling` only with the Cargo feature `web` (`cargo build --features web`), which is not a default feature. The `ling` binary used to test this chapter (version 2030.1.53) was built with it. To check yours, run `db_open(":memory:")` and then print `db_exec("CREATE TABLE t(a)")`: a web-enabled build prints `0`, and a build without it prints `()`, because the names simply do not exist and unknown names evaluate to `()` silently. The `web` feature also turns on `video`.
- **Aliases.** Each entry lists aliases for Chinese, Japanese, Korean and Thai when the interpreter defines them. Many network builtins have only Thai aliases, and several (`net_clients`, `ssh_*`, `lingtp_*`, `p2p_*`) have none. The localized names that appear in `lexicons/*.ling` for `http_get` (such as `网取`) are **not** wired into the interpreter and evaluate to `()`.

Lobby networking: `net_*`

`net_*` is a tiny, dependable transport for game state and LAN tools. One process is either a **host** that accepts up to 64 clients, or a **client** connected to exactly one host. The wire format is one UTF-8 line per message. Socket I/O runs on background threads, so a game loop never blocks; `net_send` queues a line and `net_recv` dequeues one.

There is exactly one role per process. Calling `net_host` or `net_join` replaces the previous role.

Security. The host listens on `0.0.0.0`, the traffic is plain text and there is no authentication or encryption. Use it on trusted LANs or inside a tunnel. For anything that crosses the internet, use `lingtp_*`, `ssh_*` or `http_serve_tls`. Messages must not contain newline characters: an embedded newline is sent as two messages. Each inbound queue holds 4096 lines; when full, the oldest are dropped.

`net_recv()` and `net_recv_from()` drain the same queue. Choose one style per program.

Builtin	Purpose
`net_host`	Listen for clients on a TCP port and become the host.
`net_join`	Connect to a host and become a client.
`net_send`	Send one line. On a host, broadcast to every client; on a client, send to the host.
`net_send_to`	Host only: send one line to a single client.
`net_recv`	Take the oldest queued message.
`net_recv_from`	Like `net_recv` but prefixes the sender.
`net_clients`	Host: the ids of connected clients.
`net_client_count`	How many peers are connected.
`net_events`	Drain the connection event log.
`net_status`	Current state of the network role.
`net_close`	Tear down the current host or client and return to idle.
`net_announce`	Host: broadcast a lobby description on the LAN about once per second.
`net_announce_stop`	Stop broadcasting.
`net_discover`	Client: list the hosts heard announcing in the last 5 seconds.
`net_test`	Host self-test: can the TCP port be bound, and what is this machine's LAN address?

`net_host(port)`

Listen for clients on a TCP port and become the host.

- **Aliases:** th: `เน็ตโฮสต์`
- **Parameters:**
- `port` (number, default 7777).
- **Returns:** Nothing. Poll `net_status()` for the result.
- **Note:** If the port cannot be bound, the status silently returns to `0`. Accepts up to 64 clients; further connections are dropped.

```
GUI
bind start = do {
  net_host(7777)
  bind n = 0
  while net_status() == 1 && n  HttpResult {
        state.db.with_conn(|conn| {
```

```
            let mut stmt = conn.prepare("SELECT id, title, body, created_at FROM notes
ORDER BY id")?;
            let rows = stmt.query_map([], row_to_note)?;
            rows.collect()
        }).await
    }

    async fn create(state: &AppState, body: CreateNote) -> HttpResult {
        state.db.with_conn(move |conn| {
            conn.query_row(
                "INSERT INTO notes (title, body) VALUES (?1, ?2) RETURNING id, title,
body, created_at",
                params![body.title, body.body],
                row_to_note,
            )
        }).await
    }

    async fn show(state: &AppState, id: i64) -> HttpResult {
        state.db.with_conn(move |conn| {
            conn.query_row(
                "SELECT id, title, body, created_at FROM notes WHERE id = ?1",
                params![id],
                row_to_note,
            )
        }).await
    }

    async fn update(state: &AppState, id: i64, body: UpdateNote) -> HttpResult {
        let existing = Self::show(state, id).await?;
        let title = body.title.unwrap_or(existing.title);
        let note_body = body.body.unwrap_or(existing.body);
        state.db.with_conn(move |conn| {
            conn.query_row(
                "UPDATE notes SET title = ?1, body = ?2 WHERE id = ?3 \
                 RETURNING id, title, body, created_at",
                params![title, note_body, id],
                row_to_note,
            )
```

```
        }).await
    }

    async fn destroy(state: &AppState, id: i64) -> HttpResult {
        let deleted = state.db
            .with_conn(move |conn| conn.execute("DELETE FROM notes WHERE id = ?1",
params![id]))
            .await?;
        if deleted == 0 {
            return Err(HttpError::NotFound);
        }
        Ok(())
    }
}

#[tokio::main]
async fn main() -> anyhow::Result {
    let db = Db::connect_memory().await?;          // or Db::connect("app.db").await?
    db.run_migrations(MIGRATIONS).await?;
    let router: Router = App::new()
        .merge(Router::new().route("/health", get(|| async { Json(serde_json::json!
({"status": "ok"})) })))
        .merge(ling_http::resource::("/notes"))
        .with_state(AppState { db });
    ling_http::serve_http(router, "127.0.0.1:6666".parse()?).await?;
    Ok(())
}
```

The shipped example was compiled and run on loopback. A session against it:

```
OUTPUT
$ curl -s http://127.0.0.1:18091/health
{"service":"ling-http/hello","status":"ok"}
$ curl -s -X POST http://127.0.0.1:18091/notes -H 'content-type: application/json' -d
'{"title":"hi","body":"first note"}'
{"id":1,"title":"hi","body":"first note","created_at":"2026-10-28 23:33:01"}
$ curl -s http://127.0.0.1:18091/notes
[{"id":1,"title":"hi","body":"first note","created_at":"2026-10-28 23:33:01"}]
$ curl -s -X PUT http://127.0.0.1:18091/notes/1 -H 'content-type: application/json' -d
'{"title":"edited"}'
{"id":1,"title":"edited","body":"first note","created_at":"2026-10-28 23:33:01"}
$ curl -s -i -X DELETE http://127.0.0.1:18091/notes/1 | head -1
HTTP/1.1 204 No Content
$ curl -s http://127.0.0.1:18091/notes/99
{"error":"database error: Query returned no rows"}
$ curl -s -X POST http://127.0.0.1:18091/notes -H 'content-type: application/json' -d
'{"body":"no title"}'
Failed to deserialize the JSON body into the target type: missing field `title` at line
1 column 19
```

Note the last response: a body that fails to deserialize is rejected by `axum` before your code runs, with a plain-text message rather than the JSON error shape.

`ling-web`

灵网站, the Flask-flavoured layer on `ling-http`: server-side sessions in SQLite, flash messages, CSRF tokens, Jinja-style templates (`minijinja`), static files and blueprints.

Item	Purpose
`session_layer(&db, secure)`	Middleware for SQLite-backed sessions. The cookie is `ling_session`, `HttpOnly`, `SameSite=Lax`, with a 7-day sliding expiry; the session data lives on the server, so deleting the row revokes it. Pass `secure = true` whenever the site is served over HTTPS.
`Session`	The `tower_sessions::Session` extractor (`get`, `insert`, `remove`).
`csrf::csrf_token(&session)`	Returns (creating on first use) the session's 256-bit token. Put it in a hidden form field.
`csrf::verify_csrf(&session, submitted)`	Constant-time comparison; `false` for a missing or mismatched token.

Item	Purpose
`flash::flash(&session, category, message)`	Queue a one-time message.
`flash::take_flashes(&session)`	Pop all queued `Flash { category, message }` values.
`Templates::load(dir)` and `.render(name, context)`	Load templates from a directory; render to `Html` using `minijinja::context!`.
`serve_static(prefix, dir)`	Mount a directory of static files.
`blueprint(prefix, router)`	Group routes under a prefix (`Router::nest`).

Using it from Ling

`ling-web` has no Ling builtins; Ling programs get the equivalent pieces from `cookie_get`, `random_hex`, `html_escape` and `http_static`, and build sessions with `db_*`. The Rust crate is for programs written in Rust.

Using it from Rust: the guestbook

`crates/ling-web/examples/guestbook.rs` shows sessions, CSRF-protected forms, flash messages and templates together. The handler for `GET /` counts visits in the session, issues a CSRF token and renders a template; the handler for `POST /` verifies the token before acting.

```
async fn sign(
    State(state): State,
    session: Session,
    Form(form): Form,          // SignForm { name, csrf_token }
) -> Result {
    let ok = ling_web::csrf::verify_csrf(&session, &form.csrf_token)
        .await.map_err(HttpError::Internal)?;
    if !ok {
        ling_web::flash::flash(&session, "error", "That form expired - try again.")
            .await.map_err(HttpError::Internal)?;
        return Ok(Redirect::to("/"));
    }
    state.entries.lock().unwrap().push(form.name.clone());
    ling_web::flash::flash(&session, "success", format!("Thanks, {}!", form.name))
        .await.map_err(HttpError::Internal)?;
    Ok(Redirect::to("/"))
}

#[tokio::main]
async fn main() -> anyhow::Result {
    let db = ling_http::Db::connect_memory().await?;
    let sessions = ling_web::session_layer(&db, /* secure = */ false).await?;
    let templates = Templates::load("examples/templates")?;
    let state = AppState { templates, entries: Default::default() };

    let router: Router = Router::new()
        .route("/", get(index).post(sign))
        .merge(ling_web::serve_static("/static", "examples/static"))
        .with_state(state)
        .layer(sessions);                 // the session layer must wrap the routes

    ling_http::serve_http(router, "127.0.0.1:7777".parse()?).await?;
    Ok(())
}
```

The template uses ordinary Jinja syntax; `{{ value }}` is HTML-escaped automatically:

```
OUTPUT

  
  
  Sign the guestbook

{% for f in flashes %}{{ f.message }}{% endfor %}
```

A cookie-jar session against the compiled example, where the token was read from the first page:

```
OUTPUT
GET /                  ->  "You've visited 1 time(s) this session."  (hidden csrf_token
field present)
POST / name=Ada&csrf_token=   ->  303 See Other, location: /,
                                         set-cookie: ling_session=...; HttpOnly;
SameSite=Lax; Path=/; Max-Age=604800
GET /                  ->  "visited 3 time(s)", flash "Thanks, Ada!", entry
Ada
POST / name=Eve&csrf_token=bad  ->  redirected; next page shows the flash "That form
expired - try again." and no Eve
```

Building an HTTPS, JSON and SQLite service in Ling

This is the whole of a small service: create and list notes, fetch one by id, HTML output with escaping, SQLite persistence, JSON responses with proper status codes, and the port taken from the command line. It was run on loopback and exercised with `curl`; transcript below.

Design points, in the order you will meet them:

- **Input.** There is no JSON parser, so clients post `application/x-www-form-urlencoded` bodies and the handler reads fields with `query_param(req.body, ...)`. Identifiers travel in the query string.
- **Output.** JSON is concatenated. Every string goes through `json_escape`; numbers from the database are already numbers.
- **Status codes.** A handler returns a `Response` struct to set `201`, `400` or `404`.
- **SQL.** Parameters only. No string is ever pasted into a query.

- **Loops.** The list handler builds the array with a `while` loop and an index, because a `for` body cannot update an outer binding.
- **Escaping HTML.** The greeting page passes the user-supplied name through `html_escape`.

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

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

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

fn list_notes(req) {
  bind rows = db_query("SELECT id, title, body FROM notes ORDER BY id")
  bind out = "["
  bind i = 0
  while i  0 { bind out = out + "," }
    bind out = out + row_json(rows[i])
    bind i = i + 1
  }
  return json(200, out + "]")
}

fn create_note(req) {
  bind title = query_param(req.body, "title", "")
  bind body = query_param(req.body, "body", "")
  if title == "" {
    return json(400, "{\"error\":\"title is required\"}")
  }
  db_exec("INSERT INTO notes(title, body) VALUES (?1, ?2)", title, body)
  bind rows = db_query("SELECT id, title, body FROM notes ORDER BY id DESC LIMIT 1")
  return json(201, row_json(rows[0]))
}

fn show_note(req) {
  bind id = to_number(query_param(req.query, "id", "0"))
  bind rows = db_query("SELECT id, title, body FROM notes WHERE id = ?1", id)
```

```
  if len(rows) == 0 {
    return json(404, "{\"error\":\"not found\"}")
  }
  return json(200, row_json(rows[0]))
}

fn page(req) {
  bind name = html_escape(query_param(req.query, "name", "world"))
  return "Hello, " + name + ""
}

bind start = do {
  db_open(":memory:")
  db_exec("CREATE TABLE notes(id INTEGER PRIMARY KEY AUTOINCREMENT, title TEXT NOT
NULL, body TEXT NOT NULL DEFAULT '')")
  http_route("GET", "/", page)
  http_route("GET", "/notes", list_notes)
  http_route("GET", "/note", show_note)
  http_route("POST", "/notes", create_note)
  http_serve("127.0.0.1", to_number(cli_arg("port", "18093")))
}
```

Run with `ling run notes.ling --port 8090`. Transcript from the tested version (port 18093):

```
OUTPUT
$ curl -s "http://127.0.0.1:18093/?name=%3Cb%3EAda%3C/b%3E"
Hello, &lt;b&gt;Ada&lt;/b&gt;
$ curl -s -i -X POST -d 'title=Hello "quoted"&body=line' http://127.0.0.1:18093/notes
HTTP/1.1 201 Created
{"id":1,"title":"Hello \"quoted\"","body":"line"}
$ curl -s -X POST -d 'title=Second&body=x%20y' http://127.0.0.1:18093/notes
{"id":2,"title":"Second","body":"x y"}
$ curl -s -i -X POST -d 'body=nothing' http://127.0.0.1:18093/notes
HTTP/1.1 400 Bad Request
{"error":"title is required"}
$ curl -s http://127.0.0.1:18093/notes
[{"id":1,"title":"Hello \"quoted\"","body":"line"},{"id":2,"title":"Second","body":"x
y"}]
$ curl -s "http://127.0.0.1:18093/note?id=2"
{"id":2,"title":"Second","body":"x y"}
$ curl -s -i "http://127.0.0.1:18093/note?id=9"
HTTP/1.1 404 Not Found
{"error":"not found"}
```

Going to HTTPS

Replace the last line of `start` with `http_serve_tls("0.0.0.0", 443, "/etc/letsencrypt/live/example.org/fullchain.pem", "/etc/letsencrypt/live/example.org/privkey.pem")` for a public deployment, or `http_serve_dev_tls("127.0.0.1", 8443)` for local development. A tested dev-TLS server answered `curl -k https://127.0.0.1:18094/login` with `HTTP/2 303`, a `set-cookie` header and `location: /me`, and served `http_static` files over the same listener. When you set a session cookie over HTTPS, add `Secure; HttpOnly; SameSite=Lax; Path=/` yourself, as in the `Response` example earlier. Behind a reverse proxy, read the visitor address from `req.client_ip` only if the proxy is the sole way in and it overwrites `X-Forwarded-For`.

Production checklist

1. Terminate TLS in the process (`http_serve_tls`) or at a proxy; never serve credentials over plain HTTP.
2. Bind to `127.0.0.1` unless remote access is intended.

3. Pass every user value to SQL as a parameter and to HTML through `html_escape` and to JSON through `json_escape`.
4. Hash passwords with `password_hash`; compare with `password_verify`. Create session identifiers with `random_hex(16)` or longer and store them in SQLite with an expiry.
5. Rate-limit by account name or API key as well as `client_ip`; the IP header is spoofable without a trusted proxy.
6. Remember the handler loop is single-threaded: do no slow work in a handler, and use `http_post_async` for outbound calls.
7. Do not let errors escape handlers; the 500 response body contains the internal error text.
8. Back up the SQLite file and the `.lingtp` and `.ling-ssh` host keys.

CHAPTER 15

Cryptography

This chapter covers the cryptographic builtins that ship in the `ling` runtime and the `ling-crypto` Rust crate they are built on. The builtins are a thin, string-oriented layer: bytes cross the language boundary as **lowercase hex strings**, long-lived secrets are kept inside the interpreter behind **handles**, and the heavy lifting is done by vetted Rust implementations (RustCrypto, dalek, BLAKE3, `ml-kem`, `ml-dsa`, Argon2).

Read the security guidance first. A few of the builtins have defaults that are convenient for demos and unsafe for production, and the book would rather you learn that here than from an incident report.

Conventions

- **Hex in, hex out.** Keys, nonces, ciphertexts, digests and signatures are lowercase hex strings. A 32-byte key is 64 hex characters.
- **Handles.** `ed25519_keygen`, `mldsa_keygen`, `mldsa87_keygen` and `knot_keygen` return a small integer that indexes a per-interpreter key table. The `lw_*` family returns an unguessable 32-character hex handle instead. Secret key bytes never reach Ling code.
- **Errors.** Builtins that can fail on bad input return a result value. Print it and you see `Ok(...)` or `Err(message)`; destructure it with `match r { ok(v) => ..., bad(e) => ... }`. Plain-valued builtins either return an empty string (`""`) or raise a runtime error; each entry says which.
- **Unknown names are silent.** Calling a function that does not exist, including a misspelt builtin, evaluates to `()` and does not raise an error. If a crypto call "does nothing", check the spelling and read the builtin list below.
- **Capabilities.** On the main thread every capability is granted. Inside a restricted isolate the key-holding `lw_*` builtins require the `"keys"` capability, and `p2p_*` (Chapter 30) requires `"p2p"`.
- **Aliases.** Every entry lists the localized names that exist in the interpreter's match table. Other scripts (Arabic, Hebrew, Persian, Urdu, Russian, French, German) have aliases too; see the keyword reference.

Security guidance for professional use

1. **Never invent cryptography.** Use the vetted primitives below and the highest-level builtin that does the job. For encrypting a message with a key you already hold, that is `crypto_seal`. For storing passwords, that is `password_hash`. Do not build your own scheme from `hash_hex` and XOR.
2. **Keys must be 32 bytes of real entropy.** `crypto_seal`, `crypto_open`, `ed25519_keygen_from_seed` and the other seeded builtins decode their hex argument leniently: a short key is **zero-padded**, a long one is truncated, and a pair of characters that is not valid hex is **silently dropped**. A typo therefore yields a weak key, not an error. Generate keys with `random_hex(32)`. Derive a key from a passphrase with `lw_password_key` (Argon2id), never by hashing the passphrase once.
3. `password_hash` **is for verification,** `lw_password_key` **is for keys.** `password_hash` produces a self-describing PHC string (`$argon2id$v=19$m=65536,t=3,p=4$...`) with a random salt and 64 MiB of memory. Use it for login databases. Use `lw_password_key` when you need 32 bytes of key material from a passphrase.
4. **Nonces.** `crypto_seal` draws a fresh 24-byte random nonce for every call, which is safe for XChaCha20-Poly1305 without bookkeeping. `lw_aead_seal` takes the nonce from you. Never reuse a `(key, nonce)` pair; take 24 bytes from `random_hex(24)` for each message.
5. **Do not use the game RNG for secrets.** `rand` and friends are for game logic. Use `random_hex` or `lw_random`, which draw from the operating-system CSPRNG.
6. **Strings are not zeroized.** `lw_*` secret keys are wiped when dropped (`lw_key_drop`), and the Rust key types zeroize themselves. A hex secret that you copy into a Ling variable lives as long as that string does. Keep secrets in handles where the API offers them.
7. **Knot keys cannot be saved.** `knot_keygen` has no seed argument and no export. The private key exists only for the life of the process. For a persistent hybrid KEM identity use `lw_kem_keypair`, which takes a 96-byte seed, or the `HybridKeypair` / `MlKem768Keypair::from_seed` APIs in Rust.
8. **Comparisons.** `==` on strings is not constant time. Use the verification builtins (`ed25519_verify`, `mldsa87_verify`, `password_verify`, `totp_verify`) rather than comparing MACs or signatures yourself.
9. `crypto_hash` **is not plain SHA3-256.** It prepends the label `ling-geo-holo-hash-v1`, so its output will not match any other SHA3 implementation. Use `sha256_hex` or `hash_hex` (BLAKE3) when you must interoperate. See the next section.
0. **Treat the Ling-specific constructions as unaudited.** The "geometric" layer (`knot_points`, `holo_points`, `crypto_hash`), the DICE-42 cascade used inside `lingtp://`, `MandalaHash`, and the LRS-87 ring signature (`lw_ring_sign`) are original designs layered over standard primitives. The source labels LRS-87 as experimental, and this repository contains no

third-party audit report for any of them. Their security rests on the vetted primitives underneath, never on the geometry. For regulated or high-value deployments, use the standard primitives directly and commission your own review.

1. **Post-quantum is hybrid.** The KEM builtins combine X25519 with ML-KEM so that the result stays secure if either half is broken. ML-KEM decapsulation has *implicit rejection*: a tampered ciphertext yields a different secret instead of an error, so always confirm the shared secret by using it to authenticate something (as the `lingtp` handshake does with its Finished messages).
2. **TOTP is SHA-1, six digits, 30 seconds.** That matches Google Authenticator and Authy. `totp_verify` accepts one step either side and does not track replay; keep the last accepted counter in your database.

Hashing and digests

Builtin	Purpose
`sha256_hex`	SHA-256 of a string, as 64 hex characters.
`hash_hex`	BLAKE3 hash of a string, as 64 hex characters.
`file_hash`	BLAKE3 hash of a file's raw bytes, as 64 hex characters.
`crypto_hash`	Domain-separated SHA3-256 (the "holographic hash") of a string.
`shake_hex`	SHAKE-256 extendable-output hash, squeezed to `len` bytes.
`hash_str`	Fast non-cryptographic FNV-1a hash, normalised to a number in the range 0 to 1.
`hash_int`	FNV-1a hash of a string reduced modulo `n`.
`random_hex`	Cryptographically secure random bytes from the operating system, as hex.

`sha256_hex(text)`

SHA-256 of a string, as 64 hex characters.

- **Aliases:** zh: `SHA256哈希`
- **Parameters:**
- `text` (string): the UTF-8 bytes to hash.
- **Returns:** 64-character lowercase hex string.
- **Note:** Matches the browser's `crypto.subtle.digest("SHA-256", ...)`, which is why proof-of-work challenges use it.

```
bind start = do {
  print(sha256_hex("abc"))
}
```

```
OUTPUT
ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
```

`hash_hex(text)`

BLAKE3 hash of a string, as 64 hex characters.

- **Aliases:** th: `แฮชสตริง`
- **Parameters:**
- `text` (string): the UTF-8 bytes to hash.
- **Returns:** 64-character lowercase hex string (32-byte digest).
- **Note:** BLAKE3 is fast and collision resistant. It is not a password hash.

```
bind start = do {
  print(hash_hex("abc"))
}
```

```
OUTPUT
6437b3ac38465133ffb63b75273a8db548c558465d79db03fd359c6cd5bd9d85
```

`file_hash(path)`

BLAKE3 hash of a file's raw bytes, as 64 hex characters.

- **Aliases:** th: `แฮชไฟล์`
- **Parameters:**
- `path` (string): file to read. The whole file is read into memory.
- **Returns:** 64-character hex digest, or `""` if the file cannot be read.
- **Note:** Binary safe, unlike `read_file`. Use it for content fingerprints and integrity checks.

```
bind start = do {
  write_file("hf.txt", "hello file")
  print(file_hash("hf.txt"))
  print(file_hash("hf.txt") == hash_hex("hello file"))
  print(file_hash("no_such_file") == "")
  remove_file("hf.txt")
}
```

```
OUTPUT
84c9e992e61433f9a18d396b2062eacd7341c9669dfa8bbbc35a28ec54597cdb
true
true
```

`crypto_hash(text)`

Domain-separated SHA3-256 (the "holographic hash") of a string.

- **Aliases:** zh: `几何哈希` ; ja: `幾何ハッシュ` ; ko: `기하해시` ; th: `แฮชเข้ารหัส`
- **Parameters:**
- `text` (string): the data to hash.
- **Returns:** 64-character hex digest.
- **Note:** Computes `SHA3-256("ling-geo-holo-hash-v1" || text)`. The label means the result differs from every other SHA3-256 tool. It is the digest that `knot_points` and `knot_label` visualise.

```
bind start = do {
  print(crypto_hash("abc"))
  print(crypto_hash("abc") == crypto_hash("abc"))
}
```

```
OUTPUT
0f5ae6a06fa73d3e18615b33ee9a9efafdb1a25a7bbf5f3ca9c4d7f1ac4ddfb9
true
```

`shake_hex(text, len)`

SHAKE-256 extendable-output hash, squeezed to `len` bytes.

- **Aliases:** zh: `SHAKE哈希`
- **Parameters:**
- `text` (string): input.
- `len` (number, default 32): output length in bytes. Negative values become 0.
- **Returns:** Hex string of length `2 * len`.

```
bind start = do {
  print(shake_hex("abc", 16))
  print(len(shake_hex("abc", 64)) / 2)
}
```

```
OUTPUT
483366601360a8771c6863080cc4114d
64
```

`hash_str(text)`

Fast non-cryptographic FNV-1a hash, normalised to a number in the range 0 to 1.

- **Aliases:** th: `แฮช`
- **Parameters:**
- `text` (string).
- **Returns:** Number between 0.0 and 1.0.
- **Note:** Deterministic, for seeds, colours and bucketing. **Not secure**; never use it for anything an attacker can influence.

```
bind start = do {
  print(hash_str("abc"))
}
```

```
OUTPUT
0.2552382502101809
```

`hash_int(text, n)`

FNV-1a hash of a string reduced modulo `n`.

- **Aliases:** th: `แฮชจำนวน`
- **Parameters:**
- `text` (string).
- `n` (number, default 100): modulus, at least 1.
- **Returns:** Whole number from 0 to `n - 1`.
- **Note:** Same warning as `hash_str`: non-cryptographic.

```
bind start = do {
  print(hash_int("player-7", 10))
}
```

```
OUTPUT
0
```

`random_hex(n)`

Cryptographically secure random bytes from the operating system, as hex.

- **Aliases:** ko: `무작위16진수`
- **Parameters:**
- `n` (number, default 16): how many bytes. Negative values become 0.
- **Returns:** Hex string of length `2 * n`.
- **Note:** Use this for session identifiers, nonces and keys. `random_hex(32)` is a ready-made 32-byte key.

```
bind start = do {
  bind key = random_hex(32)
  print(len(key))
}
```

```
OUTPUT
64
```

Authenticated encryption

Builtin	Purpose
`crypto_seal`	Encrypt and authenticate a string with XChaCha20-Poly1305.
`crypto_open`	Verify and decrypt the output of `crypto_seal`.

`crypto_seal(key_hex, text)`

Encrypt and authenticate a string with XChaCha20-Poly1305.

- **Aliases:** zh: `封印`; ja: `封印する`; ko: `봉인`; th: `ผนึก`
- **Parameters:**
- `key_hex` (string): 32-byte key as hex. Shorter keys are zero-padded, longer ones truncated, and invalid hex pairs are dropped (see the security guidance).
- `text` (string): plaintext.
- **Returns:** Hex string `nonce(24) || ciphertext || tag(16)`; the length is `2 * (40 + len(text))`. On failure, `Err(message)`.
- **Note:** A fresh random nonce is generated for every call, so sealing the same text twice gives different output.

```
bind start = do {
  bind key = "000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f"
  bind ct = crypto_seal(key, "attack at dawn")
  print(len(ct))
  print(crypto_open(key, ct))
}
```

```
OUTPUT
108
attack at dawn
```

`crypto_open(key_hex, ciphertext_hex)`

Verify and decrypt the output of `crypto_seal`.

- **Aliases:** zh: `解封`; ja: `封印解除`; ko: `봉인해제`; th: `เปิดผนึก`
- **Parameters:**

- `key_hex` (string): the same 32-byte key.
- `ciphertext_hex` (string): output of `crypto_seal`.
- **Returns:** The plaintext string, or `Err("decryption/auth failed")` if the key is wrong or a single bit was changed, or `Err("too short")`.
- **Note:** The plaintext is decoded as UTF-8 with replacement, so binary data does not round-trip. Encode it first (for example with `base64_encode`).

```
bind start = do {
  bind key = "000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f"
  bind ct = crypto_seal(key, "attack at dawn")
  print(crypto_open("ff" + key, ct))
  print(crypto_open(key, "00" + ct))
}
```

```
OUTPUT
Err(decryption/auth failed)
Err(decryption/auth failed)
```

Signatures

Three signature families are available, all with the same shape: create a keypair (random or from a seed), read the public key, sign a string, verify with the public key. Public keys and signatures are hex strings; the private key never leaves the interpreter.

Family	Builtin prefix	Public key	Signature	Notes
Ed25519	`ed25519_`	32 bytes	64 bytes	Fast, classical; deterministic signatures.
ML-DSA-65 (FIPS 204)	`mldsa_`	1952 bytes	3309 bytes	Post-quantum, NIST category 3.
ML-DSA-87 (FIPS 204)	`mldsa87_`	2592 bytes	4627 bytes	Post-quantum, NIST category 5; used by `lingtp://`.

For long-lived artifacts (licence files, release manifests) sign with Ed25519 **and** ML-DSA-87 and accept the artifact only if both verify.

Builtin	Purpose
`ed25519_keygen`	Generate a random Ed25519 keypair.
`ed25519_keygen_from_seed`	Derive an Ed25519 keypair deterministically from a 32-byte seed.

Builtin	Purpose
`ed25519_public`	Public (verifying) key of an Ed25519 handle.
`ed25519_sign`	Sign a string with an Ed25519 key.
`ed25519_verify`	Verify an Ed25519 signature.
`mldsa_keygen`	Generate a random ML-DSA-65 (post-quantum) signing keypair.
`mldsa_keygen_from_seed`	Derive an ML-DSA-65 keypair from a 32-byte seed.
`mldsa_public`	Public key of an ML-DSA-65 handle.
`mldsa_sign`	Sign a string with ML-DSA-65.
`mldsa_verify`	Verify an ML-DSA-65 signature.
`mldsa87_keygen`	Generate a random ML-DSA-87 (NIST category 5) signing keypair.
`mldsa87_keygen_from_seed`	Derive an ML-DSA-87 keypair from a 32-byte seed.
`mldsa87_public`	Public key of an ML-DSA-87 handle.
`mldsa87_sign`	Sign a string with ML-DSA-87.
`mldsa87_verify`	Verify an ML-DSA-87 signature.

`ed25519_keygen()`

Generate a random Ed25519 keypair.

- **Aliases:** zh: `生成签名密钥`; ja: `署名鍵生成`; ko: `생성서명키`; th: `สร้างกุญแจลายเซ็น`
- **Returns:** Integer handle for use with `ed25519_public` and `ed25519_sign`.
- **Note:** Handles are indexes into a table that lives as long as the interpreter. Passing an unknown handle to `ed25519_public` or `ed25519_sign` returns `""`.

```
bind start = do {
  bind id = ed25519_keygen()
  print(len(ed25519_public(id)))
}
```

```
OUTPUT
64
```

`ed25519_keygen_from_seed(seed_hex)`

Derive an Ed25519 keypair deterministically from a 32-byte seed.

- **Aliases:** zh: `从种子生成签名密钥`; ja: `シードから署名鍵生成`; ko: `씨앗에서생성서명키`; th: `สร้างกุญแจลายเซ็นจากเมล็ด`
- **Parameters:**
- `seed_hex` (string): 32-byte seed as 64 hex characters (padded or truncated if not).
- **Returns:** Integer handle.
- **Note:** A seed lets you persist only 64 hex characters and re-derive the identical keypair after a restart. Protect it like a private key.

```
bind start = do {
  bind id =
ed25519_keygen_from_seed("0101010101010101010101010101010101010101010101010101010101010101
  print(ed25519_public(id))
}
```

```
OUTPUT
8a88e3dd7409f195fd52db2d3cba5d72ca6709bf1d94121bf3748801b40f6f5c
```

`ed25519_public(handle)`

Public (verifying) key of an Ed25519 handle.

- **Aliases:** zh: `签名公钥`; ja: `署名公開鍵`; ko: `서명공개키`; th: `กุญแจสาธารณะลายเซ็น`
- **Parameters:**
- `handle` (number): from `ed25519_keygen` or `ed25519_keygen_from_seed`.
- **Returns:** 64-character hex string, or `""` for an unknown handle.

`ed25519_sign(handle, message)`

Sign a string with an Ed25519 key.

- **Aliases:** zh: `签名`; ja: `署名する`; ko: `서명하다`; th: `เซ็นชื่อ`
- **Parameters:**
- `handle` (number).
- `message` (string): the exact bytes to sign.
- **Returns:** 128-character hex signature, or `""` for an unknown handle.
- **Note:** Ed25519 is deterministic: the same key and message always give the same signature.

`ed25519_verify(public_hex, message, signature_hex)`

Verify an Ed25519 signature.

- **Aliases:** zh: `验证签名` ; ja: `署名検証` ; ko: `서명확인` ; th: `ยืนยันลายเซ็น`
- **Parameters:**
- `public_hex` (string): 32-byte public key.
- `message` (string).
- `signature_hex` (string): 64-byte signature.
- **Returns:** `true` if valid, `false` for a bad signature or any malformed input.

```
bind start = do {
  bind id =
ed25519_keygen_from_seed("0101010101010101010101010101010101010101010101010101010101010101
  bind pk = ed25519_public(id)
  bind sig = ed25519_sign(id, "msg")
  print(len(sig))
  print(ed25519_verify(pk, "msg", sig))
  print(ed25519_verify(pk, "msg2", sig))
  print(ed25519_verify("00", "msg", sig))
}
```

```
OUTPUT
128
true
false
false
```

`mldsa_keygen()`

Generate a random ML-DSA-65 (post-quantum) signing keypair.

- **Aliases:** zh: `生成量子签名密钥` ; ja: `量子署名鍵生成` ; ko: `양자서명키생성` ; th: `สร้างกุญแจลายเซ็นควอนตัม`
- **Returns:** Integer handle.

`mldsa_keygen_from_seed(seed_hex)`

Derive an ML-DSA-65 keypair from a 32-byte seed.

- **Aliases:** zh: `从种子生成量子签名密钥`; ja: `シードから量子署名鍵生成`; ko: `씨앗에서양자서명키생성`; th: `สร้างกุญแจลายเซ็นควอนตัมจากเมล็ด`
- **Parameters:**
- `seed_hex` (string): 32-byte seed as hex.
- **Returns:** Integer handle.
- **Note:** A seed lets you persist only 64 hex characters and re-derive the identical keypair after a restart. Protect it like a private key.

`mldsa_public(handle)`

Public key of an ML-DSA-65 handle.

- **Aliases:** zh: `量子签名公钥`; ja: `量子署名公開鍵`; ko: `양자서명공개키`; th: `กุญแจสาธารณะลายเซ็นควอนตัม`
- **Returns:** 3904-character hex string (1952 bytes), or `""` for an unknown handle.

`mldsa_sign(handle, message)`

Sign a string with ML-DSA-65.

- **Aliases:** zh: `量子签名`; ja: `量子署名する`; ko: `양자서명하다`; th: `เซ็นชื่อควอนตัม`
- **Parameters:**
- `handle` (number).
- `message` (string).
- **Returns:** Hex signature of 3309 bytes, or `""` for an unknown handle.

`mldsa_verify(public_hex, message, signature_hex)`

Verify an ML-DSA-65 signature.

- **Aliases:** zh: `验证量子签名`; ja: `量子署名検証`; ko: `양자서명확인`; th: `ยืนยันลายเซ็นควอนตัม`
- **Returns:** `true` or `false`; malformed input gives `false`.

```
bind start = do {
  bind id =
mldsa_keygen_from_seed("0101010101010101010101010101010101010101010101010101010101010101")
  bind sig = mldsa_sign(id, "release 1.0")
  print(len(mldsa_public(id)) / 2)
  print(len(sig) / 2)
  print(mldsa_verify(mldsa_public(id), "release 1.0", sig))
}
```

```
OUTPUT
1952
3309
true
```

`mldsa87_keygen()`

Generate a random ML-DSA-87 (NIST category 5) signing keypair.

- **Returns:** Integer handle.

`mldsa87_keygen_from_seed(seed_hex)`

Derive an ML-DSA-87 keypair from a 32-byte seed.

- **Parameters:**
- `seed_hex` (string): 32-byte seed as hex.
- **Returns:** Integer handle.
- **Note:** A seed lets you persist only 64 hex characters and re-derive the identical keypair after a restart. Protect it like a private key.

`mldsa87_public(handle)`

Public key of an ML-DSA-87 handle.

- **Returns:** Hex string of 2592 bytes, or `""` for an unknown handle.

`mldsa87_sign(handle, message)`

Sign a string with ML-DSA-87.

- **Returns:** Hex signature of 4627 bytes, or `""` for an unknown handle.

`mldsa87_verify(public_hex, message, signature_hex)`

Verify an ML-DSA-87 signature.

- **Returns:** `true` or `false`.

```
bind start = do {
  bind seed = "0101010101010101010101010101010101010101010101010101010101010101"
  bind ed = ed25519_keygen_from_seed(seed)
  bind pq = mldsa87_keygen_from_seed(seed)
  bind doc = "licence: acme, expires 2031-01-01"
  bind sig_a = ed25519_sign(ed, doc)
  bind sig_b = mldsa87_sign(pq, doc)
  bind ok_a = ed25519_verify(ed25519_public(ed), doc, sig_a)
  bind ok_b = mldsa87_verify(mldsa87_public(pq), doc, sig_b)
  print(ok_a && ok_b)
  print(mldsa87_verify(mldsa87_public(pq), doc + "!", sig_b))
}
```

```
OUTPUT
true
false
```

Key exchange (hybrid X25519 + ML-KEM-768)

The "knot" builtins expose the hybrid key-encapsulation mechanism from `ling-crypto`. The public key is 1216 bytes (`X25519 || ML-KEM-768`), the ciphertext 1120 bytes, and the shared secret 32 bytes. The shared secret is `SHA3-256(label || ss_mlkem || ss_x25519 || eph_pk || recipient_pk)`, so it stays safe if either X25519 or ML-KEM is later broken. The `hybrid_*` names are English synonyms for the `knot_*` ones.

The protocol is: the recipient calls `knot_keygen` and publishes `knot_public`; the sender calls `knot_encapsulate` on that public key, keeps the shared secret and sends the ciphertext; the recipient calls `knot_decapsulate` and gets the same secret. Use the secret as a `crypto_seal` key.

Builtin	Purpose
`knot_keygen`	Generate a hybrid X25519 + ML-KEM-768 keypair.
`knot_public`	Hybrid public key of a handle.
`knot_encapsulate`	Encapsulate a fresh shared secret to a hybrid public key.
`knot_decapsulate`	Recover the shared secret from a ciphertext.

`knot_keygen()`

Generate a hybrid X25519 + ML-KEM-768 keypair.

- **Aliases:** zh: `生成密钥`; ja: `鍵生成`; ko: `키생성`; th: `สร้างกุญแจปม`; also: `hybrid_keygen`
- **Returns:** Integer handle.
- **Note:** The private key lives only in the interpreter and cannot be exported or seeded. It is lost when the process exits.

`knot_public(handle)`

Hybrid public key of a handle.

- **Aliases:** zh: `公钥`; ja: `公開鍵`; ko: `공개키`; th: `กุญแจสาธารณะปม`; also: `hybrid_public`
- **Parameters:**
- `handle` (number) from `knot_keygen`.
- **Returns:** 2432-character hex string (1216 bytes), or `""` for an unknown handle.

`knot_encapsulate(public_hex)`

Encapsulate a fresh shared secret to a hybrid public key.

- **Aliases:** zh: `封装密钥`; ja: `カプセル化`; ko: `캡슐화`; th: `ห่อกุญแจปม`; also: `hybrid_encapsulate`
- **Parameters:**
- `public_hex` (string): the recipient's public key.
- **Returns:** `[ciphertext_hex, shared_secret_hex]` (a list), or `Err("hybrid public key wrong length")`.

`knot_decapsulate(handle, ciphertext_hex)`

Recover the shared secret from a ciphertext.

- **Aliases:** zh: `解封装密钥`; ja: `カプセル解除`; ko: `캡슐해제`; th: `แกะกุญแจปม`; also: `hybrid_decapsulate`
- **Parameters:**
- `handle` (number): the recipient's keypair.
- `ciphertext_hex` (string).
- **Returns:** 64-character hex shared secret, or `""` for an unknown handle or a ciphertext of the wrong length.
- **Note:** A well-formed but tampered ciphertext does not raise an error; it yields a different secret (implicit rejection). Authenticate with the secret to detect this.

```
bind start = do {
  bind bob = knot_keygen()
  bind enc = knot_encapsulate(knot_public(bob))
  bind ct = enc[0]
  bind secret_alice = enc[1]
  bind secret_bob = knot_decapsulate(bob, ct)
  print(len(knot_public(bob)) / 2)
  print(len(ct) / 2)
  print(secret_alice == secret_bob)

  bind msg = crypto_seal(secret_alice, "hello bob")
  print(crypto_open(secret_bob, msg))
}
```

```
OUTPUT
1216
1120
true
hello bob
```

Visual fingerprints and holographic splitting

The geometric builtins turn a digest into something you can look at. They are deterministic encodings of a SHA3 digest and add no cryptographic strength of their own; see item 10 of the security guidance.

Builtin	Purpose
`knot_points`	A 3-D torus-knot fingerprint of any string, as a flat list of coordinates.
`knot_label`	Short human-readable name of the fingerprint knot.
`holo_points`	Four-dimensional coordinates of the fragments of a holographic (all-or-nothing) split of a string.
`holo_fragment_count`	How many fragments `scatter` would produce for a string.

`knot_points(text)`

A 3-D torus-knot fingerprint of any string, as a flat list of coordinates.

- **Aliases:** zh: `结点坐标`; ja: `結び目点`; ko: `매듭점`; th: `จุดปม`
- **Parameters:**
- `text` (string): usually a public key hex or a file hash.
- **Returns:** Flat list `[x, y, z, x, y, z, ...]` of 256 points (768 numbers). Equal input gives an identical curve; one changed bit reshapes it.
- **Note:** Draw consecutive triples with `draw_line_3d` to show a user a picture of a key they can compare by eye.

```
bind start = do {
  print(len(knot_points("abc")))
}
```

```
OUTPUT
768
```

`knot_label(text)`

Short human-readable name of the fingerprint knot.

- **Aliases:** zh: `结点标签`; ja: `結び目ラベル`; ko: `매듭라벨`; th: `ป้ายปม`

- **Parameters:**
- `text` (string).
- **Returns:** String such as `knot-17_12-v5.2` (torus-knot type and a variant tag).

```
bind start = do {
  print(knot_label("abc"))
}
```

```
OUTPUT
knot-17_12-v5.2
```

`holo_points(text)`

Four-dimensional coordinates of the fragments of a holographic (all-or-nothing) split of a string.

- **Aliases:** zh: `全息点` ; ja: `ホログラム点` ; ko: `홀로그램점` ; th: `จุดโฮโลแกรม`
- **Parameters:**
- `text` (string).
- **Returns:** Flat list of four numbers per fragment, each point on the unit 3-sphere. The coordinates are decorative.
- **Note:** Every call re-randomises the internal key, so the coordinates change between calls. The underlying `scatter` transform is Rivest's all-or-nothing package transform: every fragment is needed to recover any byte. See `scatter` and `gather` in the Rust section.

`holo_fragment_count(text)`

How many fragments `scatter` would produce for a string.

- **Aliases:** zh: `全息碎片数` ; ja: `ホログラム断片数` ; ko: `홀로그램조각수` ; th: `จำนวนชิ้นโฮโลแกรม`
- **Parameters:**
- `text` (string).
- **Returns:** Number: `ceil((8 + len(text)) / 32)` data blocks (the 8 is a length prefix) plus one anchor block.

```
bind start = do {
  print(holo_fragment_count("a"))
  print(holo_fragment_count("hello"))
}
```

```
OUTPUT
2
2
```

Passwords and one-time codes

Builtin	Purpose
`password_hash`	Hash a password with Argon2id and return a PHC string.
`password_verify`	Check a password against a stored `password_hash` string.
`totp_secret`	Generate a random TOTP secret (RFC 6238) for two-factor authentication.
`totp_uri`	Build the `otpauth://` URI that authenticator apps import.
`totp_verify`	Check a six-digit code against a secret, allowing one 30-second step either side.
`totp_now`	The currently valid six-digit code for a secret, for tests and tooling.

`password_hash(password)`

Hash a password with Argon2id and return a PHC string.

- **Aliases:** ko: `비밀번호해시`
- **Parameters:**
- `password` (string).
- **Returns:** String like `$argon2id$v=19$m=65536,t=3,p=4$$`. Returns `""` if hashing fails.
- **Note:** Parameters are fixed: 64 MiB of memory, 3 passes, 4 lanes, with a fresh random 128-bit salt. Expect tens of milliseconds per call; do not call it on a hot path. Store the whole string.

```
bind start = do {
  bind h = password_hash("correct horse")
  print(substr(h, 0, 30))
  print(password_verify("correct horse", h))
  print(password_verify("wrong", h))
}
```

```
OUTPUT
$argon2id$v=19$m=65536,t=3,p=4
true
false
```

`password_verify(password, phc_string)`

Check a password against a stored `password_hash` string.

- **Aliases:** ko: `비밀번호확인`
- **Parameters:**
- `password` (string).
- `phc_string` (string): the stored hash.
- **Returns:** `true` or `false`. A malformed hash string gives `false`.
- **Note:** The parameters are read from the stored string, so old hashes keep verifying if the defaults ever change.

`totp_secret()`

Generate a random TOTP secret (RFC 6238) for two-factor authentication.

- **Aliases:** th: `โทเทนลับ`
- **Returns:** 32-character Base32 string (160 bits), compatible with Google Authenticator and Authy.
- **Note:** Needs the web-enabled build of `ling` (the `web` feature).

`totp_uri(secret, account, issuer)`

Build the `otpauth://` URI that authenticator apps import.

- **Aliases:** th: `โทเทนยูอาร์ไอ`
- **Parameters:**
- `secret` (string): Base32 secret.
- `account` (string, default `"user"`).
- `issuer` (string, default `"lingfu"`).
- **Returns:** String `otpauth://totp/:?secret=...&issuer=...&algorithm=SHA1&digits=6&period=30`.
- **Note:** Nothing is URL-encoded; do not put `&` or spaces in the account or issuer. Render the URI as a QR code with `lw_qr`.

```
bind start = do {
  print(totp_uri("JBSWY3DPEHPK3PXP", "alice@example.com", "Acme"))
}
```

```
OUTPUT
otpauth://totp/Acme:alice@example.com?
secret=JBSWY3DPEHPK3PXP&issuer=Acme&algorithm=SHA1&digits=6&period=30
```

`totp_verify(secret, code)`

Check a six-digit code against a secret, allowing one 30-second step either side.

- **Aliases:** th: `โทเทนตรวจ`
- **Parameters:**
- `secret` (string): Base32 secret.
- `code` (string): digits typed by the user (whitespace is trimmed).
- **Returns:** `true` or `false`.
- **Note:** Does not prevent replay of an accepted code within its window.

`totp_now(secret)`

The currently valid six-digit code for a secret, for tests and tooling.

- **Aliases:** th: `โทเทนตอนนี้`
- **Returns:** Six-digit string.

```
bind start = do {
  bind s = totp_secret()
  print(len(s))
  print(totp_verify(s, totp_now(s)))
  print(totp_verify(s, "000000"))
}
```

```
OUTPUT
32
true
false
```

Wallet-grade primitives (`lw_*`)

The `lw_` family implements the LW-1 primitive set that the ling-wallet project specifies (hashing, KEM, identities, ring signatures, proof-of-work, recovery phrases). Everything is hex in and hex out, with these differences from the simpler builtins above:

- **Strict parsing.** Hex arguments must be well formed and exactly the documented length. Bad input returns `Err(reason)` instead of being repaired.
- **Secret handles.** Functions that create secrets return `Ok(handle)` where the handle is 32 random hex characters valid only in the isolate that made it. Secrets are zeroized when the handle is dropped with `lw_key_drop`, or when the interpreter exits.
- **Capability.** Builtins that create or use a secret handle need the `"keys"` capability (`lw_key_drop` never does, so any isolate can discard a key).
- **Verifiers never throw.** `lw_id_verify`, `lw_ring_verify` and `lw_pk_valid` return `false` for any malformed input.

The examples define a small helper that unwraps a result:

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

Builtin	Purpose
`lw_h256`	Domain-separated SHAKE-256 hash with a 32-byte output.
`lw_xof`	Domain-separated SHAKE-256 with an output of `len` bytes.
`lw_random`	Random bytes from the OS CSPRNG, up to 1 MiB.

Builtin	Purpose
`lw_aead_seal`	XChaCha20-Poly1305 with a caller-chosen nonce and associated data.
`lw_aead_open`	Verify and decrypt `lw_aead_seal` output.
`lw_password_key`	Derive a 32-byte key from a passphrase with Argon2id.
`lw_kem_keypair`	Create an X25519 + ML-KEM-1024 hybrid KEM key from a 96-byte seed. Needs the `"keys"` capability.
`lw_kem_public`	Public key of a KEM handle.
`lw_kem_encaps`	Encapsulate a secret to a KEM public key, bound to a context.
`lw_kem_decaps`	Recover the shared secret. Needs the `"keys"` capability.
`lw_id_keypair`	Create an ML-DSA-87 identity key from a 32-byte seed. Needs the `"keys"` capability.
`lw_id_public`	Public key of an identity handle.
`lw_id_sign`	Sign a message under a context string.
`lw_id_verify`	Verify an identity signature.
`lw_id_fingerprint`	A short, human-comparable fingerprint of an identity public key.
`lw_key_drop`	Destroy a secret handle and zeroize the key it held.
`lw_spend_keypair`	Create a ring-signature spend key from a 32-byte seed (experimental LRS-87). Needs `"keys"`.
`lw_spend_public`	Public key of a spend handle.
`lw_onetime_secret`	Derive a one-time secret from a spend key, a 32-byte shared secret and an output index. Needs `"keys"`.
`lw_onetime_public`	Compute the one-time public key a sender can derive without the spend secret.
`lw_onetime_public_of`	Public key of a one-time handle.
`lw_onetime_key_image`	The linkable key image of a one-time key; signing twice with one key yields the same image.
`lw_ring_sign`	Sign a 32-byte message with a linkable ring signature. Needs `"keys"`.
`lw_ring_verify`	Verify a ring signature against a ring and key image.
`lw_pk_valid`	Is this a canonical 2944-byte ring public key?
`lw_mnemonic_from`	Encode 32 bytes of entropy as a 24-word BIP-39 English recovery phrase.
`lw_mnemonic_to`	Decode a 24-word phrase back to its entropy.
`lw_pow_hash`	HyperTorus memory-hard proof-of-work hash of a header.
`lw_pow_trace`	Like `lw_pow_hash` but also records the walk for visualisation.

Builtin	Purpose
`lw_meets_difficulty`	Does a 32-byte hash satisfy a difficulty target?
`lw_qr`	Encode text as a QR code (error-correction level M).

`lw_h256(domain, parts)`

Domain-separated SHAKE-256 hash with a 32-byte output.

- **Parameters:**
- `domain` (string): a label that separates uses of the hash.
- `parts` (list of hex strings): the pieces to hash; each is length-prefixed internally, so `["ab","cd"]` differs from `["abcd"]`.
- **Returns:** 64-character hex digest.

```
bind start = do {
  print(lw_h256("app.v1", ["616263"]))
  print(lw_h256("app.v2", ["616263"]))
}
```

```
OUTPUT
b3591f3c59e2a95a244cefed9cb3db5999634fe8945e04a220574d362da63bd7
638276538316ed09adc2ec32fb4fab608788081ca40003117bfc37e022b50c8f
```

`lw_xof(domain, parts, len)`

Domain-separated SHAKE-256 with an output of `len` bytes.

- **Parameters:**
- `domain`, `parts`: as for `lw_h256`.
- `len` (number): 0 to 1,048,576.
- **Returns:** Hex string of `2 * len` characters.
- **Note:** The first 32 bytes equal `lw_h256` for the same domain and parts.

```
bind start = do {
  print(lw_xof("app.v1", ["616263"], 8))
}
```

```
OUTPUT
b3591f3c59e2a95a
```

`lw_random(n)`

Random bytes from the OS CSPRNG, up to 1 MiB.

- **Parameters:**
- `n` (number): byte count, 0 to 1,048,576.
- **Returns:** Hex string.

`lw_aead_seal(key, nonce, aad, plaintext)`

XChaCha20-Poly1305 with a caller-chosen nonce and associated data.

- **Parameters:**
- `key` (hex, 32 bytes).
- `nonce` (hex, 24 bytes): must be unique per key.
- `aad` (hex): authenticated but not encrypted (use `""` for none).
- `plaintext` (hex).
- **Returns:** Hex `ciphertext || tag(16)`. Raises a runtime error on malformed arguments.

`lw_aead_open(key, nonce, aad, ciphertext)`

Verify and decrypt `lw_aead_seal` output.

- **Returns:** `Ok(plaintext_hex)` or `Err("authentication failed")`.

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

bind start = do {
  bind key = "0202020202020202020202020202020202020202020202020202020202020202"
  bind nonce = "000000000000000000000000000000000000000000000000"
  bind sealed = lw_aead_seal(key, nonce, "", "68656c6c6f")
  print(sealed)
  print(u(lw_aead_open(key, nonce, "", sealed)))
  print(u(lw_aead_open(key, nonce, "00", sealed)))
}
```

```
OUTPUT
68363e98f64c000d149ac45df2e231de2ce5fda831
68656c6c6f
ERR:authentication failed
```

`lw_password_key(password, salt, m_kib, t, p)`

Derive a 32-byte key from a passphrase with Argon2id.

- **Parameters:**
- `password` (string).
- `salt` (hex, 16 bytes): store it next to the ciphertext; it need not be secret.
- `m_kib` (number): memory in KiB, up to 4,194,304.
- `t` (number): passes, up to 1024.
- `p` (number): lanes, up to 64.
- **Returns:** `Ok(key_hex)` (64 hex characters) or `Err("invalid Argon2id parameters")`.
- **Note:** The ling-wallet profile is `m = 262144` (256 MiB), `t = 3`, `p = 1`. The example uses tiny parameters so it runs instantly; do not copy them.

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

bind start = do {
  bind salt = "00112233445566778899aabbccddeeff"
  print(u(lw_password_key("hunter2", salt, 1024, 1, 1)))
}
```

```
OUTPUT
a97cf9fed0c0703d0b60b7351990a06b086636d9fec2c5ef4d2b7a4bae25bb34
```

`lw_kem_keypair(seed)`

Create an X25519 + ML-KEM-1024 hybrid KEM key from a 96-byte seed. Needs the `"keys"` capability.

- **Parameters:**
- `seed` (hex, 96 bytes = 32 for X25519 and 64 for ML-KEM).
- **Returns:** `Ok(handle)`.

`lw_kem_public(handle)`

Public key of a KEM handle.

- **Returns:** Hex of 1600 bytes.

`lw_kem_encaps(public_hex, context_hex)`

Encapsulate a secret to a KEM public key, bound to a context.

- **Parameters:**
- `public_hex`: 1600-byte public key.
- `context_hex`: application context bytes mixed into the secret.
- **Returns:** `Ok([ciphertext_hex, shared_secret_hex])`; the ciphertext is 1600 bytes.

`lw_kem_decaps(handle, ciphertext_hex, context_hex)`

Recover the shared secret. Needs the `"keys"` capability.

- **Returns:** `Ok(secret_hex)` or `Err("ciphertext must be 1600 bytes")` / `Err("malformed ciphertext")`.
- **Note:** The context must match the one used by the sender, otherwise a different secret results.

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

fn rep(s, n) {
  bind out = ""
  bind i = 0
  while i  v, bad(e) => "ERR:" + e } }

bind start = do {
  bind seed = "0101010101010101010101010101010101010101010101010101010101010101"
  bind id = u(lw_id_keypair(seed))
  bind pk = lw_id_public(id)
  bind sig = lw_id_sign(id, "ctx", "68656c6c6f")
  print(len(pk) / 2)
  print(len(sig) / 2)
  print(lw_id_verify(pk, "ctx", "68656c6c6f", sig))
  print(lw_id_verify(pk, "other", "68656c6c6f", sig))
  print(lw_id_fingerprint(pk))
  print(lw_key_drop(id))
}
```

```
OUTPUT
2592
4627
true
false
9ebc-56af-97f2-61af
true
```

`lw_key_drop(handle)`

Destroy a secret handle and zeroize the key it held.

- **Returns:** `true` if the handle existed.

`lw_spend_keypair(seed)`

Create a ring-signature spend key from a 32-byte seed (experimental LRS-87). Needs `"keys"`.

- **Returns:** `Ok(handle)`.

`lw_spend_public(handle)`

Public key of a spend handle.

- **Returns:** Hex of 2944 bytes.

`lw_onetime_secret(spend_handle, shared_hex, index)`

Derive a one-time secret from a spend key, a 32-byte shared secret and an output index. Needs `"keys"`.

- **Parameters:**
- `shared_hex` (32 bytes).
- `index` (number, up to 4,294,967,295).
- **Returns:** `Ok(handle)`.

`lw_onetime_public(spend_public_hex, shared_hex, index)`

Compute the one-time public key a sender can derive without the spend secret.

- **Returns:** `Ok(public_hex)`.

`lw_onetime_public_of(onetime_handle)`

Public key of a one-time handle.

- **Returns:** Hex of 2944 bytes.

`lw_onetime_key_image(onetime_handle)`

The linkable key image of a one-time key; signing twice with one key yields the same image.

- **Returns:** Hex of 2944 bytes.

`lw_ring_sign(message_hex, ring, signer_index, onetime_handle)`

Sign a 32-byte message with a linkable ring signature. Needs `"keys"`.

- **Parameters:**
- `message_hex` (32 bytes).
- `ring` (list of hex public keys): 1 to 64 members, each a valid 2944-byte key.
- `signer_index` (number): position of the signer's key in the ring.
- `onetime_handle`.
- **Returns:** `Ok(signature_hex)`, or `Err` if the ring size, the index or the key does not match `ring[signer_index]`.
- **Note:** LRS-87 is marked experimental in the source. Use it for research and prototypes, not for assets you cannot afford to lose.

`lw_ring_verify(message_hex, ring, key_image_hex, signature_hex)`

Verify a ring signature against a ring and key image.

- **Returns:** `true` or `false`; never raises.

`lw_pk_valid(public_hex)`

Is this a canonical 2944-byte ring public key?

- **Returns:** `true` or `false`.

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

fn rep(s, n) {
  bind out = ""
  bind i = 0
  while i  v, bad(e) => "ERR:" + e } }

bind start = do {
  bind entropy = "111111111111111111111111111111111111111111111111111111111111111111111111"
  bind words = lw_mnemonic_from(entropy)
  print(len(words))
  print(words[0] + " " + words[1] + " " + words[23])
  bind phrase = ""
  bind i = 0
  while i  String { b.iter().map(|x| format!("{x:02x}")).collect() }

fn main() {
    println!("{}", hex(&Blake3::hash(b"abc")));
    println!("{}", hex(&Sha3_256::hash(b"abc")));
    println!("{}", hex(&Shake256::hash(b"abc", 16)));
}
```

```
OUTPUT
6437b3ac38465133ffb63b75273a8db548c558465d79db03fd359c6cd5bd9d85
3a985da74fe225b2045c172d6bd390bd855f086e3e9d525b46bfe24511431532
483366601360a8771c6863080cc4114d
```

The first value equals the Ling builtin `hash_hex("abc")`; the third equals `shake_hex("abc", 16)`.

Authenticated encryption: AES-256-GCM and XChaCha20-Poly1305

Both types are constructed from a 32-byte key (`new`), can make a key (`generate_key`), and have `encrypt`/`decrypt` that return `Result`. The output layout is `nonce || ciphertext || tag`; the nonce (12 bytes for AES-GCM, 24 for XChaCha20) is random per call and prepended for you. Prefer XChaCha20: its 24-byte random nonce is safe for an unbounded number of messages under one key, while a 12-byte GCM nonce is not.

```
use ling_crypto::{AesGcm256, XChaCha20};

let aes = AesGcm256::new(AesGcm256::generate_key());
let ct = aes.encrypt(b"attack at dawn").unwrap();
assert_eq!(ct.len(), 12 + 14 + 16);
assert_eq!(aes.decrypt(&ct).unwrap(), b"attack at dawn");

let xc = XChaCha20::new(XChaCha20::generate_key());
let mut ct = xc.encrypt(b"attack at dawn").unwrap();
assert_eq!(ct.len(), 24 + 14 + 16);
ct[30] ^= 1;                                  // flip one bit
assert_eq!(xc.decrypt(&ct), Err("decryption/auth failed"));
```

Passwords and key derivation

```
use ling_crypto::{hkdf_sha3, Argon2idParams};

// Tuned down for the example; the default is 64 MiB, 3 passes, 4 lanes.
let p = Argon2idParams { m_cost: 1024, t_cost: 1, p_cost: 1 };
let phc = p.hash_password(b"hunter2").unwrap();          //
"$argon2id$v=19$m=1024,t=1,p=1$..."
assert!(Argon2idParams::verify_password(b"hunter2", &phc).is_ok());
assert_eq!(Argon2idParams::verify_password(b"x", &phc), Err("password incorrect"));

let key = p.derive_key(b"hunter2", b"0123456789abcdef", 32).unwrap();   //
Zeroizing
let sub = hkdf_sha3(b"secret", b"salt", b"context label", 16).unwrap();
```

`hash_password` and `verify_password` exist only with the `std` feature. `derive_key` and `hkdf_sha3` work everywhere and return `Zeroizing` buffers. Use a unique, random salt of at least 16 bytes per password.

Signatures and key exchange

```
use ling_crypto::{Ed25519Keypair, MlDsa87Keypair, MlKem768Keypair, HybridKeypair,
                  mlkem768_encapsulate, hybrid_encapsulate};

let ed = Ed25519Keypair::from_seed([1u8; 32]);
let sig = ed.sign(b"msg");
Ed25519Keypair::verify(&ed.public_key(), b"msg", &sig).unwrap();

let ml = MlDsa87Keypair::from_seed([1u8; 32]);
let s = ml.sign(b"msg");                        // 4627 bytes; public key is 2592 bytes
MlDsa87Keypair::verify(&ml.public_key(), b"msg", &s).unwrap();

let kp = MlKem768Keypair::generate();           // encapsulation key 1184 B, ciphertext
1088 B
let (ct, ss_sender) = mlkem768_encapsulate(&kp.encapsulation_key()).unwrap();
assert_eq!(ss_sender, kp.decapsulate(&ct).unwrap());

let hk = HybridKeypair::generate();             // public key 1216 B, ciphertext 1120 B
let (ct, ss) = hybrid_encapsulate(&hk.public_key()).unwrap();
assert_eq!(ss, hk.decapsulate(&ct).unwrap());
```

`MlKem768Keypair` also offers `from_seed([u8; 64])` and `seed()`; `MlDsa65Keypair` mirrors `MlDsa87Keypair` with 1952-byte public keys and 3309-byte signatures. `X25519Secret` provides plain Diffie-Hellman (`generate`, `from_bytes`, `public_key`, `diffie_hellman`); prefer the hybrid KEM for new protocols.

Shamir secret sharing

`split_secret(secret, threshold, n)` returns `n` `Share { x, y }` values (x runs from 1 to n); any `threshold` of them reconstruct the secret over GF(2^8) with `reconstruct_secret`. The functions **panic** if `threshold attack at dawn
tampered: Err("decryption/auth failed")
mlkem ek 1184 ct 1088 equal true
hybrid pk 1216 ct 1120 equal true
recovered launch code 1234
schnorr ok true wrongmsg false
vrf deterministic true
mldsa87 pk 2592 sig 4627 Ok(())
```

CHAPTER 16

AI, Language Models and Video

Ling ships three layers of machine intelligence, from tiny to large:

1. **A game-AI toolkit that is always available.** `nn_*` (a small feed-forward neural network trained with stochastic gradient descent), `bt_*` (behaviour trees) and `dialog_*` (a miniature neural language model that learns a character's voice). Pure Rust, CPU only, no model files, kilobytes of state. These builtins are part of every `ling`.
2. **Local large-language-model and vision inference.** `llm_*` loads a quantised GGUF checkpoint (Qwen3 or GLM-4) and generates text on a background thread; `vision_*` loads a quantised Moondream2 checkpoint and describes images. These need a `ling` built with the `llm` and `vision` features and a model file you supply. They are **not** present in the default build; there the names evaluate to `()` without an error.
3. **Rust libraries.** `ling-ai` (tensors, networks, behaviour trees, the dialog model, and the GGUF engine behind layer 2) and `ling-llm` (a freestanding `no_std` Qwen3 engine that LingOS uses).

Video playback (`video_*`) is also documented here because it decodes H.264 and AAC in the runtime and exposes frames as images.

Two honest limits. First, there is no builtin named `ai_chat` or `ai_embed`; those two names appear in the language lexicons but nothing in the interpreter implements them, and calling them evaluates to `()`. Second, there is no hosted-model client in the standard library: to call a remote model API you use `http_post_async` (Chapter 30) and parse the response yourself, with `json_escape` to build the request body.

Conventions

- **Handles are integers.** `nn_new`, `bt_build`, `dialog_new`, `llm_load` and `vision_load` return a small integer. A failed load returns `-1`. Passing a bad handle to any other function is safe and returns a neutral value (`[]`, `0`, `""`, `false`); it never crashes the interpreter.
- **Lists of numbers** are Ling lists such as `[0.5, 1]`; they are converted to 32-bit floats inside. Non-numbers in the list are skipped.

- **Aliases.** The interpreter defines Thai names for the game-AI builtins, which are listed with each entry. Chinese, Japanese and Korean lexicon names for them exist only in `lexicons/*.ling`, which the interpreter does not read.
- **Jobs.** Long operations (`llm_generate_start`, `vision_analyze_start`) return a job id string at once and run on a background thread. You poll in a loop with `sleep(ms)`.

Neural networks: `nn_*`

`nn_*` is a dense feed-forward network: you give the number of inputs, add layers, then call `nn_train` once per example. Training is plain stochastic gradient descent with a mean-squared-error loss, with no autograd graph and no allocations on the inference path, so it is cheap enough to run every frame. Weights are initialised from the seed you pass to `nn_new`, so a given seed gives a reproducible result.

Activation names are `"relu"`, `"tanh"`, `"sigmoid"` (or `"logistic"`), and anything else means linear. Names are case-insensitive.

Builtin	Purpose
`nn_new`	Create an empty network that takes `n_inputs` numbers.
`nn_dense`	Append a fully connected layer.
`nn_forward`	Run the network on one input vector.
`nn_train`	One gradient-descent step on one example.
`nn_save`	Write the network to a file in the compact `LNN1` binary format.
`nn_load`	Load a network saved by `nn_save`.

`nn_new(n_inputs, seed)`

Create an empty network that takes `n_inputs` numbers.

- **Aliases:** zh: `建神经网` ; ja: `ニューラル作成` ; ko: `신경망생성` ; th: `สร้างโครงข่าย`
- **Parameters:**
- `n_inputs` (number, default 1).
- `seed` (number, default 1): seeds the weight initialiser.
- **Returns:** Network handle.

`nn_dense(net, units, activation)`

Append a fully connected layer.

- **Aliases:** zh: `密集层` ; ja: `密層追加` ; ko: `밀집층` ; th: `ชั้นหนาแน่น`
- **Parameters:**
- `net` : handle.
- `units` (number, at least 1): neurons in the layer.
- `activation` (string, default `"relu"`).
- **Returns:** Nothing. Layers are applied in the order added; the last layer's width is the output size.

`nn_forward(net, inputs)`

Run the network on one input vector.

- **Aliases:** zh: `神经前向` ; ja: `順伝播` ; ko: `순전파` ; th: `ส่งต่อโครงข่าย`
- **Parameters:**
- `inputs` (list of numbers): shorter lists are zero-padded and longer ones truncated to the network's input width.
- **Returns:** List of output numbers; `[]` for an invalid handle.

`nn_train(net, inputs, targets, learning_rate)`

One gradient-descent step on one example.

- **Aliases:** zh: `训练网` ; ja: `ニューラル学習` ; ko: `신경망학습` ; th: `ฝึกโครงข่าย`
- **Parameters:**
- `inputs` , `targets` (lists of numbers).
- `learning_rate` (number, default 0.01).
- **Returns:** The loss before the update, `0.5 * sum((output - target)^2)` ; `0` for an invalid handle.
- **Note:** Call it in a loop over your whole data set for many epochs. Shuffle the order if the examples are ordered.

```
bind start = do {
  bind net = nn_new(2, 7)
  nn_dense(net, 8, "tanh")
  nn_dense(net, 1, "sigmoid")
  for epoch in 0..2000 {
    nn_train(net, [0, 0], [0], 0.5)
    nn_train(net, [0, 1], [1], 0.5)
    nn_train(net, [1, 0], [1], 0.5)
    nn_train(net, [1, 1], [0], 0.5)
  }
  print(nn_forward(net, [0, 0]))
  print(nn_forward(net, [0, 1]))
  print(nn_forward(net, [1, 0]))
  print(nn_forward(net, [1, 1]))
}
```

```
OUTPUT
[0.010534209199249744]
[0.982124388217926]
[0.9828534722328186]
[0.019928984344005585]
```

`nn_save(net, path)`

Write the network to a file in the compact `LNN1` binary format.

- **Aliases:** zh: `保存网` ; ja: `網保存` ; ko: `신경망저장` ; th: `บันทึกโครงข่าย`
- **Parameters:**
- `path` (string, default `"model.lnn"`).
- **Returns:** `true` on success.
- **Note:** Little-endian and stable across runs and platforms. The format has no checksum or signature; do not load `.lnn` files from untrusted sources into a service without validating them.

`nn_load(path)`

Load a network saved by `nn_save` .

- **Aliases:** zh: `载入网` ; ja: `網読込` ; ko: `신경망불러오기` ; th: `โหลดโครงข่าย`
- **Parameters:**
- `path` (string, default `"model.lnn"`).
- **Returns:** A new handle, or `-1` if the file is missing or not a valid `LNN1` file.

```
bind start = do {
  bind net = nn_new(1, 3)
  nn_dense(net, 4, "tanh")
  nn_dense(net, 1, "linear")
  print(nn_save(net, "tiny.lnn"))
  bind again = nn_load("tiny.lnn")
  print(nn_forward(net, [0.5])[0] == nn_forward(again, [0.5])[0])
  print(nn_load("missing.lnn"))
  remove_file("tiny.lnn")
}
```

```
OUTPUT
true
true
-1
```

Behaviour trees: `bt_*`

A behaviour tree is the decision layer of an NPC. You write it in a small text language and tick it against a **blackboard** of numeric facts (health, distance, ammo). A tick returns the name of the action the tree chose; your program stays in charge of how that action runs.

```
OUTPUT
selector {                      # try children in order until one succeeds
    sequence {                  # all children must succeed
        ?enemy_visible > 0      # condition: blackboard["enemy_visible"] > 0
        ?ammo >= 1
        !attack                 # action: choose "attack"
    }
    sequence {
        ?health `, `=`, ` 0 ?ammo >= 1 !attack }
sequence { ?enemy_visible > 0 !flee } !patrol }")
  print(bt_tick(brain))
  bt_set(brain, "enemy_visible", 1)
  bt_set(brain, "ammo", 5)
  print(bt_tick(brain))
  print(bt_status(brain))
  bt_set(brain, "ammo", 0)
  print(bt_tick(brain))
  print(bt_build("selector {"))
}
```

```
OUTPUT
patrol
attack
1
flee
-1
```

Dialog models: `dialog_*`

`dialog_*` gives each NPC a miniature language model. It is a Bengio-style neural probabilistic model: word embeddings feed a fixed context window into a dense network that predicts the next word, and the embeddings and network are trained together with cross-entropy loss. It is deliberately tiny (kilobytes, one CPU core). It will not write a novel: it captures the **cadence and vocabulary** of the lines it was trained on and samples fresh lines in that voice. Expect it to reproduce phrases from a small corpus.

Training text can be fed with `dialog_learn` (one line at a time) or from a file in the **Ling Dialog Standard** (`.lingdialog`), a human-writable corpus format:

```
OUTPUT
# comments start with '#'
@meta title: Tavern Keeper

= welcome                                  # '=' starts a conversation (the label is
optional)
bram: welcome traveler rest your boots     # "speaker: text" is one turn
hero: any news from the road
bram: bandits on the north pass
                                           # an indented line without a colon continues
the previous turn
```

Only the turn text is used for training (speaker names are dropped), and the turns of a conversation are joined with an end-of-line marker so the model learns to continue across turns. Text is lower-cased and split into words, with punctuation as separate tokens.

Builtin	Purpose
`dialog_new`	Create an untrained dialog model.
`dialog_learn`	Add one line of text to the training corpus.
`dialog_load`	Add a `.lingdialog` file to the training corpus.
`dialog_train`	Train on everything learned so far.
`dialog_say`	Generate a line in the learned voice.
`dialog_save`	Save the trained model (vocabulary, embeddings, weights).
`dialog_load_model`	Load a model saved by `dialog_save`.

`dialog_new(context, embed, hidden, seed)`

Create an untrained dialog model.

- **Aliases:** zh: `建对话模型`; ja: `対話モデル作成`; ko: `대화모델생성`; th: `สร้างโมเดลสนทนา`
- **Parameters:**
- `context` (number, default 3): how many preceding words the model sees.
- `embed` (number, default 32): embedding width.
- `hidden` (number, default 64): hidden-layer width.
- `seed` (number, default 1).
- **Returns:** Model handle.

`dialog_learn(model, text)`

Add one line of text to the training corpus.

- **Aliases:** zh: `对话学习`; ja: `対話学習`; ko: `대화학습`; th: `เรียนรู้สนทนา`
- **Returns:** Nothing.

`dialog_load(model, path)`

Add a `.lingdialog` file to the training corpus.

- **Aliases:** zh: `对话载入`; ja: `対話読込`; ko: `대화불러오기`; th: `โหลดชุดสนทนา`
- **Parameters:**
- `path` (string).
- **Returns:** The number of training lines added (one per conversation), or `-1` if the file cannot be read or the handle is invalid.

`dialog_train(model, epochs, learning_rate)`

Train on everything learned so far.

- **Aliases:** zh: `对话训练`; ja: `対話訓練`; ko: `대화훈련`; th: `ฝึกสนทนา`
- **Parameters:**
- `epochs` (number, default 20).
- `learning_rate` (number, default 0.1).
- **Returns:** Average cross-entropy loss of the final epoch (lower is better); `0` if there is no data.

- **Note:** Adding more text after training grows the vocabulary and rebuilds the output layer while keeping the embeddings; train again afterwards.

`dialog_say(model, prompt, max_tokens, temperature)`

Generate a line in the learned voice.

- **Aliases:** zh: `对话生成` ; ja: `対話生成` ; ko: `대화생성` ; th: `พูดสนทนา`
- **Parameters:**
- `prompt` (string): seeds the context window.
- `max_tokens` (number, default 24).
- `temperature` (number, default 0.8): `0` or less is greedy; higher is more random. Sampling uses top-k 40 and top-p 0.9.
- **Returns:** Generated text. Stops at an end-of-line marker. `""` if the model has not been trained.
- **Note:** Output is deterministic for a given seed, corpus and arguments.

`dialog_save(model, path)`

Save the trained model (vocabulary, embeddings, weights).

- **Aliases:** zh: `对话存模` ; ja: `対話モデル保存` ; ko: `대화모델저장` ; th: `บันทึกโมเดลสนทนา`
- **Parameters:**
- `path` (string, default `"model.llm"`).
- **Returns:** `true` on success.

`dialog_load_model(path)`

Load a model saved by `dialog_save`.

- **Aliases:** zh: `对话载模` ; ja: `対話モデル読込` ; ko: `대화모델불러오기` ; th: `โหลดโมเดลสนทนา`
- **Parameters:**
- `path` (string, default `"model.llm"`).
- **Returns:** A new handle, or `-1` for a missing or invalid file.

```
bind start = do {
  write_file("tavern.lingdialog", "= welcome\nbram: welcome traveler rest your boots by
the fire\nhero: any news from the road\nbram: bandits on the north pass nothing my stew
cannot cure\n= rumors\nbram: they say the old mine woke something deep and
hungry\nhero: should i be worried\nbram: keep your blade sharp and your purse close
traveler\n")
  bind npc = dialog_new(3, 16, 32, 1)
  print(dialog_load(npc, "tavern.lingdialog"))
  bind loss = dialog_train(npc, 40, 0.1)
  print(loss = 0)
  print(dialog_load_model("missing.llm"))
  remove_file("tavern.lingdialog")
  remove_file("bram.llm")
}
```

```
OUTPUT
2
true
rest your boots by the fire
true
true
-1
```

Local language models: `llm_*`

`llm_*` runs a chat-tuned model inside the `ling` process: there is no external server. It loads a quantised **GGUF** checkpoint with `candle`, plus a Hugging Face `tokenizer.json`, on the CPU or a CUDA device. The architecture is read from the file's `general.architecture` metadata; `qwen3` and `glm4` are supported.

> **Availability.** These builtins exist only in a `ling` built with the Cargo feature `llm` (add `llm-cuda` for GPU). In the default build the names evaluate to `()`. Test for it before relying on it: `llm_load` returns `-1` for a bad file in a capable build and

`()` in an incapable one.

Generation never blocks the interpreter: `llm_generate_start` returns a job id at once and decodes on a background thread; you poll `llm_generate_poll` for the text produced so far. This matters most inside `http_serve` handlers, where a multi-second decode on the request thread would stall every visitor.

History format. Conversation history is passed as one string in a fixed "wire format":

```
OUTPUT

What is a lens?

A curved piece of glass that bends light.

And a mirror?
```

For Qwen3 models the runtime re-renders this as ChatML and appends an empty `` block so the model answers directly. The `system_prompt` argument becomes the system turn.

Limits and honesty. A generation job holds the model's lock, so only one generation runs at a time per handle. Output quality is whatever the checkpoint gives you; there is no moderation layer, so filter output before showing it to untrusted users and do not give model output direct access to tools, files or SQL. Tool calling is declared in the API but not implemented: no job ever reaches the `tool_call` status.

Builtin	Purpose
`llm_load`	Load a GGUF chat model and its tokenizer.
`llm_generate_start`	Start generating a reply on a background thread.
`llm_generate_poll`	The text generated so far.
`llm_generate_done`	Has the job finished?
`llm_job_status`	Three-state job status.
`llm_job_tool_name`	Name of a requested tool call.
`llm_job_provide_tool_result`	Resume a job that paused for a tool call.
`llm_job_cancel`	Mark a job as done.

`llm_load(gguf_path, tokenizer_path, device)`

Load a GGUF chat model and its tokenizer.

- **Parameters:**
- `gguf_path` (string): the `.gguf` checkpoint (for example a Q8_0 or Q4_K quantisation of Qwen3).
- `tokenizer_path` (string): the matching `tokenizer.json`.
- `device` (number, default 0): a negative number selects the CPU; otherwise the CUDA device index.
- **Returns:** Model handle, or `-1` on any load failure (missing file, unsupported architecture, no CUDA device).
- **Note:** Loading reads the whole model into memory and can take seconds. Load once at start-up.

```
GUI
bind start = do {
  bind model = llm_load("models/Qwen3-0.6B-Q8_0.gguf", "models/tokenizer.json", 0 - 1)
  if model = 0 {
      image_draw(img, 0, 0)
    }
    present()
  }
  video_close(v)
}
```

`video_close(id)`

Stop decoding and release the video; its image handle becomes a 1x1 blank.

- **Returns:** Nothing.

```
bind start = do {
  bind v = video_open("nofile.mp4")
  bind n = 0
  while video_state(v) == 0 && n  0)
  print(video_width(v))
  print(video_frame(v))
  video_close(v)
  print(video_state(v))
  print(video_error(v))
}
```

```
OUTPUT
-1
true
0
-1
-1
no such video
```

Speech recognition (`stt_*`) is a separate family built on the Whisper port in `ling-ai` and is covered with the audio builtins.

`ling-ai` (Rust)

`ling-ai` holds the engine behind the builtins above plus a few types that have no builtin of their own. It is not a general machine-learning framework: the tensor type is deliberately minimal, and the heavy features are opt-in.

Feature	Enables
(default)	`tensor`, `nn`, `bt`, `dialog_lm`, `dataset`, `model`, `prompt`, `tokenizer`
`llm`	`llm::LoadedModel` (GGUF chat inference on `candle`), `whisper`, `rvc`
`vision`	`vision::VisionModel` (Moondream2); implies `llm`
`cuda`, `llm-cuda`, `metal`	GPU back ends for the above

Using it from Ling

The builtins `nn_*`, `bt_*` and `dialog_*` call `ling_ai::Net`, `Tree` and `DialogLM` through three process-wide handle tables (a `Vec` guarded by a mutex; the handle is the slot number). `llm_*` and `vision_*` call `LoadedModel` and `VisionModel`, adding the job table and a background thread. Handles are never reused, so a program that creates thousands of networks leaks memory until it exits.

Using it from Rust

```
[dependencies]
ling-ai = "2030"                          # add features = ["llm"] for GGUF inference
```

Tensors. `Tensor { shape, data }` is an N-dimensional array of `f32` with `zeros`, `ones`, `from_vec`, `get`, `set`, element-wise `add` and `mul`, a naive triple-loop `matmul` for 2-D tensors, `softmax` along the data and `relu`. Shape mismatches **panic**.

```
use ling_ai::Tensor;

let a = Tensor::from_vec(vec![2, 3], vec![1., 2., 3., 4., 5., 6.]);
let b = Tensor::from_vec(vec![3, 2], vec![1., 0., 0., 1., 1., 1.]);
let c = a.matmul(&b);
assert_eq!(c.shape.0, vec![2, 2]);
assert_eq!(c.data, vec![4.0, 5.0, 10.0, 11.0]);
let p = Tensor::from_vec(vec![3], vec![1., 2., 3.]).softmax();   // [0.0900, 0.2447,
0.6652]
```

Networks. `Net::new(n_inputs, seed)`, then `.dense(units, Activation::Tanh)` (chainable), `forward`, `train_mse(input, target, lr)`, and `to_bytes` / `from_bytes` for the `LNN1` format. For custom training loops, `forward_cache` returns each layer's activations and `backward(&acts,`

`&d_out, lr)` applies a gradient and returns the gradient with respect to the input.

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

let mut net = Net::new(2, 7);
net.dense(4, Activation::Tanh).dense(1, Activation::Sigmoid);
for _ in 0..3000 {
    for (x, y) in [([0., 0.], 0.), ([0., 1.], 0.), ([1., 0.], 0.), ([1., 1.], 1.)] {
        net.train_mse(&x, &[y], 0.5);
    }
}
println!("AND(1,1) = {:.2}", net.forward(&[1., 1.])[0]);          // AND(1,1) = 0.99
assert!(Net::from_bytes(&net.to_bytes()).is_some());
```

Behaviour trees and dialog models.

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

let mut t = Tree::parse("selector { sequence { ?hp  Result {
    // Load: parse the GGUF, read the weights, build the tokenizer.
    let llm = Llm::load(&model_bytes, &mut |done, total| eprintln!("{done}/{total}"))?;

    // A fresh KV cache for up to 2048 tokens.
    let mut state = llm.model.new_state(2048);

    let mut out = String::new();
    llm.generate(
        &mut state,
        "You are concise.",                // system prompt
        "\nName three colours.\n",  // wire-format history
        64,                                // max tokens
        0.0,                               // temperature (0 = greedy)
        1.0,                               // top-p
        1,                                 // seed
        &mut |piece| { out.push_str(piece); true },   // return false to stop early
    )?;
    Ok(out)
}
```

For a file on disk, implement `Source` yourself (`len` and `read_at(offset, buf)`) or enable the `std` feature in the crate for file-backed sources used by its tests. `Llm::generate` returns the number of tokens produced and fails with `Error::Format` if the prompt does not fit the context. The `Error` enum is `Io`, `Format`, `Unsupported` and `Missing`.

The prompt builder and history parser were run against the crate; for the system prompt `be brief` and history `\nhi\n` they produced:

```
OUTPUT
system
be brief
user
hi
assistant



```

Running generation requires a Qwen3 GGUF file that is not part of the repository; that part was checked against the signatures only.

CHAPTER 17

The Toolchain: Compiler Crates, Embedding and Extension

This chapter explains how a `.ling` file becomes a running program, what each crate in the workspace is for, which parts are load-bearing and which are scaffolding, and how to embed or extend the toolchain. It is written for people who read or change the implementation. Everything here comes from the source, and where the repository's own documents disagree with the code, the code wins and the difference is flagged.

The pipeline at a glance

```
OUTPUT
 source text (.ling)
      |  src/lexer      hand-written lexer; keywords in many scripts resolved here
      v
   tokens
      |  src/parser     hand-written recursive-descent parser -> AST
(src/parser/ast.rs)
      v
    AST ------------------------------------------------+
      |  src/mir (flatten `use`/`mod`, lower)           |
      v                                                 |  tree-walking interpreter
    MIR  (crate ling-mir: ir.rs)                        |  (src/runtime/mod.rs)
      |  ling-mir Optimizer (O0..O3)                    |  the semantic reference
and
      v                                                 |  the home of every builtin
  optimized MIR                                         |
      |  ling-codegen                                   |
      +--> Cranelift JIT   (ling run, default)  ---------->+  builtins and oversized
functions
      +--> Cranelift AOT object files (ling build --aot)      call back into the
interpreter
      +--> bytecode + stack VM (lingc compile)
      +--> WasmBackend  (stub: writes only the 8-byte header)
```

Three facts shape everything else.

1. **The interpreter is the language.** Every builtin lives in one very large `match` on the function name in `src/runtime/mod.rs` (the `Interpreter`). The JIT does not reimplement builtins: compiled code calls the runtime function `ling_builtin(name, args)`, which dispatches into the same interpreter, and any function the JIT skips (for being oversized) is run by a "fallback interpreter" primed with the same program. A new builtin therefore works under both backends the moment you add its match arm.
2. **Two front ends exist; one is live.** The `ling` binary uses its own `src/lexer` and `src/parser`. The workspace crates `ling-lex` and `ling-ast` (and `ling-polyglot`, `ling-macro`) are parallel, much smaller efforts that the binary does not call. `ling-mir` and `ling-codegen` are shared and live; they import `BinOp`, `UnOp` and `Span` from `ling-ast`.

3. **Static checking is not on the `ling run` path.** The repository contains `src/semantic` (a type checker) and `src/borrowck`, but nothing in `ling run`, `lingc` or `ling build` calls them: the call in `mir::compile_and_optimize` is commented out and a search for other callers found none. The ownership keywords (`own`, `lend`, `share`, `move`, `copy`) parse, but a program is not rejected for misusing them in this version. Do not rely on the compiler to prove memory or aliasing safety.

Running a program: the modes

Command	What runs	Notes
`ling run file.ling`	Cranelift JIT at optimisation level O3	Default. Falls back to the interpreter if the JIT cannot compile the program.
`ling run --interp file.ling`	The tree-walking interpreter	Semantic reference.
`ling run --O0` (also `--O1`, `--O2`, `--O3`)	JIT at that MIR optimisation level	Escape hatch when O2 or O3 compile time blows up on a giant function.
`ling run --wasm file.ling`	Builds a WebGL/WASM bundle via `lingc webgl`, serves it locally, opens the browser	The WASM program is the **interpreter** (`ling-wasm`) compiled to `wasm32` with your source, not a translation of your code to WebAssembly.
`ling build file.ling --platform ...`	Distributable	Platforms: `web`, `win`, `lin`, `mac`, `macarm`, `pios`, `kernel`, `rpi`, `lingos`, `lingos-rpi`, `all`. `--aot` compiles to native code through Cranelift object files; `--pack` embeds resources.
`lingc mir file.ling [-O 0..3]`	Prints the MIR	The best tool for seeing what the optimiser did.
`lingc compile file.ling [-O n] [--run]`	Writes `file.lingbc` (bytecode); with `--run`, executes it in the VM	See the caveat below.
`ling visualize file.ling`	Prints the AST as an SVG to standard output	`ling ast` renders a whole project into `./AST/`.

Programs that serve requests run on the interpreter. If the source text contains `http_serve` (or its localized names) or `lingtp_serve`, `ling run` chooses the tree-walker regardless of flags. The source comment explains why: handlers are closures passed as call arguments, and the JIT currently lowers closure arguments to `()`. The practical consequence for any program, not only servers, is that **passing a function value as an argument is unreliable under the JIT**. Test such code with `--interp`.

The bytecode VM is not a full backend. `lingc compile fib.ling --run` on a ten-line recursive Fibonacci program reported `Codegen error: unexpected end of code`, although the same program prints `6765` under `ling run` and `ling run --interp`. Treat `lingc compile` as experimental.

Entry point. The program starts at a top-level `bind` whose name is one of the recognised entry names (`start`, `main`, and their translations such as `启`, `เริ่ม`, `시작`, `начать`, `inicio`, `commencer`, `anfang`, `শুরু`, `התחל`, `شروع`, `ابدأ`), or else the first top-level `bind` whose value is a `do { }` block. A function named `main` is not an entry point:

```
OUTPUT
$ ling run e1.ling        # fn main() { print("from fn main") }
error[runtime]: no entry point - need `bind start = do {...}` or `ผูก เริ่ม = ทำ {...}`
$ ling run e2.ling        # bind main = do { print("from bind main") }
from bind main
```

Top-level statements other than declarations are a parse error: `print("x")` at file level fails with `unexpected token at top level`.

Seeing the pipeline

The next program is small enough to follow through every stage. Run it first:

```
fn fib(n) {
  if n  bool }`, returning whether it changed anything; that is the extension point for a new optimisation. `Liveness` and `find_loops` (dominator-based) are available to pass authors.

### Using it from Ling

You meet it through `ling run --O0 ... --O3` and `lingc mir`. The default for `ling run` is O3.

### Using it from Rust

```
use ling_mir::{MirFunction, OptLevel, Optimizer};

fn optimise(funcs: &mut [MirFunction]) {
    Optimizer::new(OptLevel::O2).run(funcs);
}
```

To add a pass, implement `Transform`, push it into the right list in `Optimizer::new` in `crates/ling-mir/src/optimizer.rs` (`scalar_passes` for all levels, or the per-level `late_passes`), and test it with `lingc mir`.

`ling-codegen`

Turns MIR into something that runs. `MirProgram::new(mir, name)` wraps a program; the `CodegenBackend` trait is `fn emit(&mut self, mir: &MirProgram, out: &Path) -> anyhow::Result`.

Backend	Type	Status
Cranelift JIT	`JitBackend`	The default for `ling run`. Compiles every function to machine code in memory; arithmetic on numbers is emitted inline and everything else calls registered runtime symbols (`__ling_f64_add`, `__ling_list_push`, `__ling_builtin`, and so on) which the host registers with `JitBackend::new(|builder| ...)`. `compile(&mir)`, then `run_main()`.
Cranelift AOT	`CraneliftBackend`	Emits object files for `ling build --aot`, for desktop targets and for the bare-metal kernel and LingOS targets (it can include the `ling_kernel_*` and `ling_sys_*` intrinsics, and bundle a program as a library entry). Cross-compiles to aarch64.
Bytecode	`BytecodeBackend`, `Vm`, `compile_mir_program`, `VmProgram`	A small stack VM with about 30 opcodes. Experimental: it could not run a recursive-Fibonacci test (see above).
WebAssembly	`WasmBackend`	**A stub.** `emit` writes a file containing only the eight bytes `00 61 73 6d 01 00 00 00` (the WASM magic and version). There is no WebAssembly code generation from MIR.

Using it from Ling

You use it by running programs. `ling build --aot` and `ling build --platform kernel|lingos` call the AOT backend.

Using it from Rust

```
use ling_codegen::{compile_mir_program, JitBackend, MirProgram, Vm};

// Bytecode path (experimental)
let vm_prog = compile_mir_program(&mir);
let mut vm = Vm::new();
vm.load(vm_prog);
vm.run_main().expect("vm");
```

The JIT needs the host to register the runtime symbols it will call; `ling::core::LingCompiler::run_jit_mir` in the root crate is the reference for the full list, and `ling::run_jit` is the supported entry point.

`ling-runtime`

A tiny library: `LingBox` (a boxed value with `Deref`), `Gc` (a reference-counted pointer, a stand-in for a future tracing collector), raw `raw_alloc` / `raw_dealloc`, and four helpers in `std` (`ling_print`, `ling_debug`, `ling_panic`, `ling_sleep_ms`, `ling_time`). Only `ling-game` and `ling-ui` depend on it. **It is not the interpreter's runtime**: the interpreter, its values and every builtin are in the root crate's `src/runtime`.

Using it from Ling and from Rust

Nothing is exposed to Ling. From Rust it is usable but trivial; use `Arc` and `Box` directly unless you need these exact names.

`ling-macro`

Procedural macros. It contains one derive macro, `#[derive(LingType)]`, whose body is empty: it parses its input and returns no tokens. It is a placeholder for future type-metadata generation and has no effect today.

```
use ling_macro::LingType;

#[derive(LingType)]      // accepted, generates nothing
struct Player { hp: f32 }
```

Using it from Ling

Not applicable.

`ling-polyglot` and the multilingual system

`ling-polyglot` models the idea of language lexicons: `LanguageCode` (17 codes: En, Zh, Ja, Ko, Th, Ru, Hi, Ar, Es, Fr, De, Pt, Vi, El, He, Tr, Pl), `Lexicon` and `LexiconEntry` for canonical-to-native mappings, `translate_keyword(canonical, &lexicon)`, `detect_language`, and a built-in Chinese table. The `ling` binary does not use this crate.

How the multilingual system really works. Four separate mechanisms exist, and only the first is what makes your program run.

1. **The lexer's** `classify_word` (`src/lexer/mod.rs`) recognises keyword spellings for English, Chinese, Japanese, Korean, Thai, Russian, Spanish, French, German and others. This is what lets `令 x = 1` and `bind x = 1` mean the same thing.
2. **Builtin aliases in the interpreter** are extra string patterns on each builtin's match arm in `src/runtime/mod.rs`, for example `"net_status" | "เน็ตสถานะ" | "وضعيت_شبكه" | ...`. Only names written there work. Which scripts have aliases varies by builtin; the reference chapters list the Chinese, Japanese, Korean and Thai ones that exist.
3. **The** `lexicons/*.ling` **files** (en, zh, ja, ko, th, ru, ar, he, fa, ur, fr, de) are TOML-like tables of `canonical = "native"` for keywords, types and builtins. **The compiler does not read them.** The loader in `src/lexicon` is a stub, and `build.rs`'s `generate_lexicon_tables` is empty. A lexicon entry therefore documents intent but does not by itself make a word work: with the default front end, `http_get` is callable in Thai as `เว็บรับ` (a match-arm alias) but not as the lexicon's Chinese `网取`, which evaluates to `()` without an error. Several lexicon builtins (`ai_chat`, `ai_embed`, `http_post`, `ws_connect`) name functions that do not exist at all.
4. `lingfu normalize ` rewrites the keywords and builtins of a whole project into another language using tables inside `crates/ling-fu/src/normalize.rs` (not the `.ling` files). It works for the common keywords: normalising a Fibonacci program to Chinese gives `函 fib(n) { 若 n  Token::Bind`). Keep each word as one lexer word (letters, digits, underscore and combining marks; no spaces).
2. **Entry point.** Add your word for "start" to `ENTRY_NAMES` in `src/entry.rs`, so `bind  = do { }` works.

3. **Builtins.** For each builtin you care about, add a string to its match arm in `src/runtime/mod.rs`. Start with `print` and the I/O builtins. Aliases must not collide with an existing alias.
4. **Diagnostics.** To get error messages in your language, extend `OutputLang` in `src/diag.rs` (currently English, Thai, Korean, Japanese, Chinese, Persian, Arabic, Hebrew, Urdu) and the banner detection in `detect_language` in `src/lib.rs`.
5. **Tooling.** Add a `lexicons/.ling` file (copy `en.ling`, set `canonical = false` and `base = "en"`) so the language is documented, and add your rows to the tables in `crates/ling-fu/src/normalize.rs` so `lingfu normalize ` can produce it.

```
OUTPUT
name = "Français"
code = "fr"
canonical = false
base = "en"

[keywords]
bind = "lier"
do = "faire"
fn = "fonction"
```

`ling-wasm`

A `cdylib` of `wasm-bindgen` entry points for the browser: `init_canvas(OffscreenCanvas)`, `register_module(path, source)` (so `use "path"` works on `wasm32`), `run_program(source)`, `on_key_down`, `on_key_up` and `clear_frame_keys`. It is the interpreter itself compiled to `wasm32` with a WebGL renderer; `ling run --wasm` and `lingc webgl` package it with your source.

Using it from Rust

Build with `wasm-pack` or `cargo build --target wasm32-unknown-unknown`, then call the exports from JavaScript in a worker.

`ling-py`

Python bindings built with PyO3 (0.22), compiled as a `cdylib` named `ling_py`. **It does not run Ling code from Python.** It exposes three engine pieces:

Python name	What it wraps
`AudioEngine`	`ling_audio::AudioEngine`: 4-D positional tone synthesis. Methods `set_tone(idx, x, y, z, w, freq, amp, lfo_rate, lfo_depth)` (all but `idx` optional), `clear_tone(idx)`, `set_listener(cry, sry, crx, srx)`, `load_bgm(path, vol)`, `set_bgm_volume`, `set_master_volume`.
`HyperbolicWorld(radius, curvature, gravity)`	Poincare-ball hyperbolic physics. Methods `gravity_dir`, `gravity_force`, `up_at`, `world_distance`, `to_poincare`, `from_poincare`.
`NetClient`	A background-thread **WebSocket** client (`tokio-tungstenite`) with `connect(url)`, `send(msg)`, `try_recv()` (a string or `None`) and `is_connected()`. Despite the crate description it does not use `ling-net`.
functions	`hyp_distance`, `exp_map`, `log_map` for hyperbolic geometry.

Using it from Ling

Not applicable.

Using it from Rust and Python

Build the module with `maturin develop` (or `maturin build --release`) in `crates/ling-py`; it needs a Python toolchain and was not built in the environment used to test this book. Usage from Python, from the module's own documentation:

```
import ling_py

net = ling_py.NetClient()
net.connect("ws://host:8765")        # non-blocking; spawns a background thread
net.send('{"type": "state"}')
msg = net.try_recv()                 # None, or a str

w = ling_py.HyperbolicWorld(10.0, 1.0, 9.8)
d = ling_py.hyp_distance(0.0, 0.0, 0.0, 0.1, 0.0, 0.0)
```

LingOS: `ling-kernel` and `ling-user` (overview)

LingOS is the bare-metal operating system written in Ling's own toolchain. This is a map, not a manual.

`ling-kernel` is a very large (more than 80,000 lines) `#![no_std]` kernel library. It has a real global allocator over a slab heap and a frame allocator (`mm`), a scheduler and ELF loader (`proc`), a syscall ABI for ring-3 programs (`abi`), and two architecture back ends (`arch/x86_64` and `arch/aarch64`, the latter for Raspberry Pi). Drivers (`drivers/`) include VGA and framebuffer consoles, keyboard, mouse and gamepads, ATA, AHCI and USB mass storage, AC97 and HDA audio, an e1000 network card with its own TCP/IP stack (`netstack`), a TLS 1.3 client built on its own crypto suite (SHA-256, HMAC, HKDF, ChaCha20-Poly1305, X25519), an SSH server (`sshd`), SFTP, a package manager, a window manager, a terminal, a browser, a DAW, and the `llm` driver that runs `ling-llm`. The file system is `lingfs`, content-addressed with BLAKE3. The kernel includes generated `no_std` copies of the compiler front end (`lingc`) and a MIR interpreter, so `ling run` and `ling eval` work inside the Terminal. The `lwp` module is the LWP-1 peer-to-peer transport. The kernel reuses `ling-crypto` built with `--no-default-features --features alloc`; the entropy source is wired to RDRAND with `ling_crypto::rng::set_entropy_source`.

Two limits stated in the source are worth repeating: the kernel has no CSPRNG beyond hardware RNG instructions where available (earlier code seeded a salt from the cycle counter and flags that as unsuitable for key material), and the `ssh_*` builtins are deliberately not available there because the userland network stack was not complete enough to make them safe.

`ling-user` is the userland runtime for programs that run on LingOS: `#![no_std]`, with modules `alloc`, `collections`, `entry`, `gfx`, `io`, `math`, `strings` and `syscall`. `entry` provides `_start`, which calls the compiled `__main__` and exits with its result. The syscall numbers are `SYS_EXIT` 0, `SYS_WRITE` 1, `SYS_READ` 2, `SYS_OPEN` 3, `SYS_CLOSE` 4, `SYS_LSEEK` 5, `SYS_MMAP` 6, `SYS_MUNMAP` 7, `SYS_SPAWN` 8, `SYS_WAITPID` 9, `SYS_YIELD` 10, `SYS_GETPID` 11, `SYS_SLEEP_MS` 12, `SYS_POLL_INPUT` 13, `SYS_FB_MAP` 14 and `SYS_UNAME` 15.

Building. The project kind `kernel` in `Ling.toml` builds a bare-metal image; `lingos` and `lingos-rpi` build applications for the desktop:

```
OUTPUT
$ rustup toolchain install nightly
$ rustup target add x86_64-unknown-none --toolchain nightly
$ ling build examples/kernel --platform kernel
$ qemu-system-x86_64 -kernel dist/kernel/ling-kernel-example
```

```
OUTPUT
[project]
name = "ling-kernel-example"
version = "0.1.0"
entry = "main.ling"
kind = "kernel"
```

```
GUI
bind start = do {
    ling_kernel_vga_clear()
    ling_kernel_vga_write_str("Ling Kernel\n")
    asm!("cli")
    asm!("hlt")
}
```

The kernel intrinsics that bare-metal Ling code can call (from the example's README) are `ling_kernel_vga_clear()`, `ling_kernel_vga_write_str(s)`, `ling_kernel_vga_write_char(c)`, `ling_kernel_serial_write(ptr, len)`, `ling_kernel_halt()`, `ling_kernel_cli()`, `ling_kernel_sti()`, `ling_kernel_inb(port)`, `ling_kernel_outb(port, val)`, `ling_kernel_panic(msg)` and `ling_kernel_init()`. A LingOS desktop program is an ordinary Ling file that draws with the `px_*` functions and is started with `ling run` from the Terminal; `examples/lingos/spiral.ling` is the shipped sample.

The kernel snippet was not built or booted for this book (it needs the nightly toolchain and QEMU); it is checked against the example in the repository only.

Extending the interpreter: adding a builtin

This is the most common extension and it needs only one file. The interpreter dispatches builtins in `Interpreter::call_named_inner(name, args, env)` in `src/runtime/mod.rs`, a single `match name { ... }`.

1. Add an arm, with aliases as extra patterns, and read arguments with the `arg_str`, `arg_num` and `arg_list_str` helpers, which supply defaults for missing arguments:

```
// Returns the number of words in a string.
"word_count" | "คำนับ" => {
    let s = self.arg_str(&args, 0, "");
    return Ok(Value::Number(s.split_whitespace().count() as f64));
},
```

2. Return `Ok(Value::...)` for a result, `Ok(Value::Unit)` for nothing, or `Err(EvalErr::from("message"))` for a runtime error. For a fallible operation that should not abort the program, return `Value::Ok(Box::new(v))` or `Value::Err(Box::new(Value::Str(msg)))`, which Ling code destructures with `ok(v)` and `bad(e)`.
3. Gate platform-specific arms with `#[cfg(not(target_arch = "wasm32"))]`, and optional families with `#[cfg(feature = "...")]`, the way the `web` and `llm` arms are.
4. For state that outlives a call (network connections, models), follow the existing pattern: either an `Interpreter` field, or a process-wide `static` table with integer handles that fail softly on a bad handle.
5. Rebuild and test with both `ling run` and `ling run --interp`.

Because the JIT calls `ling_builtin(name, args)`, which dispatches to this same `match`, nothing else needs to change. Adding a builtin to the `semantic` module's tables is not needed, because that checker is not used.

An unknown name is not an error: it evaluates to `()`. This is convenient for forward compatibility and treacherous for typos, so add a test that calls your new builtin and checks the result.

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.