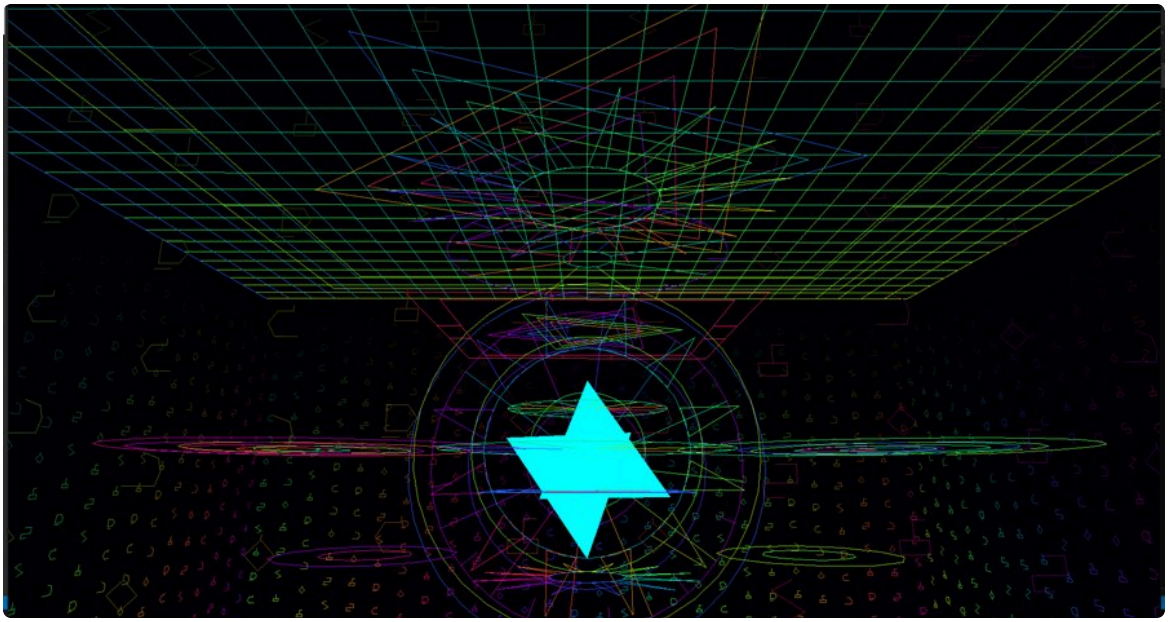
LEARN LING



灵 · 靈 · 령 · āɴ

Learn Ling

Beginner Edition

LING PRESS

Learn Ling: Beginner Edition

Copyright © 2026 Ling Press. All rights reserved.

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

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

First edition, 2026.

ISBN: TBD

Contents

WELCOME 5

Preface 6

How to Use This Book 10

LEARN LING 15

1. Welcome to Ling 16
2. Values and Bindings 27
3. Decisions and Loops 41
4. Functions 56
5. Lists and Text 73
6. Your First 3D Room 91
7. Drawing and Games 115
8. Sound and Music 130
9. Your First Project 147

REFERENCE 161

10. Keywords 162
11. Syntax, Types and Semantics 176
12. Core Builtins Reference 194
13. Command Line Tools: `ling` and `lingfu` 220
14. Window, Frame Loop, Input and Camera 229
15. Shapes, Vector Textures, Pixel Textures and Effects 246
16. Vector Fonts, the UI Toolkit and Dialogs 254
17. Animation: Anima 263
18. Audio, Music and the DAW 270
19. Physics, Game Helpers and the Body Toolkit 284
20. Input, Microphone and Speech-to-Text 294

SHOWCASE 300

21. Case Studies: Applications Built in Ling 301

Known Limitations and Gotchas 317

PART

Welcome

Install Ling, write your first program, and learn how this book works.

Preface

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

Who this book is for

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

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

What is Ling?

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

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

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

The three editions

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

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

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

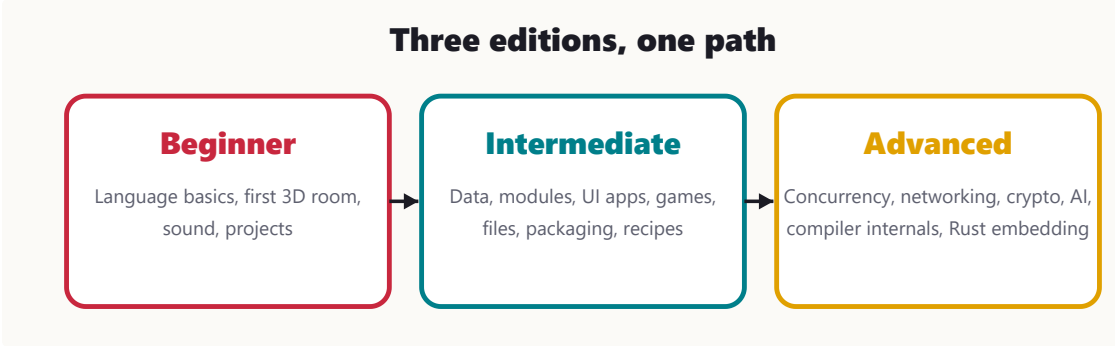


Figure: The Beginner edition covers the language basics, your first 3D room, sound and projects. Intermediate covers data, modules, UI apps, games, files, packaging and recipes. Advanced covers concurrency, networking, crypto, AI, compiler internals and Rust embedding.

How this book is organised

The nine tutorial chapters build on each other.

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

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

How to get the code and the examples

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

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

Conventions

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

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

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

A last word before you begin

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

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

Welcome to Ling.

How to Use This Book

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

Code listings

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

Complete programs

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

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

```
OUTPUT
Hello, world!
```

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

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

Windowed (GUI) programs

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

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

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

Fragments

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

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

Terminal sessions

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

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

Callouts

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

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

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

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

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

Chapter structure

Every chapter has the same shape.

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

Figures

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

Version note

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

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

Spelling and language

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

Typing the programs

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

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

Now turn the page and begin.

PART

Learn Ling

Nine hands-on chapters from your first line of code to a finished project.

CHAPTER 1

Welcome to Ling

What you'll learn. In this chapter you will find out what Ling is and why it can be written in five (and more) human languages. You will install it, run your first program, and learn the edit-run loop that you will repeat thousands of times in your programming life. You will also learn how this book is laid out so you can find your way around it.

What is Ling?

A computer program is a list of instructions written for a machine that follows them exactly, quickly, and without ever getting bored. A *programming language* is the vocabulary and grammar you use to write those instructions down. Python, JavaScript, Rust and C are programming languages. **Ling** is one more, and it has a rare property: it is *polyglot*.

Most programming languages are written in English. The words `if`, `while`, `return` and `function` come from English, and a student who does not speak English must learn those words before learning anything else. In Ling, every keyword has several spellings, one for each of many human languages, and you can use whichever you like. The program below says hello in English:

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

```
OUTPUT
Hello, world!
```

Here is the same program written with Chinese keywords:

```
令 start = 执 {
  println("你好，世界！")
}
```

```
OUTPUT
你好，世界！
```

The words `令` and `执` are the Chinese spellings of `bind` and `do`. Nothing else changed except the greeting. Japanese, Korean and Thai work in just the same way:

```
束縛 start = 実行 {
  println("こんにちは、世界！")
}
```

```
OUTPUT
こんにちは、世界！
```

```
바인드 start = 실행 {
  println("안녕하세요, 세계!")
}
```

```
OUTPUT
안녕하세요, 세계!
```

```
ผูก start = ทำ {
  println("สวัสดีชาวโลก!")
}
```

```
OUTPUT
สวัสดีชาวโลก!
```



Figure: The same keywords in five languages. All of these spellings are valid, even in the same file.

When you run a non-English file, Ling prints a small notice such as `[detected language: Chinese (中文)]` on a separate channel (called "standard error"), so it does not mix with your program's own output. You can safely ignore it.

You can even mix spellings inside one file. This is allowed:

```
// Keywords from different languages can share one file
令 start = do {
  println("English and Chinese, together")
}
```

```
OUTPUT
English and Chinese, together
```

> **NOTE**
>
> Ling also accepts keyword spellings in French, German, Spanish, Portuguese, Russian, Arabic, Persian, Hebrew, Urdu and Hindi. In this book we use the English spellings because they are the most common, and because the book itself is written in English. The Reference part of the book lists every keyword in every supported language.

Being polyglot is a friendly feature, but it would not matter much if Ling could not

do

anything. It can. The same language drives windows, 3D graphics, sound and music, networking, databases and games. In the beginner chapters of this book you will write text programs first, and then build a 3D room, a small game and a tune. All of it with the same handful of ideas.

Where does the name come from?

Ling (灵) is a Chinese word that can mean "spirit", "spiritual" or "quick-witted". The command-line program that runs your code is called `ling`, and the project manager that creates and packages bigger projects is called `lingfu` (灵符, "spirit token"). You will meet both of them soon.

Installing Ling

Ling is installed from the command line. If you have never used a command line (also called a *terminal* or *shell*), do not worry: it is a window where you type a command, press Enter, and the computer replies. You will only need a few commands.

> TIP
>
> On Windows, open the Start menu and type "PowerShell". On macOS, open the app called "Terminal". On Linux, press Ctrl+Alt+T or look for "Terminal" in your applications.

The one-line installers

The project provides an installer for each family of systems. It first makes sure that a Rust toolchain is present (Ling is built from Rust source code), and then builds `ling` and `lingfu` for you.

On **macOS** and **Linux**:

```
curl --proto '=https' --tlsv1.2 -sSf https://ling-lang.org/install.sh | sh
```

On **Windows**, in PowerShell:

```
irm https://ling-lang.org/install.ps1 | iex
```

The first build takes a few minutes, because your computer is compiling real software. Make a cup of tea.

If you already have Rust

If you already use Rust, you can install both tools directly from the Rust package registry:

```
cargo install ling-lang lingfu
```

After installing, the programs `ling` and `lingfu` are placed in a folder called `.cargo/bin` inside your home folder. The installers normally add that folder to your `PATH`, which is the list of places your terminal searches for programs.

Checking that it worked

Ling has no `--version` flag; instead, if you run `ling` with no arguments it prints a help text, and the first line tells you which version you have:

```
OUTPUT
$ ling
ling 2030.1.53 — The Omniglot Systems Language
Usage:
  ling run <file.ling>                run using the Cranelift JIT backend (default)
  ling run --interp <file.ling>       run using the tree-walking interpreter
  ...
```

If your terminal says that the command `ling` is not found, close the terminal and open a new one (the `PATH` is only read when a terminal starts). If it still fails, check that the `.cargo/bin` folder is on your `PATH`.

> NOTE
>
> Every example in this book was run with **Ling 2030.1.53**. Newer versions may add features. They should not break the examples, but if a program behaves differently on your machine, check your version first.

Your first program

Programs are ordinary text files. Do not use a word processor for them: Word and Pages add hidden formatting that Ling cannot read. Use a plain text editor. Notepad works, and so does any code editor (more on those below).

1. Create a new folder somewhere easy to find, for example `ling-book` in your Documents folder.
2. Inside it create a file called `greet.ling`. The extension `.ling` is the convention for Ling files.
3. Type or paste the following text into the file and save it.

```
# My first real program
bind start = do {
  println("Hello, world!")
  println("My name is Ling.")
  println("2 + 2 = ", 2 + 2)
}
```

```
OUTPUT
Hello, world!
My name is Ling.
2 + 2 = 4
```

4. Open a terminal *in that folder*. (On Windows 11 you can right-click inside the folder and choose "Open in Terminal"; on a Mac, drag the folder onto the Terminal icon.)
5. Type `ling run greet.ling` and press Enter.

You should see exactly the output shown above. Congratulations. You are now a programmer.

Reading the program

Let us look at it line by line, because every Ling program in this book has the same skeleton.

- `# My first real program` is a **comment**. Everything from a `#` to the end of the line is ignored by Ling. Comments are notes to human readers. You can also start a comment with `//`.
- `bind start = do { ... }` is the **entry point**. Think of it as the front door of your program: when you run a file, Ling looks for a binding called `start` (or `main`) and runs the instructions between the curly braces `{` and `}`. The word `bind` means "give this name to this thing". Here, the name `start` is given to a block of instructions (`do { ... }`).
- `println("Hello, world!")` is a **function call**. `println` is a function that prints a line of text on your screen. What you want printed goes in parentheses.
- `"Hello, world!"` is a piece of **text**, written between double quotes. Programmers call text a *string*.

The very last line, `println("2 + 2 = ", 2 + 2)`, shows something interesting: Ling did the arithmetic for you. It computed `2 + 2`, got 4, and printed it right after the text. Notice that there is no space or separator inserted between the two things you gave to `println`; that is why the text ends in `=` with a space. We will come back to this in the next chapter.

> **WARNING**
>
> The entry point must be a `bind` of a `do { }` block. If you write `fn main() { ... }` as you would in some other languages, Ling will say "no entry point" and do nothing. And top-level lines such as a bare `println("hi")` outside any block are a parse error. Everything you want to *run* goes inside `bind start = do { ... }`.

The edit-run loop

Programming is a loop with three steps, and you will repeat it more often than you can count:

1. **Edit** the file in your editor and save it.
2. **Run** it with `ling run yourfile.ling`.
3. **Look** at what happened, think, and go back to step 1.

Professionals do this too. They are simply faster at it. The most important habit you can form is to run your program *often*, after every small change, instead of writing fifty lines and hoping they all work. If something breaks, you will know it was your latest small change.

Try it now. Change the greeting in `greet.ling`, save the file, and run it again:

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

Did your change appear? If yes, you have mastered the loop.

When things go wrong

You will make mistakes; everybody does. Ling reports a mistake with a message that starts with `error[parse]` (the program is not written in a form Ling understands) or `error[runtime]` (the program was understandable but failed while running). Here is a classic one. In many languages you change a variable by writing `x = x + 1`. Ling has no such statement:

FRAGMENT

```
bind start = do {
  bind x = 5
  x = x + 1
  println(x)
}
```

OUTPUT

```
$ ling run err1.ling
error[parse]: unexpected token in expression: Eq
 --> err1.ling
```

The message is not pretty, but it points at the problem: an unexpected `=` (the token called `Eq`). In Ling the way to "change" a value is to bind the name again: `bind x = x + 1`. We will study this in Chapter 2.

Here is another classic: a missing closing parenthesis.

OUTPUT

```
$ ling run err2.ling
error[parse]: unexpected token in expression: RBrace
 --> err2.ling
```

Here Ling reached the closing brace `}` of the block while it was still waiting for the `)` that should end `println("hi"`. The message names the thing it *found* (`RBrace`) rather than what is missing. A good rule of thumb: when the error mentions a token that seems innocent, look at the line *just before* it for a missing bracket or quote.

TIP

Read error messages from the top. Fix the first error, then run again. Later errors are often just echoes of the first one.

Choosing an editor

You can write Ling in any text editor. A *code editor* makes it more pleasant by colouring the words (called *syntax highlighting*), matching your brackets, and offering to finish words as you type. Popular free choices are Visual Studio Code, Sublime Text and Notepad++.

For **Visual Studio Code**, the Ling project provides an extension named `ling-lang` (found in the Ling repository under `editors/vscode-ling`). It offers:

- syntax highlighting for the English keywords and for all the other languages,
- auto-completion for the keywords and every built-in function, with a one-line description when you hover over a name,
- hints showing the parameters of a function as you type its call,
- a small "Ling Dark" colour theme,
- code snippets: type `start` and press Tab to get the skeleton of a program.

It also warns you when you call a function that does not exist. This is helpful for a reason you will meet in Chapter 9: Ling itself does not always complain about misspelled function names.

To install it, look for `ling-lang` in the Extensions view of VS Code. If it is not listed in the marketplace yet, install it from the file in the repository: run `code --install-extension ling-lang-0.1.4.vsix` in the folder `editors/vscode-ling`.

> **TRY IT**
>
> Open your `greet.ling` in a code editor with the extension installed. Hover over `println` and read the description that pops up. Then misspell `println` as `printn` and see what the editor says.

How to read this book

This book is the **Beginner Edition** of a three-part series.

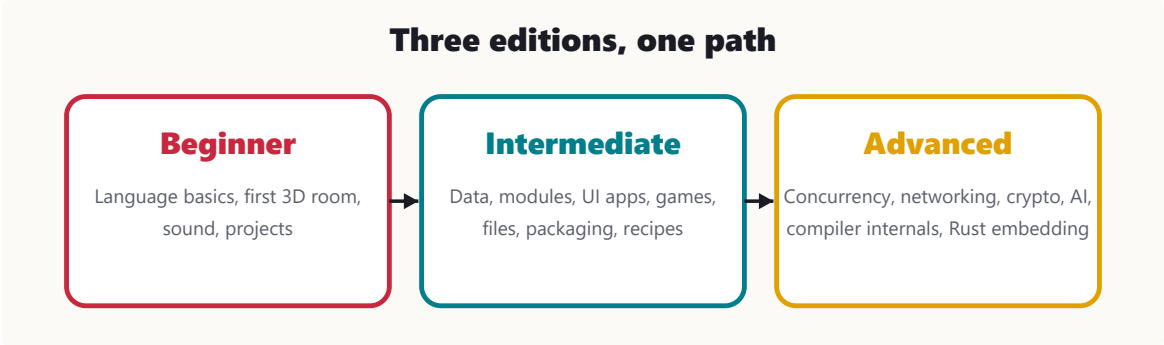


Figure: The Beginner edition covers the language basics, your first 3D room, sound, and your first project. The Intermediate edition goes into data, modules, UI apps, games, files, packaging and recipes. The Advanced edition covers concurrency, networking, cryptography, AI, compiler internals and embedding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

The chapters build on each other, so read them in order the first time. After that, use the Reference, which is organized by topic, when you need to look something up.

Each chapter has the same shape:

- **What you'll learn** at the top, so you know what to expect.
- Short sections with small, complete programs. **Every program is complete.** You can copy it into a file and run it. The output printed below a program is the real output we got.
- Four kinds of callout boxes: **Tip** (a shortcut or good habit), **Note** (extra background), **Warning** (a trap that will cost you time) and **Try it** (a small experiment for you to do right now).
- A **Chapter summary** with the main points.
- **Exercises** with answers at the end. Do try them before reading the answers. Programming is learned with the fingers, not the eyes.

Programs which open a window or play sound are labelled as such, so you know to expect something on your screen instead of text in the terminal. The next section of the book, "How to Use This Book", explains all the visual conventions in detail.

> TIP
>
> Type the examples by hand at least for the first few chapters instead of pasting them. Typing forces you to read every character, and typing errors are a wonderful teacher.

Chapter summary

- Ling is a polyglot language: its keywords can be written in English, Chinese, Japanese, Korean, Thai and more, even within a single file.

- Install it with the one-line installer or with `cargo install ling-lang lingfu`; the programs are `ling` (runs your code) and `lingfu` (manages projects).
- A program is a plain text file ending in `.ling`, run with `ling run file.ling`.
- Execution starts at `bind start = do { ... }`.
- `println(...)` prints a line; `#` and `//` start comments.
- The edit-run loop is: edit, save, run, look, repeat. Run often.
- Error messages starting with `error[parse]` mean Ling could not understand the text; `error[runtime]` means it failed while running.

Exercises

1. Make a file `about.ling` that prints three lines: your name, your favourite colour and your favourite number.
2. In `greet.ling`, delete one of the quotation marks around `"Hello, world!"` and run it. What does Ling say? Put the quote back.
3. Change the line `println("2 + 2 = ", 2 + 2)` so that it prints the result of `100 / 8`.
4. Write the hello-world program with the keywords `令` and `执` instead of `bind` and `do`. Does the output change?
5. Add a comment to the top of one of your programs using `#`, and another using `//`. Confirm that both are ignored.

Answers

1. For example: `ling-module bind start = do { println("Mina") println("green") println(7) }`
2. You get an `error[parse]` message. The exact wording varies, because the missing quote makes Ling treat the rest of the file as part of the text. Putting the quote back fixes it.
3. `println("100 / 8 = ", 100 / 8)` prints `100 / 8 = 12.5`.
4. The output is identical (Ling may print `[detected language: Chinese (中文)]` on the side). Only the spelling of the keywords changed.
5. Both `# note` and `// note` are comments; the program behaves exactly as before.

CHAPTER 2

Values and Bindings

What you'll learn. In this chapter you will meet the basic kinds of information a Ling program works with (numbers, text and true/false answers), learn how to give them names with `bind`, and do arithmetic with them. You will also discover how printing works, including one surprising detail that catches every beginner exactly once.

Values: the things programs know about

Every program is about *something*: the score in a game, the name of a player, the temperature outside. In programming, a single piece of information is called a **value**. The three kinds of value you will use most are:

Kind	What it holds	Examples
number	any number, whole or with a fraction	`42`, `3.14`, `-7`
text	letters, words, sentences; also called a *string*	`"hello"`, `"Room 101"`
boolean	one of exactly two answers: yes or no	`true`, `false`

Let us print one of each:

```
bind start = do {
  println(42)
  println(3.14)
  println("a piece of text")
  println(true)
  println(false)
}
```

```
OUTPUT
42
3.14
a piece of text
true
false
```

Notice three things. Numbers are written bare, with no quotes. Text is written between double quotes. And `true` and `false` are written in lower case with no quotes, because they are not text: they are the two answers to a yes-or-no question.

> **NOTE**
>
> Ling is *dynamically typed*: it does not make you declare in advance what kind of value a name will hold. The name `x` can hold a number now and text later. Ling will not stop you, though it will not protect you either, so keep your values tidy.

A few small rules about how to write values:

- Numbers use ordinary digits and at most one decimal point: `3.14` is fine. Ling has no scientific notation (`1e3`), no hexadecimal (`0xff`), no digit separators (`1_000`), and you cannot write `.5` or `3.`. Write `0.5` and `3.0`.
- Text uses double quotes only. Single quotes are not text delimiters. Text can span several lines, and it understands a few escape sequences that start with a backslash: `\n` for a new line, `\t` for a tab, `\"` for a double quote, and `\\` for a backslash itself.
- Ling also has *lists* (Chapter 5), a "nothing" value written `()`, and more. We will meet them when we need them.

Bindings: giving names to values

If you had to type `3.14159265358979` every time you needed pi, you would soon lose your patience. Instead you give the value a **name** and use the name from then on. In Ling that is called a **binding**, and you make one with `bind`:

```
bind start = do {
  bind name = "Mina"
  bind age = 12
  println("Hello, " + name)
  println(name, " is ", age)
  println("next year: ", age + 1)
}
```

```
OUTPUT
Hello, Mina
Mina is 12
next year: 13
```

You can read `bind name = "Mina"` aloud as "bind the name `name` to the text Mina". Think of a name as a sticky label that you put on a value. Wherever you write the label, Ling reads the value stuck to it.

Names may contain letters, digits and underscores, and must not start with a digit. Use names that say what the value *means*: `age` is better than `a`, and `tip_percent` is better than `t`. Programmers write for other humans first and for the computer second, and that includes the human you will be next month.

Rebinding: how Ling "changes" things

Here is the biggest difference between Ling and many other languages. **Ling has no assignment statement.** You cannot write `score = score + 5`. Instead, you bind the name again:

```
bind start = do {
  bind score = 10
  println("score is ", score)
  bind score = score + 5
  println("score is ", score)
  bind score = score * 2
  println("score is ", score)
}
```

```
OUTPUT
score is 10
score is 15
score is 30
```

Read `bind score = score + 5` as: "work out what `score + 5` is, using the current value of `score`, then stick the label `score` onto the result". The old value 10 is not modified; the label is just moved to a new value. The right-hand side is always worked out first.

If you prefer a physical analogy, think of a row of labelled jars. `bind` does not pour new contents into a jar; it throws away the old label and sticks the same name onto a new jar.

> **WARNING**
>
> Writing `score = score + 5` is a parse error in Ling. If you come from another language, this will bite you more than once. The cure is a habit: whenever you want to change something, type `bind`.

Printing

You have already used `println`. There are two printing functions, `print` and `println`, and in Ling they behave identically: both finish with a new line. When you give them several things, separated by commas, they print all of them one after the other:

```
FRAGMENT
println(name, " is ", age)
```

prints `Mina is 12`, as the previous example showed. Here is the surprise that I promised you.

> **WARNING**
>
> `println` puts **nothing** between the things you give it. Not a space, not a comma. `println("sum", 1 + 2)` prints `sum3`, not `sum 3`. If you want a space, put the space inside your text.

```
bind start = do {
  println("sum", 1 + 2)
  println("a", "b", "c")
  println("sum: ", 1 + 2)
  println("line one\nline two")
  println("She said \"hi\"")
}
```

```
OUTPUT
sum3
abc
sum: 3
line one
line two
She said "hi"
```

The first line prints `sum3`, the second `abc`, and the third `sum: 3` because we included `": "` in the text ourselves. The last two examples show escapes: `\n` starts a new line in the middle of the text, and `\"` puts a double quote inside text without ending it.

Joining text with `+`

You can also join pieces of text into one bigger text using `+`. This is called *concatenation*:

```
FRAGMENT
println("Hello, " + name)
println("Count: " + to_str(7))
```

When one of the pieces is a number, convert it to text first using `to_str(...)` (the short name `str(...)` also works).

Building messages with `format`

When a message has many gaps to fill, `format` is tidier. It takes a template containing `{}` placeholders, and fills them in with the remaining arguments, in order. The special placeholder `{:.2}` shows a number with two decimal places:

```ling
bind start = do {
  bind radius = 5
  bind area = pi() * radius * radius
  println("Area: ", round(area * 100) / 100)
  println(format("Area is about {:.2}", area))
  println(format("{} + {} = {}", 2, 3, 2 + 3))
  println("Count: " + to_str(7))
  println(to_number("12") + 1)
  println(len("hello"))
}
```

OUTPUT

```
Area: 78.54
Area is about 78.54
2 + 3 = 5
Count: 7
13
5
```

`round(area * 100) / 100` is a common trick for rounding to two decimals: multiply by 100, round to a whole number, divide by 100. `format("...{:.2}...", area)` does the same job more neatly. Notice that `to_number("12")` turns text into a number, and `len("hello")` counts the characters in a text.

NOTE

`format` only understands `{}` and `{:.N}` where N is a number of decimals. There is no padding (`{:5}`), no numbered placeholders (`{0}`), and no way to write a literal brace.

Comments

A comment is a note for humans. Everything after a `#` or after `//` on a line is ignored. Use them to explain *why* you did something, not to repeat *what* the code obviously does:

FRAGMENT

```ling
# The shop gives 10 percent off on Fridays
bind discount = 10   // percent
```

Ling has no block comments (the `/* ... */` of other languages). To comment out several lines, put `#` at the start of each.

Arithmetic

Ling does arithmetic with the usual symbols. Let us try them all:

```
# Arithmetic playground
bind start = do {
  println(7 + 3)
  println(7 - 3)
  println(7 * 3)
  println(7 / 2)
  println(7 % 2)
  println(2 + 3 * 4)
  println((2 + 3) * 4)
  println(-5 + 2)
  println(10 / 4)
}
```

```
OUTPUT
10
4
21
3.5
1
14
20
-3
2.5
```

The operators are `+` (add), `-` (subtract), `*` (multiply), `/` (divide) and `%` (the remainder after division, also called *modulo*). Look at what the outputs tell us:

- `7 / 2` is `3.5`. Division always gives an exact answer, not a rounded-down whole number. If you want the whole-number part, use `floor`, which we meet shortly.

- `7 % 2` is `1`, because 2 goes into 7 three times with 1 left over. The remainder operator is surprisingly useful. `n % 2 == 0` is the test for "n is even", and a value `% 10` gives the last digit.
- `2 + 3 * 4` is `14`, not `20`. Multiplication and division happen before addition and subtraction, just as in school mathematics. Use parentheses to say what you mean: `(2 + 3) * 4` is `20`.
- Negative numbers work as you expect: `-5 + 2` is `-3`.

Order of operations in full, from strongest to weakest: a minus sign in front of a number, then `* / %`, then `+ -`, then comparisons, then `&&`, then `||`.

> **TIP**
>
> When you are not sure about the order, add parentheses. They cost nothing, and they tell the reader what you meant.

A small example: splitting a bill

Let us put bindings and arithmetic to work. Four friends share a restaurant bill with an 18 percent tip:

```
# A tiny bill splitter
bind start = do {
  bind bill = 84.50
  bind tip_percent = 18
  bind people = 4

  bind tip = bill * tip_percent / 100
  bind total = bill + tip
  bind each = total / people

  println("Bill:     ", bill)
  println("Tip:      ", round(tip * 100) / 100)
  println("Total:    ", round(total * 100) / 100)
  println("Per head: ", round(each * 100) / 100)
  println(format("Each person pays {:.2}", each))
}
```

```
OUTPUT
Bill:     84.5
Tip:      15.21
Total:    99.71
Per head: 24.93
Each person pays 24.93
```

Every line does one small thing, and every step has a name. That is the secret of readable programs: break a big calculation into small named pieces. Notice the blank lines. Ling ignores them, but they let us group the inputs, the calculation and the report.

A word about decimal numbers

Computers store fractions in binary, and some tidy-looking decimal numbers cannot be stored exactly. You will occasionally see this:

```
bind start = do {
  println(0.1 + 0.2)
  println(1 / 3)
  println(100 * 1.1)
}
```

```
OUTPUT
0.30000000000000004
0.3333333333333333
110.00000000000001
```

This is not a bug in Ling. Every programming language that uses standard floating-point numbers behaves the same. The tiny errors are far too small to matter in games and drawings; when you need to show a number to people, round it, as we did in the bill splitter.

Math functions

Beyond the five operators, Ling provides a toolbox of math functions. A function is called by writing its name and, in parentheses, the values it needs (its *arguments*). Here are the ones you will use most:

```ling
bind start = do {
  println(sqrt(81))
  println(pow(2, 10))
  println(abs(-7.5))
  println(floor(3.9))
  println(ceil(3.1))
  println(round(2.5))
  println(round(3.14159 * 100) / 100)
  println(min(4, 9))
  println(max(4, 9))
  println(clamp(15, 0, 10))
  println(pi())
  println(sin(0))
  println(cos(0))
}
```

```
OUTPUT
9
1024
7.5
3
4
3
3.14
4
9
10
3.1415926535897932
0
1
```

Function	What it does
`sqrt(x)`	square root: `sqrt(81)` is 9
`pow(a, b)`	a raised to the power b
`abs(x)`	drop the minus sign
`floor(x)` / `ceil(x)`	round down / round up to a whole number

Function	What it does
`round(x)`	round to the nearest whole number
`min(a, b)` / `max(a, b)`	the smaller / the larger of two numbers
`clamp(x, low, high)`	force x into the range from low to high
`pi()`	the number 3.14159...
`sin(x)`, `cos(x)`	trigonometry, angles in radians

Two details deserve attention. First, `pi` needs its parentheses: it is the function `pi()`, not a bare name. Second, `sin` and `cos` use *radians*, where a half circle is `pi()`, not 180. You will use them to make things circle and wobble in the graphics chapters.

> TRY IT
>
> Predict the output of `println(round(2.5))`, `println(round(-2.5))` and `println(floor(-2.5))`, then run them. Were you right?

The full list of math builtins, including `tan`, `exp`, `ln`, `log10`, `atan2`, `lerp` and `smoothstep`, is in the Reference.

Comparisons and booleans

A comparison asks a yes-or-no question and gives a boolean answer.

```
bind start = do {
  println(5 > 3)
  println(5 == 5)
  println(5 != 5)
  println("cat" == "cat")
  println(true && false)
  println(true || false)
  println(!true)
  bind big = 100 > 50
  println("big is ", big)
}
```

```
OUTPUT
true
true
false
true
false
true
false
big is true
```

The comparison operators are `==` (equal), `!=` (not equal), `<`, `>`, `<=` and `>=`. A single `=` is never a comparison in Ling; it appears only in `bind`. The logic operators are `&&` ("and": both must be true), `||` ("or": at least one must be true) and `!` ("not": flip the answer).

We will use these answers to make decisions in the next chapter.

> **WARNING**
>
> Do not chain comparisons the way you do in mathematics. `1 < x < 10` does not mean "x is between 1 and 10" in Ling. Write `x > 1 && x < 10`.

Mistakes that fail quietly

Because Ling is forgiving, a few mistakes do not produce an error message. Here is one:

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

You might hope for `1500`. The number syntax stops at `1.5`, so Ling reads `e3` as a separate name, which has no value. On the default engine, the program prints `1.5()`. The interpreter (which we meet in Chapter 9) is stricter and stops with `error[runtime]: undefined: 'e3'`. Another quiet one: dividing by zero gives `()` on the default engine and `inf` in the interpreter. If you ever see a `()` in your output where you expected a number, suspect a typo or a mistake like these.

Chapter summary

- A *value* is one piece of information: a number (`42`, `3.14`), text (`"hello"`), or a boolean (`true` or `false`).
- `bind name = value` gives a value a name. To "change" a name, bind it again: `bind score = score + 1`. There is no `x = 1`.
- `print` and `println` are the same; they add a new line and put nothing between their arguments.
- Join text with `+` (convert numbers with `to_str`), or fill gaps with `format("{} and {:.2}", a, b)`.
- Comments start with `#` or `//`.
- Arithmetic: `+ - * / %`, with the usual precedence; parentheses for clarity.
- Handy functions: `sqrt`, `pow`, `abs`, `floor`, `ceil`, `round`, `min`, `max`, `clamp`, `pi()`, `sin`, `cos`.
- Comparisons (`== != < > <= >=`) produce booleans, which combine with `&&`, `||` and `!`.
- Rounding for display: `round(x * 100) / 100`.

Exercises

1. Write a program that binds your age to a name and prints how old you will be in 10 years, in the form `In 10 years I will be 22`. (Remember: `println` adds no spaces for you.)
2. A rectangle is 12.5 wide and 8 tall. Bind these to names and print its area and its perimeter, each with a label.
3. Convert 98.6 degrees Fahrenheit to Celsius. The formula is `(f - 32) * 5 / 9`. Print the answer rounded to one decimal place.
4. What do `17 % 5`, `17 / 5`, `floor(17 / 5)` and `pow(2, 8)` print? Work them out on paper first, then check by running.
5. Bind `n` to 10. Using only `bind`, make `n` become 11, then 22, then 4 (by adding one, doubling, and then subtracting 18). Print `n` after each step.

Answers

1. `ling-module bind start = do { bind age = 12 println("In 10 years I will be ", age + 10) }` This prints `In 10 years I will be 22`.
2. `ling-module bind start = do { bind w = 12.5 bind h = 8 println("Area: ", w * h) println("Perimeter: ", 2 * (w + h)) }` This prints `Area: 100` and `Perimeter: 41`.

3. `bind c = (98.6 - 32) * 5 / 9` followed by `println(round(c * 10) / 10)` prints `37`. (98.6 degrees Fahrenheit is body temperature, 37 degrees Celsius.)
4. `17 % 5` is 2, `17 / 5` is 3.4, `floor(17 / 5)` is 3 and `pow(2, 8)` is 256.
5. `ling-module bind start = do { bind n = 10 bind n = n + 1 println(n) bind n = n * 2 println(n) bind n = n - 18 println(n) }` This prints 11, 22 and 4.

CHAPTER 3

Decisions and Loops

What you'll learn. Programs become interesting when they can choose between actions and repeat actions. In this chapter you will learn `if` and `else` to make decisions, `while` and `for` to repeat work, and `match` to pick between many cases. You will also learn two traps that catch nearly every Ling beginner, and how to walk around them.

Making decisions with `if`

Imagine a robot following written instructions: "If it is hot, drink water." The robot checks a condition, and only if the condition is true does it perform the action. The Ling version looks almost the same:

```
bind start = do {
  bind temperature = 31
  if temperature > 30 {
    println("It is hot. Drink water.")
  }
  println("Done checking.")
}
```

```
OUTPUT
It is hot. Drink water.
Done checking.
```

The pattern is `if condition { ... }`. The condition is any expression that produces `true` or `false`, usually a comparison like `temperature > 30`. If it is true, the instructions inside the braces run. If it is false, Ling skips them. The last `println` is outside the braces, so it always runs.

Notice there are **no parentheses** around the condition and the braces are **required**, even for a single instruction.

> TIP
>
> Change `temperature` to `20` and run again. The first message disappears and only `Done checking.` is printed. Experiments like this are the quickest way to understand a program.

`else`: the other road

Often you want one thing to happen when the condition is true, and a different thing when it is false. That is `else`:

```
bind start = do {
  bind age = 15
  if age >= 18 {
    println("Adult ticket")
  } else {
    println("Youth ticket")
  }
}
```

```
OUTPUT
Youth ticket
```

Exactly one of the two blocks runs, never both and never neither. Think of a fork in the road.

Many roads: `else if`

You can chain decisions. Ling tests the conditions from top to bottom and runs the first block whose condition is true. The rest are skipped.

```
bind start = do {
  bind score = 83
  if score >= 90 {
    println("Grade A")
  } else if score >= 80 {
    println("Grade B")
  } else if score >= 70 {
    println("Grade C")
  } else {
    println("Keep practising")
  }
}
```

```
OUTPUT
Grade B
```

An 83 is not at least 90, so the first test fails. It is at least 80, so the second succeeds, and Ling prints `Grade B` and ignores the remaining branches (even though 83 is also at least 70). Because order matters, always put the most specific or the largest threshold first.

> **WARNING**
>
> If you wrote the grade tests in the opposite order (`score >= 70` first), every score from 70 up would be a "C". Read your chain from the top and ask at each step: "can this branch steal a case that belongs to a later one?"

Combining conditions

The logic operators from the last chapter let you ask compound questions. Remember: `&&` means "and", `||` means "or", and `!` means "not".

```
bind start = do {
  bind n = 7
  if n % 2 == 0 {
    println(n, " is even")
  } else {
    println(n, " is odd")
  }
  bind hour = 14
  if hour >= 9 && hour < 17 {
    println("The shop is open")
  }
  bind raining = true
  bind have_umbrella = false
  if raining && !have_umbrella {
    println("You will get wet")
  }
}
```

OUTPUT

```
7 is odd
The shop is open
You will get wet
```

`n % 2 == 0` is the classic evenness test: an even number leaves no remainder when divided by 2. The shop is open when the hour is 9 or later *and* before 17. And you get wet if it is raining *and* you do not have an umbrella (`!have_umbrella` flips `false` to `true`).

NOTE

Comparisons and logic combine in a fixed order: arithmetic first, then comparisons, then `&&`, then `||`. In `hour >= 9 && hour < 17` the two comparisons are worked out first and then joined. When in doubt, use parentheses.

Bindings inside `if`

Names that you bind inside an `if` block stay alive after the block ends. In some languages, a variable created inside braces disappears at the closing brace; in Ling, the `if` (and the `while`) blocks share the surrounding scope. That means you can write this and it works as you would hope:

```
FRAGMENT
if score > 50 {
  bind result = "pass"
} else {
  bind result = "fail"
}
println(result)
```

We will use this in the Collatz example below.

Repeating with `while`

A `while` loop repeats a block as long as its condition remains true. Picture a child on a swing: "while I am still swinging, keep pushing".

```
bind start = do {
  bind i = 1
  while i <= 5 {
    println("Count: ", i)
    bind i = i + 1
  }
  println("Liftoff!")
}
```

```
OUTPUT
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
Liftoff!
```

Study the shape of this loop, because you will write it constantly:

1. Set a **counter** before the loop (`bind i = 1`).
2. State the **condition** that keeps the loop going (`i <= 5`).
3. Inside the loop, do the work **and move the counter forward** (`bind i = i + 1`).

Step 3 is the one that beginners forget. If you never change `i`, the condition stays true forever and the loop never ends: an **infinite loop**. If this happens, your terminal appears frozen. Press **Ctrl+C** to stop the program.

> **WARNING**
>
> An infinite loop in a text program is annoying; in a window program (Chapter 6) it can lock up the window. Always check that something inside your `while` changes the thing that its condition looks at.

Accumulating a result

The most useful trick with `while` is the **accumulator**: a name that collects a running result.

```
bind start = do {
  bind total = 0
  bind n = 1
  while n <= 100 {
    bind total = total + n
    bind n = n + 1
  }
  println("1 + 2 + ... + 100 = ", total)

  bind balance = 100
  bind years = 0
  while balance < 200 {
    bind balance = balance * 1.07
    bind years = years + 1
  }
  println("Doubling at 7 percent takes ", years, " years")
}
```

```
OUTPUT
1 + 2 + ... + 100 = 5050
Doubling at 7 percent takes 11 years
```

The first loop adds the numbers 1 to 100 into `total`. Each pass through the loop, `total` is rebound to its old value plus the current `n`. (The story goes that the mathematician Carl Friedrich Gauss did this sum in his head as a schoolboy; a computer needs a hundredth of a

millisecond.) The second loop answers a real question: how many years does it take for savings to double at 7 percent interest? Notice that we did not need to know the answer in advance. The loop keeps going until the condition `balance < 200` is false.

When you do not know how many repetitions you will need, `while` is the right tool.

A loop with a decision inside it

Loops and decisions nest in each other like boxes. Here is a famous puzzle, the Collatz sequence. Start with any number. If it is even, halve it. If it is odd, triple it and add one. Repeat until you reach 1. Nobody has ever proved that every starting number gets there, but 27 does, and we can count how many steps it takes:

```
bind start = do {
  bind n = 27
  bind steps = 0
  while n != 1 {
    if n % 2 == 0 {
      bind n = n / 2
    } else {
      bind n = n * 3 + 1
    }
    bind steps = steps + 1
  }
  println("Collatz steps for 27: ", steps)
}
```

```
OUTPUT
Collatz steps for 27: 111
```

Because `if` and `while` share their surroundings, the `bind n = ...` inside the `if` really does change the `n` that the `while` tests. This is exactly the behaviour we want.

Counting with `for`

When you *do* know how many times to repeat, `for` is more compact. A `for` loop walks through a **range** of numbers:

```
bind start = do {
  for i in 0..5 {
    println("i is ", i)
  }
  for k in 1..4 {
    println(k, " x 7 = ", k * 7)
  }
}
```

```
OUTPUT
i is 0
i is 1
i is 2
i is 3
i is 4
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
```

`0..5` means "the numbers 0, 1, 2, 3 and 4". The range starts at the first number and stops *just before* the second. That is why `for i in 0..5` runs exactly 5 times, and why computer people often count from 0. The name `i` (or `k`, or anything you like) is bound to each number in turn.

A range can go through any whole numbers: `1..4` gives 1, 2 and 3.

You can also walk straight through a list, which Chapter 5 explores in detail:

```
bind start = do {
  bind xs = [10, 20, 30]
  for x in xs {
    println(x * 2)
  }
}
```

```
OUTPUT
20
40
60
```

Trap 1: the range end must be one simple thing

This is a trap, and a nasty one because Ling does not warn you. The end of a range must be a single token: a number, a name, or a parenthesised expression. It cannot be a function call or a sum written without parentheses.

```
FRAGMENT
bind xs = [10, 20, 30]
for i in 0..len(xs) {      // WRONG: silently runs zero times
  println("item ", i)
}
for i in 0..(len(xs)) {    // RIGHT: parentheses
  println("item ", i)
}
```

Here is a complete program that runs both versions side by side:

```
bind start = do {
  bind xs = [10, 20, 30]
  println("the wrong way:")
  for i in 0..len(xs) {
    println("item ", i)
  }
  println("the right way:")
  for i in 0..(len(xs)) {
    println("item ", i)
  }
  println("straight over the list:")
  for x in xs {
    println(x)
  }
}
```

```
OUTPUT
the wrong way:
the right way:
item 0
item 1
item 2
straight over the list:
10
20
30
```

The "wrong way" prints nothing at all. Ling read `0..len` as the range and then did not know what to do with `(xs)`. (The interpreter, run with `ling run --interp`, reports an `undefined: 'len'` error instead of staying quiet. We will come back to that difference in Chapter 9, because it is one of the best debugging tools you have.)

> **TIP**
>
> Make it a rule: whenever a range end is anything other than a plain number or a plain name, wrap it in parentheses. `0..(n + 1)`, `0..(len(xs))`.

Trap 2: a `for` loop cannot update outside names

Now the most surprising rule in Ling. Inside a `for` loop, each pass works on its own fresh copy of the surrounding names. When you `bind` something inside a `for`, that binding does not escape. Look at this attempt to add up the numbers 1 to 4:

```
bind start = do {
  bind total = 0
  for i in 1..5 {
    bind total = total + i
  }
  println("total after for: ", total)
}
```

```
OUTPUT
total after for: 0
```

The total should be 10. It is 0, because the `bind total = total + i` changed only a private copy that was thrown away at the end of each pass. This happens in both of Ling's engines, so you can rely on it.

The cure is simple: **if you want to accumulate a result, use `while`**. The loops earlier in this chapter already do it correctly. If you like the neatness of `for` for printing and drawing, use it for that. If you need to collect a total, a count, a maximum or a growing text, use `while`.

> NOTE
>
> Loops that only *do* things (print, draw, play a note) are perfectly fine as `for`. The trap only bites when the loop is supposed to *leave something behind* in a name.

Nested loops: drawing a triangle

A loop inside another loop is called a *nested* loop. The outer loop picks a row; the inner loop works along the row. Here the outer loop is a `for` (it only prints), and the inner one is a `while` (it builds up the text `line`):

```
bind start = do {
  for row in 1..6 {
    bind line = ""
    bind col = 0
    while col < row {
      bind line = line + "*"
      bind col = col + 1
    }
    println(line)
  }
}
```

```
OUTPUT
*
**
***
****
*****
```

Each row is one star longer than the one before. Notice how `bind line = ""` resets the text at the start of each row, and how `col` has to be reset too. Those resets are where nested loops usually go wrong.

No `break` and no `continue`

Some languages let you jump out of a loop early with `break`. Ling's `stop` and `again` are reserved words, but in the current version they do not do anything. To leave a loop early, make the condition false. A common way is a flag:

```
FRAGMENT
bind searching = true
bind i = 0
while searching && i < 100 {
  if i * i > 50 {
    bind searching = false
  } else {
    bind i = i + 1
  }
}
```

`match`: choosing between many values

A long `else if` chain that compares one value against many possibilities is clumsy. `match` is neater. You give it a value and a list of **arms**, each written `pattern => action`. Ling tries the arms from the top and runs the first one that fits. The pattern `_` fits anything, so it makes a good "everything else" arm.

```
bind start = do {
  bind day = 3
  match day {
    1 => println("Monday")
    2 => println("Tuesday")
    3 => println("Wednesday")
    _ => println("Some other day")
  }
  bind word = "red"
  match word {
    "red" => println("stop")
    "green" => println("go")
    _ => println("wait")
  }
}
```

```
OUTPUT
Wednesday
stop
```

You can match numbers and text. The arms are checked in order, so keep `_` last.

`match` can also produce a value, so you can bind its result:

```
bind start = do {
  bind label = match 2 {
    1 => "one"
    2 => "two"
    _ => "many"
  }
  println(label)
}
```

```
OUTPUT
two
```

> **WARNING**
>
> Ling does not check that you covered every case. If no arm matches and there is no `_`, the `match` quietly produces `()`, the "nothing" value. Always end with `_` unless you are certain.

In this chapter we only matched literal values. `match` can also take apart the results of operations that might fail (`ok(...)` and `bad(...)`) and your own custom types. You will meet that when we study more advanced data.

> **TRY IT**
>
> Change `day` to 9 and run `d11` again to see the `_` arm in action. Then delete the `_` line and run it a third time. What is printed?

Chapter summary

- `if condition { ... }` runs a block only when the condition is true; add `else { ... }` for the alternative and `else if` for chains. Braces are required; parentheses around the condition are not.
- Combine conditions with `&&`, `||` and `!`. Test evenness with `n % 2 == 0`.
- `if` and `while` blocks share the surrounding names, so binds inside them persist.
- `while condition { ... }` repeats while the condition holds. Always move something toward ending the loop. Use Ctrl+C to stop a runaway.
- Use `while` with an accumulator to compute totals, counts and other running results.
- `for i in a..b` counts from `a` up to, but not including, `b`. It also walks lists: `for x in xs`.
- Trap 1: a range end must be a single token. Write `0..(len(xs))`, not `0..len(xs)`.
- Trap 2: binds inside a `for` body do not escape the loop. Accumulate with `while`.
- There is no working `break` or `continue`; use a flag in the `while` condition.
- `match value { pattern => action _ => fallback }` chooses between many cases and may produce a value.

Exercises

1. Write a program that binds a number `n` and prints `positive`, `negative` or `zero`.
2. Print the multiplication table of 6 from 6 x 1 to 6 x 10, using a `for` loop.
3. Use a `while` loop to add up all the even numbers from 2 to 50 and print the total.

4. Starting with 1, double a number until it is larger than 1000. Print how many doublings it took and the final value.
5. (Challenge) Print a triangle of numbers where row 1 is `1`, row 2 is `12`, row 3 is `123`, up to row 5. Hint: build a text in an inner `while` loop with `to_str`.

Answers

1. `ling-module bind start = do { bind n = -4 if n > 0 { println("positive") } else if n < 0 { println("negative") } else { println("zero") } }`
2. `ling-module bind start = do { for k in 1..11 { println("6 x ", k, " = ", 6 * k) } }` The range ends at 11 because the end is excluded.
3. `ling-module bind start = do { bind total = 0 bind n = 2 while n <= 50 { bind total = total + n bind n = n + 2 } println(total) }` The total is 650.
4. `ling-module bind start = do { bind v = 1 bind count = 0 while v <= 1000 { bind v = v * 2 bind count = count + 1 } println(count, " doublings, final value ", v) }` This prints `10 doublings, final value 1024`.
5. `ling-module bind start = do { for row in 1..6 { bind line = "" bind c = 1 while c <= row { bind line = line + to_str(c) bind c = c + 1 } println(line) } }`

CHAPTER 4

Functions

What you'll learn. A function is a named recipe: you write the steps once, and use them as often as you like. In this chapter you will define your own functions with `fn`, give them parameters, get answers back with `return`, and discover how a function can even call itself. You will finish by building small, complete programs from well-chosen pieces.

The problem functions solve

Suppose you want to greet three people. You could write three `println` lines with slightly different text. For three people that is fine; for three hundred it is a chore, and if you later decide the greeting should end with a smiley, you must change three hundred lines.

A **function** solves this. It is a named, reusable block of instructions. You have been *using* functions since Chapter 1: `println`, `sqrt` and `round` are all functions that someone wrote for you. Now you will write your own.

Defining and calling a function

```
fn greet(name) {
  println("Hello, " + name + "!")
}

bind start = do {
  greet("Ada")
  greet("Grace")
  greet("Mina")
}
```

```
OUTPUT
Hello, Ada!
Hello, Grace!
Hello, Mina!
```

The pattern is:

```
FRAGMENT
fn name(parameter) {
  ... instructions ...
}
```

- `fn` is the keyword that says "I am defining a function".
- `greet` is the function's **name**. Pick a verb or a clear description.
- `name` in parentheses is a **parameter**: a blank that is filled in each time the function is used. Inside the function, it behaves like a name bound to whatever value you passed in.
- The braces hold the **body**, the instructions the function carries out.

Defining a function does not run it. Just as a recipe in a cookbook produces no cake until someone cooks, the body only runs when you **call** the function: `greet("Ada")`. The value in parentheses (the **argument**) is poured into the parameter.

Note where the function sits: *outside* and above `bind start`. In Ling, the top level of a file may contain only `bind`, `fn`, and a few other definitions. The instructions that do things live inside `start`.

> TIP
>
> Think of a function as a vending machine. You put in the arguments, something happens inside, and (sometimes) something comes out. You do not need to know how the machine works to use it; you only need to know what to put in and what you get back.

More than one parameter

A function can take several parameters. Separate them with commas in the definition and in the call, and the arguments are matched up by position.

Getting answers back with `return`

So far `greet` only *prints*. Often you want a function to *compute* something and hand the answer back, so that the caller can use it in further calculations. That is what `return` does:

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

fn average(a, b) {
  return (a + b) / 2
}

bind start = do {
  println(square(7))
  println(average(10, 25))
  println(square(average(2, 4)))
}
```

```
OUTPUT
49
17.5
9
```

`return n * n` ends the function immediately and gives `n * n` back to the caller. Because the call `square(7)` *has a value*, you can use it anywhere a value is allowed: inside `println`, inside a bigger expression, or even as the argument of another call. The last line, `square(average(2, 4))`, is worked out from the inside out: `average(2, 4)` is 3, and `square(3)` is 9.

Early returns

A function can have several `return` statements. The first one that runs ends the function, so you can use them to handle special cases up front, which keeps the rest of the function simple:

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

fn describe(n) {
  if n < 0 {
    return "negative"
  }
  if n == 0 {
    return "zero"
  }
  return "positive"
}

bind start = do {
  println(cube(3))
  println(describe(-5))
  println(describe(0))
  println(describe(12))
}
```

```
OUTPUT
27
negative
zero
positive
```

Look at `describe`. If `n` is negative, the first `return` fires and the function is finished; the other tests are never even looked at. There is no need for `else` here because a `return` already leaves the function. Also notice `cube`: when a function body is just one expression, you may leave out `return`, and the value of the last expression is returned automatically. We recommend writing `return` anyway; it makes your intention obvious.

> **NOTE**
>
> A function that never returns a value gives back `()`, the "nothing" value. If you print the result of `greet("Ada")` you will see `()`. This is why you should not use the result of a function that only prints.

Types are optional

If you have seen other languages, you may wonder whether Ling lets you declare what kinds of values a function accepts. You may write `fn double(x: number) -> number { ... }`, and Ling accepts it, but it currently ignores these annotations entirely. They serve only as notes for human readers. We leave them out in this book.

Functions that call functions

Real programs are built in layers. Small functions do one tiny job; bigger functions combine them. Here is a temperature converter:

```
fn celsius_to_f(c) {
  return c * 9 / 5 + 32
}

fn fahrenheit_to_c(f) {
  return (f - 32) * 5 / 9
}

bind start = do {
  println("0 C   = ", celsius_to_f(0), " F")
  println("37 C  = ", celsius_to_f(37), " F")
  println("100 C = ", celsius_to_f(100), " F")
  println("212 F = ", fahrenheit_to_c(212), " C")
  println("")
  println("Celsius  Fahrenheit")
  bind c = -10
  while c <= 40 {
    println(c, "        ", celsius_to_f(c))
    bind c = c + 10
  }
}
```

```
OUTPUT
0 C   = 32 F
37 C  = 98.6 F
100 C = 212 F
212 F = 100 C

Celsius  Fahrenheit
-10        14
0         32
10         50
20         68
30         86
40         104
```

The two conversion functions each hold one formula. `start` uses them as a toolbox. Notice how readable `start` is: it says *what* happens (convert 100 degrees) without showing *how*. The formula lives in exactly one place, so if you find a mistake, you fix it once.

The table at the bottom uses a `while` loop (remember: loops that need to change a counter use `while`) and calls `celsius_to_f` on every pass.

> **TRY IT**
>
> Add a function `km_to_miles(km)` (multiply by 0.621371) and print a table for 10, 20, …, 100 kilometres.

What a function can see

Each time a function runs, it gets its own private set of names: its parameters, and anything it binds itself. They disappear when the function ends, and they cannot collide with names in the code that called it. The function below counts using its own `i`, but the caller's `i` is untouched:

```
fn count_up(limit) {
  bind i = 1
  while i <= limit {
    println(i)
    bind i = i + 1
  }
  println("counted to ", limit)
}

bind start = do {
  bind i = 99
  count_up(3)
  println("my own i is still ", i)
}
```

```
OUTPUT
1
2
3
counted to 3
my own i is still 99
```

A function can also see names bound at the **top level** of the file, outside any function. That is the right place for constants (values that never change) such as a tax rate:

```
bind tax_rate = 0.2

fn with_tax(price) {
  return price + price * tax_rate
}

bind start = do {
  println(with_tax(50))
  println(with_tax(9.99))
}
```

```
OUTPUT
60
11.988
```

> **WARNING**
>
> A function cannot see the *local* names of the function that called it. If `start` binds `radius = 5` inside its braces and a function you wrote uses `radius`, the function will not find it. Pass it in as an argument instead. Passing values in explicitly makes functions easier to understand and reuse.

Functions that call themselves

Here is a mind-bending idea: a function may call *itself*. This is called **recursion**, and it is surprisingly natural. The factorial of a number `n` (written `n!`) is `n` times all the whole numbers below it down to 1. So `5! = 5 * 4 * 3 * 2 * 1 = 120`. Notice that `5! = 5 * 4!`. The big problem contains a smaller version of itself.

Every recursive function needs two parts:

1. A **base case**: a situation so simple that the answer is known directly (`1! = 1`). Without it the function would call itself forever.
2. A **recursive case**: it solves a slightly smaller problem by calling itself, and combines the result.

```
fn factorial(n) {
  if n <= 1 {
    return 1
  }
  return n * factorial(n - 1)
}

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

bind start = do {
  println("5! = ", factorial(5))
  println("10! = ", factorial(10))
  bind i = 0
  while i < 12 {
    println("fib(", i, ") = ", fib(i))
    bind i = i + 1
  }
}
```

```
OUTPUT
5! = 120
10! = 3628800
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
fib(11) = 89
```

Follow `factorial(3)` by hand:

- `factorial(3)` returns `3 * factorial(2)`.
- `factorial(2)` returns `2 * factorial(1)`.
- `factorial(1)` is the base case and returns 1.
- Now the answers climb back up: `2 * 1 = 2`, then `3 * 2 = 6`.

The second function, `fib`, computes the **Fibonacci numbers**: 0, 1, 1, 2, 3, 5, 8, 13, …, where each number is the sum of the two before it. It has two base cases (0 and 1) and calls itself twice. Fibonacci numbers appear in the spirals of shells and sunflowers, which is a good reason to remember them.

> **WARNING**
>
> Recursion has costs. A function that calls itself forever runs out of room and crashes. And the doubled call in `fib` is slow for big inputs: `fib(35)` does tens of millions of calls. Use recursion where the problem is naturally nested, and loops where a plain counting job will do.

Organizing a program

Once you know functions, a good habit is to make `start` read like a table of contents. Put each job in a function with a clear name, and let `start` do little more than call them in order.

A good function:

- does **one thing**, and its name says what,
- takes everything it needs as **parameters**,
- either **returns** a value or **does** something (prints, draws), not both,
- fits on one screen.

Compare two versions of "print the area of a circle". In the first, the formula is buried in `start`. In the second, there is a function `circle_area(r)` that returns a number, and `start` prints it. The second can be reused by a game that needs the number but never wants to print it.

Also remember to order your file sensibly: constants first, then helper functions from the simplest to the most complex, and `bind start` last.

Mini-project: FizzBuzz

FizzBuzz is the best-known beginner puzzle. Count from 1 to 20. For multiples of 3 print `Fizz`, for multiples of 5 print `Buzz`, for multiples of both print `FizzBuzz`, and for everything else print the number. (It is also a famous job-interview question, so you are about to learn something with practical value.)

```
fn fizzbuzz(n) {
  if n % 15 == 0 {
    return "FizzBuzz"
  }
  if n % 3 == 0 {
    return "Fizz"
  }
  if n % 5 == 0 {
    return "Buzz"
  }
  return to_str(n)
}

bind start = do {
  for n in 1..21 {
    println(fizzbuzz(n))
  }
}
```

```
OUTPUT
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
```

The order of the tests matters. A multiple of 15 is also a multiple of 3 and 5, so it must be checked *first*, otherwise it would print just `Fizz`. The function turns a number into a word, and `start` simply loops and prints. This separation means you could reuse `fizzbuzz` anywhere.

Mini-project: a number-guessing helper

In the game "guess my number", one player thinks of a number between 1 and 100, and the other guesses. After each guess, the first player says "too low", "too high" or "correct". The smartest strategy is to always guess the middle of the remaining range, so that each answer halves the possibilities. A hundred numbers needs at most seven guesses. Let us teach the computer this strategy:

```
# A "guessing helper": it finds a secret number from 1 to 100
# by always guessing the middle of the remaining range.

fn clue(guess, secret) {
  if guess < secret {
    return "too low"
  }
  if guess > secret {
    return "too high"
  }
  return "correct"
}

fn find_secret(secret) {
  bind low = 1
  bind high = 100
  bind tries = 0
  while true {
    bind guess = floor((low + high) / 2)
    bind tries = tries + 1
    bind answer = clue(guess, secret)
    println("guess ", guess, ": ", answer)
    if answer == "correct" {
      return tries
    }
    if answer == "too low" {
      bind low = guess + 1
    } else {
      bind high = guess - 1
    }
  }
}

bind start = do {
  bind n = find_secret(37)
  println("Found it in ", n, " guesses")
}
```

```
OUTPUT
guess 50: too high
guess 25: too low
guess 37: correct
Found it in 3 guesses
```

The function `clue` is the referee. It compares a guess with the secret and returns one of three texts. `find_secret` is the player: it keeps a `low` and a `high` end of the range still possible, guesses the middle (`floor` rounds down), asks the referee, and narrows the range. Note `while true`: this loop has no natural end, so we leave it with `return` when the answer is correct. That is one safe way to write a loop with an exit in the middle.

> **TRY IT**
>
> Change the secret from 37 to 1 and to 100, and see how many guesses are needed. Then try other values. You should never need more than seven.

Notice how this program has three layers: `clue` knows the rules, `find_secret` knows the strategy, `start` knows the story. You can change any layer without touching the others.

Silent mistakes with function names

One last warning, which is so important that it gets its own section. If you misspell a function name, the default engine does **not** complain:

```
FRAGMENT
bind start = do {
  println(double(21))
  println(doubel(21))    // typo
}
```

On the default engine, the second line prints `()` and carries on as if nothing were wrong. The interpreter, however, stops with `error[runtime]: unknown function 'doubel'`. If a function seems to "do nothing", or if `()` shows up in your output, run the program with `ling run --interp yourfile.ling`. We will study this properly in Chapter 9.

Chapter summary

- `fn name(parameters) { body }` defines a function; define it at the top level of the file, outside `start`.
- A function runs only when it is **called**: `name(arguments)`.
- `return value` hands a value back and ends the function; a function can have several returns.
- Without `return`, the value of the last expression is returned; a function that does not compute anything returns `()`.
- Parameters and names bound inside a function are private to it. Functions can see top-level bindings but not the local names of their caller.
- A function may call itself (recursion) if it has a base case that stops the process.
- Build programs in layers of small functions with clear names; keep `start` short.
- Type annotations are accepted but ignored.
- Misspelled function names fail silently on the default engine; use `ling run --interp` to catch them.

Exercises

1. Write a function `is_even(n)` that returns `true` or `false`, and print `is_even(10)` and `is_even(7)`.
2. Write `max3(a, b, c)` that returns the largest of three numbers. (Hint: the built-in `max` takes two arguments, and you can nest it.)
3. Write `sum_to(n)` that returns `1 + 2 + ... + n` using a `while` loop, and then a second version `sum_to_r(n)` that uses recursion. Check both give 5050 for 100.
4. Write `power(base, exp)` recursively, using only multiplication. (`power(2, 10)` should be 1024.)
5. Write `grade(score)` that returns `"A"` for 90 and over, `"B"` for 80 to 89, `"C"` for 70 to 79 and `"F"` otherwise. Then use it to print the grades for the scores 95, 85, 75 and 40.

Answers

1. `ling-module fn is_even(n) { return n % 2 == 0 } bind start = do { println(is_even(10)) println(is_even(7)) }` This prints `true` and `false`.
2. `fn max3(a, b, c) { return max(a, max(b, c)) }`. Calling `max3(4, 9, 6)` gives 9.

3. `ling-module fn sum_to(n) { bind total = 0 bind i = 1 while i <= n { bind total = total + i bind i = i + 1 } return total } fn sum_to_r(n) { if n <= 0 { return 0 } return n + sum_to_r(n - 1) }` Both return 5050 for 100.
4. `ling-module fn power(base, exp) { if exp == 0 { return 1 } return base * power(base, exp - 1) }` The base case is `exp == 0`, since anything to the power 0 is 1.
5. `ling-module fn grade(score) { if score >= 90 { return "A" } if score >= 80 { return "B" } if score >= 70 { return "C" } return "F" } bind start = do { for s in [95, 85, 75, 40] { println(s, ": ", grade(s)) } }` This prints `95: A`, `85: B`, `75: C` and `40: F`.

CHAPTER 5

Lists and Text

What you'll learn. Until now each name held one value. In this chapter you will learn to keep many values together in a **list**, to walk through them with loops, and to take text apart and put it back together again. You will finish with three small projects: a word counter, a shopping list and a statistics tool. You will also learn what to do about the fact that Ling has no dictionary type yet.

Lists: a row of boxes

Imagine a row of numbered boxes. Each box holds one value, and you find a value by its box number. That is a **list**. Lists are how a program holds a collection: the scores of five players, the names in a class, the pixels of a row in a picture.

You write a list between square brackets, with the values separated by commas:

```
bind start = do {
  bind fruits = ["apple", "banana", "cherry"]
  println(fruits)
  println(len(fruits))
  println(list_get(fruits, 0))
  println(list_get(fruits, 2))
  bind fruits = list_push(fruits, "date")
  println(fruits)
  println("now ", len(fruits), " fruits")
  bind fruits = list_set(fruits, 1, "blueberry")
  println(fruits)
  println(list_join(fruits, ", "))
}
```

```
OUTPUT
[apple, banana, cherry]
3
apple
cherry
[apple, banana, cherry, date]
now 4 fruits
[apple, blueberry, cherry, date]
apple, blueberry, cherry, date
```

Let us unpack this:

- `["apple", "banana", "cherry"]` is a list of three pieces of text. Printing a list shows its contents in square brackets (text without its quotes).
- `len(fruits)` is the number of items: `3`. The same function counts the characters of a text.
- `list_get(fruits, 0)` fetches the item at position 0. **Positions start at 0**, not 1. The first item is number 0; the last of three is number 2. This is called *zero-based indexing* and is common in programming. If it feels strange, think of the position as "how many items do I skip before this one?"
- `list_push(fruits, "date")` produces a list with an extra item at the end.
- `list_set(fruits, 1, "blueberry")` produces a list with the item at position 1 replaced.
- `list_join(fruits, ", ")` glues the items into one text, using the separator you give.

Notice that after `list_push` and `list_set` we wrote `bind fruits = ...` again. That brings us to the most important rule of this chapter.

> WARNING
>
> **Always rebind the result of `list_push` and `list_set`.** Write `bind fruits = list_push(fruits, "date")`, never just `list_push(fruits, "date")`. On one of Ling's two engines the original list is also changed behind your back, and on the other it is not. Code that relies on either behaviour is a bug waiting to happen. By always rebinding, you get the same result everywhere.

Why rebinding matters: a demonstration

The following tiny program shows what happens if you rely on the original staying unchanged:

```
bind start = do {
  bind a = [1, 2, 3]
  bind b = list_set(a, 0, 99)
  println("a = ", a)
  println("b = ", b)
}
```

```
OUTPUT
a = [99, 2, 3]
b = [99, 2, 3]
```

With the default engine, `a` was changed by `list_set` as a side effect, so both lists show `99`. With the interpreter (`ling run --interp`), `a` keeps its original `[1, 2, 3]`. Same program, different answers. The lesson: **a list that you want to keep unchanged must not be passed to `list_set` or `list_push`.** If you need an independent copy, build one with a loop. You will see this done in the statistics project below.

Building a list in a loop

Start with an empty list from `list_new()`, then push items into it. Because of the rule from Chapter 3 that `for` loops cannot leave results behind, build lists with `while`:

```
bind start = do {
  bind squares = list_new()
  bind i = 1
  while i < 8 {
    bind squares = list_push(squares, i * i)
    bind i = i + 1
  }
  println(squares)
  println(len(squares))
}
```

```
OUTPUT
[1, 4, 9, 16, 25, 36, 49]
7
```

Here is what happens if you use `for` instead:

```
bind start = do {
  bind squares = list_new()
  for i in 1..8 {
    bind squares = list_push(squares, i * i)
  }
  println(squares)
}
```

OUTPUT

```
[1, 4, 9, 16, 25, 36, 49]
```

On the default engine this seems to work. On the interpreter, it prints `[]`: an empty list, because each pass of the `for` loop pushed onto a private copy that was thrown away. The default engine only gets the "right" answer by accident, thanks to the aliasing we saw above. Do not rely on it. **Build lists with `while`.**

NOTE

For reading, `for` is fine: `for x in xs { ... }` walks through a list. The trouble only starts when the loop body must leave something behind.

Out-of-range positions

If you ask for a position that does not exist, `list_get` does not stop the program. It quietly returns an empty text. A list of 3 items has positions 0, 1 and 2, and `list_get(fruits, 3)` is an empty text, not an error. Keep your counters within `0` and `len(xs) - 1`, and in a loop test `i < len(xs)`.

Walking a list

The usual pattern for visiting every item with an index is a `while` loop:

FRAGMENT

```
bind i = 0
while i < len(xs) {
  bind item = list_get(xs, i)
  # ... do something with item ...
  bind i = i + 1
}
```

If you only need the items and not their positions, `for item in xs { ... }` is shorter. If you need the position too, you can combine `for i in 0..(len(xs))` (with its mandatory parentheses!) with `list_get(xs, i)`.

Finding the biggest and smallest

Statistics need totals, extremes and averages. Here a loop computes them by hand, and the built-in `list_max` and `list_min` confirm the result (the second argument is a fallback for an empty list):

```
bind start = do {
  bind scores = [72, 95, 88, 61, 79]
  bind total = 0
  bind best = list_get(scores, 0)
  bind worst = list_get(scores, 0)
  bind i = 0
  while i < len(scores) {
    bind s = list_get(scores, i)
    bind total = total + s
    if s > best {
      bind best = s
    }
    if s < worst {
      bind worst = s
    }
    bind i = i + 1
  }
  println("count:   ", len(scores))
  println("total:   ", total)
  println("average: ", total / len(scores))
  println("best:    ", best)
  println("worst:   ", worst)
  println("list_max says ", list_max(scores, 0))
  println("list_min says ", list_min(scores, 0))
}
```

```
OUTPUT
count:   5
total:   395
average: 79
best:    95
worst:   61
list_max says 95
list_min says 61
```

The pattern `best` / `worst` is the "champion" technique: start with the first item as the champion, then challenge it with each item and replace it whenever the challenger wins.

Text is a sequence too

Text looks like a list of characters, and a few of the same ideas apply. But Ling gives text its own toolbox. Here is the tour:

```
bind start = do {
  bind s = "  Hello, Ling World  "
  println(len(s))
  println("[" + trim(s) + "]")
  bind s = trim(s)
  println(str_upper(s))
  println(str_lower(s))
  println(substr(s, 0, 5))
  println(str_find(s, "Ling"))
  println(str_find(s, "Python"))
  println(str_replace(s, "World", "Friend"))
  println(starts_with(s, "Hello"))
  println(ends_with(s, "World"))
  println(str_repeat("ab", 3))
  println(s.contains("Ling"))
}
```

```
OUTPUT
21
[Hello, Ling World]
HELLO, LING WORLD
hello, ling world
Hello
7
-1
Hello, Ling Friend
true
true
ababab
true
```

Function	What it does
`len(s)`	number of characters
`trim(s)`	remove spaces from both ends
`str_upper(s)` / `str_lower(s)`	change case
`substr(s, start, count)`	`count` characters beginning at position `start` (starting from 0, like lists)
`str_find(s, needle)`	position of the first match, or `-1` if there is none
`str_replace(s, from, to)`	replace every `from` with `to`
`starts_with(s, prefix)`, `ends_with(s, suffix)`	yes/no tests
`str_repeat(s, n)`	repeat the text n times
`s.contains(needle)`	does it contain the needle?

Notice `-1` from `str_find`: since valid positions are 0 or more, `-1` is the traditional "not found" signal. Test for it with an `if`.

> **TIP**
>
> Functions that change text, such as `str_upper`, never modify the original text; they return a new one. As with lists, you must bind the result to keep it: `bind s = trim(s)`.

Splitting and joining

The most valuable text function is `split`. It cuts a text into a list of pieces wherever a separator appears. Its partner `list_join` glues them back together:

```
bind start = do {
  bind line = "red,green,blue,yellow"
  bind parts = split(line, ",")
  println(parts)
  println(len(parts))
  for p in parts {
    println("colour: ", p)
  }
  println(list_join(parts, " | "))
}
```

```
OUTPUT
[red, green, blue, yellow]
4
colour: red
colour: green
colour: blue
colour: yellow
red | green | blue | yellow
```

Whenever you have data in a text (a line of comma-separated values, a sentence, a path with slashes) `split` is the first tool to reach for.

Taking text apart letter by letter

Here is a classic exercise: reversing text and testing for palindromes (words that read the same backwards, like "level"). We use `substr(s, i, 1)`, which fetches the single character at position `i`:

```
fn reverse_text(s) {
  bind out = ""
  bind i = len(s) - 1
  while i >= 0 {
    bind out = out + substr(s, i, 1)
    bind i = i - 1
  }
  return out
}

fn is_palindrome(s) {
  bind clean = str_lower(str_replace(s, " ", ""))
  return clean == reverse_text(clean)
}

bind start = do {
  println(reverse_text("Ling"))
  println(is_palindrome("never odd or even"))
  println(is_palindrome("hello"))
}
```

```
OUTPUT
gniL
true
false
```

Study `is_palindrome`: it first cleans the text (removes spaces, lowers the case), then compares it with its own reverse. Small functions, each doing one job, working together. This is the style that makes programs readable.

TRY IT

Make `is_palindrome` ignore commas as well, by adding a second `str_replace`. Test it with "A man, a plan, a canal, Panama".

There is no dictionary (yet)

In many languages you can make a *dictionary* (also called a *map* or *hash table*) that stores values under names: `ages["Mina"] = 12`. **Ling does not have such a type yet.** Neither does it have sets or tuples. What it has instead are lists and, as we will see in later editions, your own custom record types.

This is an inconvenience, not a dead end. The standard workaround is **parallel lists**: keep one list of keys and a second list of values, so that `keys[i]` goes with `values[i]`. To look something up, search the keys list for the position, then read the same position in the values list. The helper function that finds the position is the heart of the pattern:

FRAGMENT

```
fn index_of(items, target) {
  bind i = 0
  while i < len(items) {
    if list_get(items, i) == target {
      return i
    }
    bind i = i + 1
  }
  return -1
}
```

A search through a list is slower than a real dictionary when the list is huge. For the hundreds or few thousands of items in a beginner's project it is instantaneous.

NOTE

You can compare numbers, text and booleans with `==`, as we have been doing. But comparing two *lists* with `==` is always false, even if their contents are identical. Compare the items one by one, or compare their `list_join` texts.

Mini-project: word counter

How many times does each word appear in a sentence? This is a dictionary job, so we use the parallel list trick: `words` holds each distinct word, and `counts` holds how often we have seen it.

```
# Word counter. Ling has no dictionary type (yet), so we keep two lists
# side by side: words[i] is a word and counts[i] is how often we saw it.

fn index_of(items, target) {
  bind i = 0
  while i < len(items) {
    if list_get(items, i) == target {
      return i
    }
    bind i = i + 1
  }
  return -1
}

bind start = do {
  bind text = "the cat saw the dog and the dog saw the cat run"
  bind all = split(text, " ")
  bind words = list_new()
  bind counts = list_new()

  bind i = 0
  while i < len(all) {
    bind w = list_get(all, i)
    bind pos = index_of(words, w)
    if pos == -1 {
      bind words = list_push(words, w)
      bind counts = list_push(counts, 1)
    } else {
      bind counts = list_set(counts, pos, list_get(counts, pos) + 1)
    }
    bind i = i + 1
  }

  println("total words:  ", len(all))
  println("unique words: ", len(words))
  bind k = 0
  while k < len(words) {
    println(list_get(words, k), ": ", list_get(counts, k))
```

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

```
OUTPUT
total words:  12
unique words: 6
the: 4
cat: 2
saw: 2
dog: 2
and: 1
run: 1
```

Trace what happens for the first three words. `the`: `index_of` finds nothing (`-1`), so we append `the` to `words` and `1` to `counts`. `cat`: same. `saw`: same. When `the` appears again, `index_of` returns `0`; we rebind `counts` with position 0 raised by one. Notice how both `list_push` and `list_set` are used as `bind counts = ...`.

You can improve this: make the text lower-case with `str_lower` first so that "The" and "the" count as the same word, or remove punctuation with `str_replace`.

Mini-project: shopping list

Three parallel lists hold the item names, unit prices and quantities. A loop prints a bill:

```
# A shopping list with prices, kept as two lists in step.
bind start = do {
  bind items = ["milk", "bread", "apples", "cheese"]
  bind prices = [1.20, 2.50, 3.75, 4.10]
  bind qty = [2, 1, 6, 1]

  println("Item      Qty   Cost")
  bind total = 0
  bind i = 0
  while i < len(items) {
    bind cost = list_get(prices, i) * list_get(qty, i)
    println(list_get(items, i), "   ", list_get(qty, i), "   ", cost)
    bind total = total + cost
    bind i = i + 1
  }
  println("Total: ", round(total * 100) / 100)

  bind items = list_push(items, "tea")
  bind prices = list_push(prices, 3.20)
  bind qty = list_push(qty, 1)
  println("Now buying ", len(items), " things: ", list_join(items, ", "))
}
```

```
OUTPUT
Item      Qty   Cost
milk   2   2.4
bread   1   2.5
apples   6   22.5
cheese   1   4.1
Total: 31.5
Now buying 5 things: milk, bread, apples, cheese, tea
```

The list `items` gets a new entry along with one in each of the other two lists. That is the price of parallel lists: you must keep them in step. If you add an item to one but forget the others, the numbers will be out of line.

Mini-project: simple statistics

Our final project computes the mean (average), the median (the middle value when sorted) and the range of a data set. It includes a small sorting routine and a function that makes an independent copy of a list, so that the caller's data is never touched:

```
# Simple statistics with a hand-written sort.
fn copy_of(xs) {
  bind out = list_new()
  bind i = 0
  while i < len(xs) {
    bind out = list_push(out, list_get(xs, i))
    bind i = i + 1
  }
  return out
}

fn sorted(xs) {
  bind out = copy_of(xs)
  bind n = len(out)
  bind i = 0
  while i < n {
    bind j = i + 1
    while j < n {
      if list_get(out, j) < list_get(out, i) {
        bind tmp = list_get(out, i)
        bind out = list_set(out, i, list_get(out, j))
        bind out = list_set(out, j, tmp)
      }
      bind j = j + 1
    }
    bind i = i + 1
  }
  return out
}

fn mean(xs) {
  bind total = 0
  bind i = 0
  while i < len(xs) {
    bind total = total + list_get(xs, i)
    bind i = i + 1
  }
  return total / len(xs)
```

```
}

fn median(xs) {
  bind s = sorted(xs)
  bind n = len(s)
  bind mid = floor(n / 2)
  if n % 2 == 1 {
    return list_get(s, mid)
  }
  return (list_get(s, mid - 1) + list_get(s, mid)) / 2
}

bind start = do {
  bind data = [9, 2, 7, 4, 4, 10, 1]
  println("data:   ", data)
  println("sorted: ", sorted(data))
  println("mean:   ", round(mean(data) * 100) / 100)
  println("median: ", median(data))
  println("data again: ", data)
  println("range:  ", list_min(data, 0), " to ", list_max(data, 0))
}
```

```
OUTPUT
data:   [9, 2, 7, 4, 4, 10, 1]
sorted: [1, 2, 4, 4, 7, 9, 10]
mean:   5.29
median: 4
data again: [9, 2, 7, 4, 4, 10, 1]
range:  1 to 10
```

The sort works like this: for each position `i`, look at every later position `j`; whenever the later value is smaller, swap the two. After the loop, the smallest values have bubbled to the front. The swap uses a temporary name `tmp`, exactly like swapping the contents of two cups requires a third cup. This is called a *selection sort*; it is not the fastest sort but it is clear.

Because `sorted` works on a copy (`copy_of`), the line `data again:` proves the original is unchanged on both engines.

Chapter summary

- A list holds many values: `[1, 2, 3]`. Create an empty one with `list_new()`.
- Positions start at 0. `list_get(xs, i)` reads, `list_set(xs, i, v)` replaces, `list_push(xs, v)` appends, `len(xs)` counts.
- **Always rebind the result:** `bind xs = list_push(xs, v)`. Never rely on the original list staying the same or changing.
- Build and update lists with `while`, not `for`. Reading with `for x in xs` is fine.
- `list_get` with a bad position returns empty text rather than an error; stay within `0` and `len(xs) - 1`.
- `list_join(xs, sep)` makes text from a list; `split(text, sep)` makes a list from text.
- Text tools: `trim`, `str_upper`, `str_lower`, `substr`, `str_find` (-1 if absent), `str_replace`, `starts_with`, `ends_with`, `str_repeat`, `contains`.
- Ling has no dictionary type yet. Use parallel lists and an `index_of` helper.
- `==` on lists is always false; compare item by item.

Exercises

1. Make a list of five numbers and print their total with a `while` loop.
2. Write `count_vowels(s)` that returns how many of the letters a, e, i, o, u appear in a text. (Hint: lower-case it first, then test each character with `str_find("aeiou", ch) != -1`.)
3. Split the text `"3,8,15,4"` on commas, convert each piece with `to_number`, and print the biggest number.
4. Write a function `contains_item(xs, target)` that returns `true` or `false`. Use it to check whether `"cheese"` is on a shopping list.
5. Extend the word counter so that it prints the word with the highest count. (Hint: use the champion technique over the `counts` list and then read the same position from `words`.)

Answers

1. `ling-module bind start = do { bind xs = [4, 8, 15, 16, 23] bind total = 0 bind i = 0 while i < len(xs) { bind total = total + list_get(xs, i) bind i = i + 1 } println(total) }` This prints `66`.
2. `ling-module fn count_vowels(s) { bind t = str_lower(s) bind n = 0 bind i = 0 while i < len(t) { if str_find("aeiou", substr(t, i, 1)) != -1 { bind n = n + 1 } bind i = i + 1 } return n }` `count_vowels("Programming")` is 3.

3. `ling-module bind start = do { bind parts = split("3,8,15,4", ",") bind best = to_number(list_get(parts, 0)) bind i = 1 while i < len(parts) { if to_number(list_get(parts, i)) > best { bind best = to_number(list_get(parts, i)) } bind i = i + 1 } println(best) }` This prints `15`. (A `for` loop would not work here, because of the rule that binds inside `for` do not escape.)
4. Reuse the `index_of` pattern: `fn contains_item(xs, target) { return index_of(xs, target) != -1 }`, with `index_of` as defined above.
5. After the counting loop, add a "champion" loop: keep `best_pos = 0`, and for every `k` where `list_get(counts, k) > list_get(counts, best_pos)`, rebind `best_pos = k`. Finally print `list_get(words, best_pos)`. For the sentence in this chapter it prints `the`.

CHAPTER 6

Your First 3D Room

What you'll learn. It is time to leave the terminal. In this chapter you will open a window, learn the "frame loop" that powers every Ling picture, place 3D shapes with a camera and lights, draw spinning patterns, and finish with a room you can walk around using the arrow keys. You will also learn how to take screenshots of your work.

> NOTE
>
> Programs in this chapter open a window, so they are labelled **GUI** and have no output panel (the interesting output is on your screen). Press **Escape** or close the window to stop any of them.

How a picture moves

A film is a stack of still pictures shown fast enough that your eye sees movement. A Ling window works the same way. Your program draws one still picture (a **frame**), shows it, and immediately starts on the next. Done sixty times a second, this looks like animation.

Every graphical Ling program therefore has the same four-step rhythm:

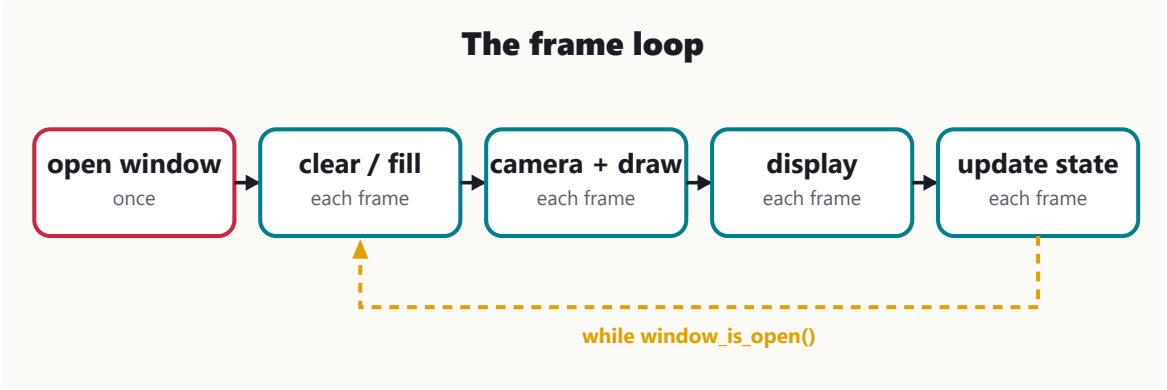


Figure: Open the window once. Then, for each frame: clear the screen, draw your scene, show it with present(), and update your numbers. Repeat while the window is open.

1. **Open** the window, once, before anything else.
2. **Clear** (`fill`) so that last frame's picture does not linger.

3. **Draw** the shapes.
4. **Present** (`present()`) to show what you drew.

The loop continues `while window_is_open()`. That function becomes `false` when the person closes the window or presses Escape, so you never need to code your own exit.

Your first window

Here is the smallest program that makes something spin:

```
GUI
bind start = do {
  open_window(640, 400, "My first window")
  bind angle = 0
  while window_is_open() {
    fill(15, 18, 40)
    set_color(255, 120, 60)
    cube(0, 0, 3, 1, 1, 1, angle, angle * 0.7, 0, 0)
    present()
    bind angle = angle + 0.02
  }
}
```

Line by line:

- `open_window(640, 400, "My first window")` makes a window 640 pixels wide and 400 pixels tall, with a title. Note the order: **numbers first, title third**.
- `bind angle = 0` creates a counter outside the loop. It stays alive for as long as the program runs.
- `while window_is_open() { ... }` is the frame loop.
- `fill(15, 18, 40)` paints the whole window dark blue. The three numbers are red, green and blue, each from 0 to 255.
- `set_color(255, 120, 60)` chooses the pen colour for the shapes that follow.
- `cube(0, 0, 3, 1, 1, 1, angle, angle * 0.7, 0, 0)` draws a cube (explained below).
- `present()` shows the picture.
- `bind angle = angle + 0.02` nudges the angle for the next frame. That is why the cube turns.

> WARNING
>
> Nothing appears on the screen until `present()` is called. If you forget it, you will see an empty window. And if you forget the `bind angle = angle + ...` line, your picture will be perfectly still. Both are the most common first mistakes.

Reading a shape call

Every solid shape in Ling, whether `cube`, `sphere`, `cone`, `torus` or a dozen others, is called the same way. There is a lot of information, so it helps to read it in groups of three:

FRAGMENT

```
cube(cx, cy, cz,   sx, sy, sz,   rx, ry, rz,   mode)
```

- `cx, cy, cz` is the **centre**: where the shape sits in the world.
- `sx, sy, sz` is the **size**: how stretched it is along each direction. A size of 1 gives roughly one unit of radius.
- `rx, ry, rz` is the **rotation**, in radians, around the x, y and z axes.
- `mode` is `0` for a solid shape, `1` for a wireframe (only the edges) and `2` for both.

Only the centre is truly required; the rest have sensible defaults. That is why you will see short calls like `sphere(0, 0, 4)` in other people's code.

Where is everything? The three axes

You need a mental map of space. In Ling's 3D world:

- **x** runs to the right,
- **y** runs **down**,
- **z** runs **away from you**, into the screen.

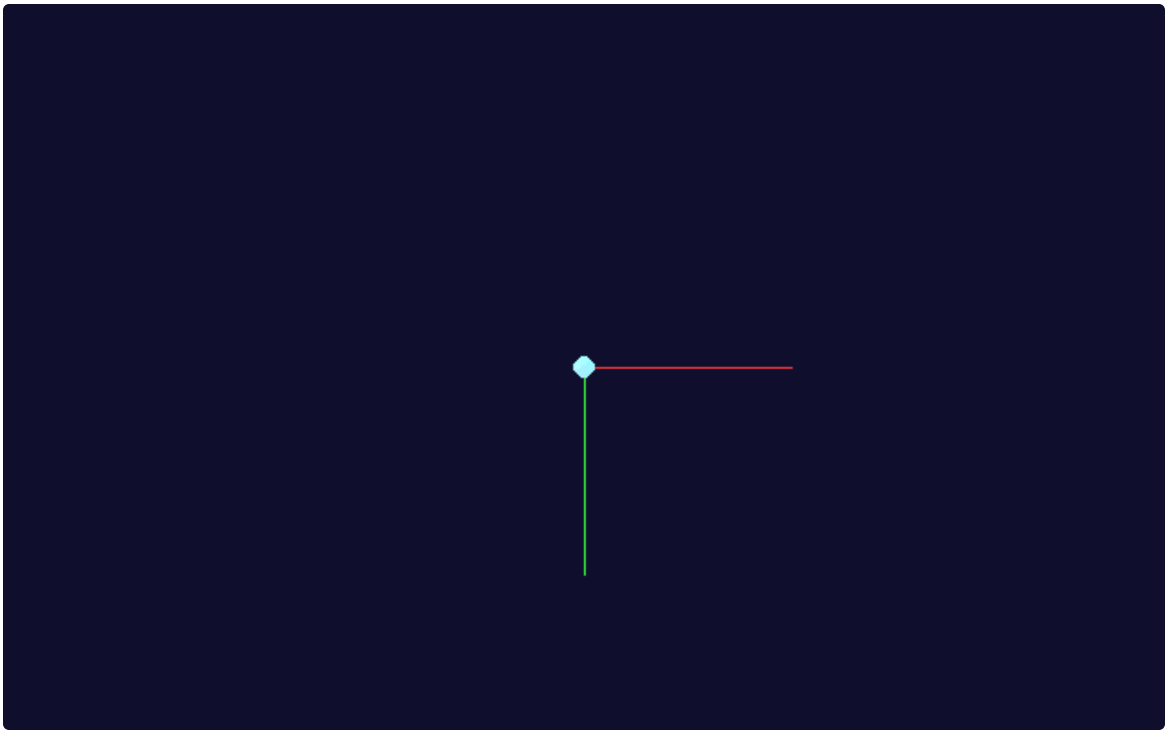


Figure: The world axes. The red line is x (right), the green line is y (down) and the blue line is z, pointing straight into the screen so that it is seen end-on.

The surprise is that **y points down**, like the rows on a text page. So "above" means *negative* y. The ceiling of a room is at `y = -3` and the floor at `y = 3`. If your shape appears upside down or in the wrong place, check your signs.

> TIP
>
> Make a note on a sticky label for your monitor: "x right, y DOWN, z away".

A row of shapes

Ling comes with more than twenty ready-made solids: `cube`, `sphere`, `cylinder`, `cone`, `torus`, `pyramid`, `gear`, `helix`, `stairs` and more. Let us put four of them in a row, each spinning at its own angle:

```
GUI
bind start = do {
  open_window(800, 500, "Shapes")
  set_shade_mode(1)
  set_rim(0, 0, 0, 0)
  set_shadow_color(30, 30, 50)
  set_ambient(0.3)
  bind t = 0
  while window_is_open() {
    fill(20, 22, 48)
    clear_lights()
    add_light(-3, -4, 0, 1, 1, 1, 0.7, 30)

    set_color(230, 70, 70)
    sphere(-4.5, 0, 4, 1, 1, 1, 0, t, 0)

    set_color(70, 190, 110)
    cube(-1.5, 0, 4, 0.8, 0.8, 0.8, t, t, 0)

    set_color(80, 140, 240)
    torus(1.5, 0, 4, 1, 1, 1, 1.2, t, 0)

    set_color(240, 200, 60)
    pyramid(4.5, 0.3, 4, 1, 1, 1, 0, t, 0)

    present()
    bind t = t + 0.03
  }
}
```

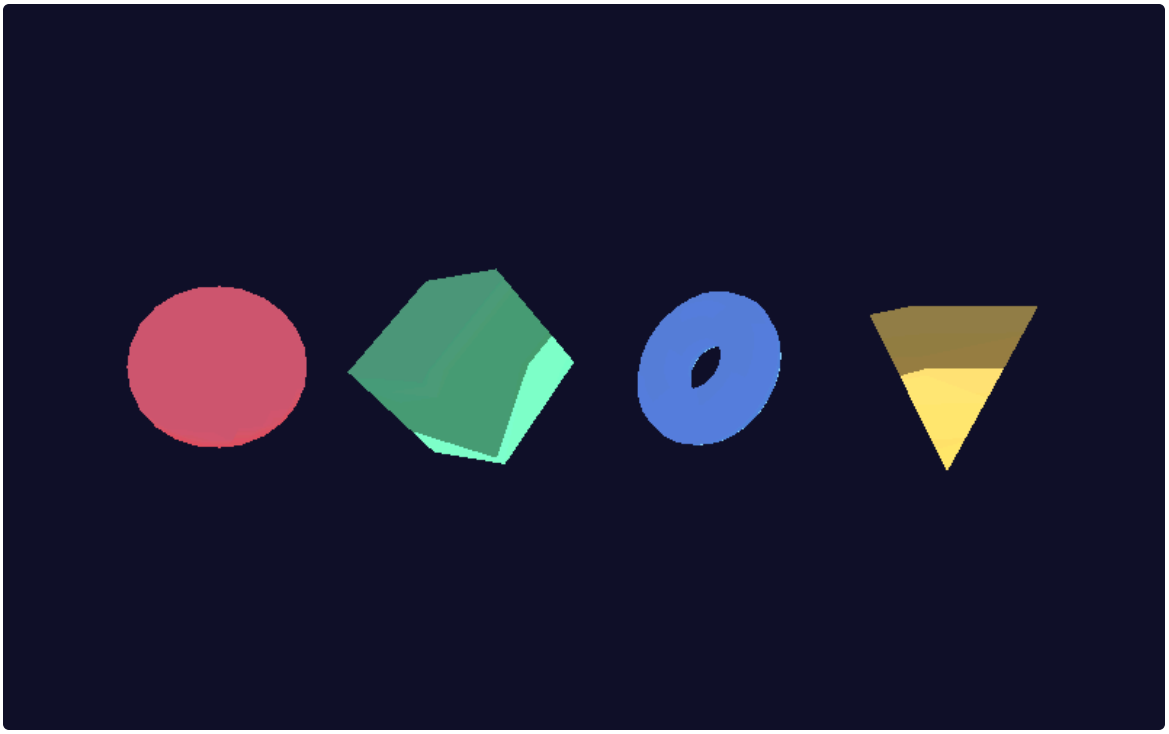


Figure: A sphere, a cube, a torus and a pyramid, as drawn by the program above.

The program starts with some "set-up" lines which run once. We will explain them in a moment; for now notice how it is organized. Everything before the `while` runs once. Everything inside runs every frame. Every frame we clear the lights, add the light again, choose a colour, and draw four shapes whose y rotation is `t`.

Look at the pyramid: it points *down* because of the y-axis rule. Rotate it by `3.14159` around the x axis (the `rx` slot) to turn it point-up.

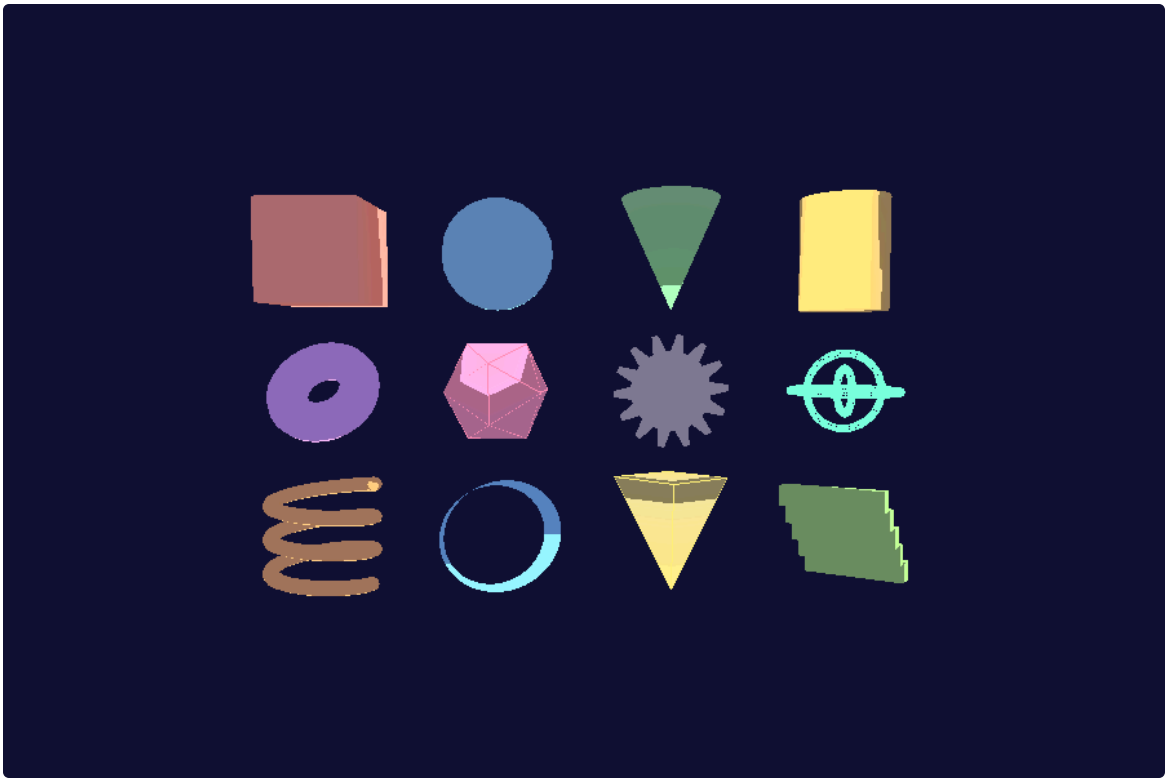


Figure: A dozen of the solids: cube, sphere, cone, cylinder, torus, icosahedron, gear, gyro, helix, arch, pyramid and stairs.

> **TRY IT**
>
> Swap the `torus` for a `gear` or a `helix`. Set the last (`mode`) argument of one shape to `1` to see its wireframe.

The camera

You are the camera. By default it sits at the origin looking along the z axis, which is why shapes at `z = 4` show up in front of you. You can change what the camera sees with three tools:

- `set_camera_pos(x, y, z)` moves the camera. (Technically it chooses the point the world is rotated around, but you can imagine you simply walked there.)

- `set_camera(cos_yaw, sin_yaw, cos_pitch, sin_pitch)` turns it. This is an odd interface: you give it the **cosine and sine** of the angle, not the angle itself. To turn left and right by an angle `a`, call `set_camera(cos(a), sin(a), 1, 0)`. The last two numbers, `1, 0`, mean "do not tilt up or down".
- `set_zdist(d)` is a depth offset that works like the position of your eye behind the picture. A small value like `0.5` puts your eye almost at the camera position, which is right for a first-person view inside a room.

A positive yaw turns the camera to the right (so the scene slides left on the screen). To walk *forward* in the direction you are facing after turning by `yaw`, move by `sin(yaw)` along x and `cos(yaw)` along z. You will see that formula in the room program below.

> NOTE
>
> Call the camera functions **every frame**, before you draw. The camera is part of the state of each frame.

Colour and light

Colours in Ling are three numbers from 0 to 255. Pure red is `255, 0, 0`; white is `255, 255, 255`; mid-grey is `128, 128, 128`. A handy way to find a colour is to open any paint program, pick a colour, and copy the red, green and blue values.

> WARNING
>
> The numbers are **not clamped**. If you write `300` for a channel, it does not become 255: it spills over into the next channel and gives a strange colour. Keep every value from 0 to 255.

Ling's default look is deliberately stylish: shadow areas are tinted indigo, edges glow cyan, and surfaces have a faint holographic sheen. That looks great for science-fiction scenes but washes out your pen colours. The set-up lines in our programs switch to calmer shading:

FRAGMENT

```
set_shade_mode(1)              # smooth, no sheen
set_rim(0, 0, 0, 0)            # no glowing edge
set_shadow_color(30, 30, 50)   # a dark neutral shadow
set_ambient(0.3)               # a little light everywhere
```

You can see the default look in the first spinning cube, which is cool-coloured even though we asked for orange. Add the four lines above and the cube turns into a flat, muted orange-red (with only ambient light and no `add_light`, it is dim and shows no shading). Experiment: the difference is the quickest way to learn what each line does.

Lights

Shapes are lit by up to several coloured point lights. A light is added with

```
FRAGMENT
add_light(x, y, z,   r, g, b,   intensity, radius)
```

The position comes first, then the colour (each from 0 to 1, or 0 to 255), then the brightness, and finally the **radius**: the distance at which the light fades out. Because lights accumulate frame after frame, call `clear_lights()` at the start of each frame and add them again.

Here is a scene with a warm orange light on the left and a cool blue one on the right:

```
GUI
bind start = do {
  open_window(800, 500, "Lights")
  set_shade_mode(1)
  set_rim(0, 0, 0, 0)
  set_shadow_color(20, 20, 40)
  set_ambient(0.25)
  bind t = 0
  while window_is_open() {
    fill(12, 12, 28)
    clear_lights()
    # a warm light on the left and a cool one on the right
    add_light(-3, -2, 1, 1.0, 0.55, 0.2, 1.6, 12)
    add_light(3, -2, 1, 0.2, 0.5, 1.0, 1.6, 12)

    set_color(235, 235, 235)
    sphere(-3, 0, 4, 1.3, 1.3, 1.3, 0, 0, 0)
    sphere(0, 0, 4, 1.3, 1.3, 1.3, 0, 0, 0)
    sphere(3, 0, 4, 1.3, 1.3, 1.3, 0, 0, 0)
    cube(0, 2.4, 4, 6, 0.1, 1.5, 0, 0, 0, 0)

    present()
    bind t = t + 1
  }
}
```

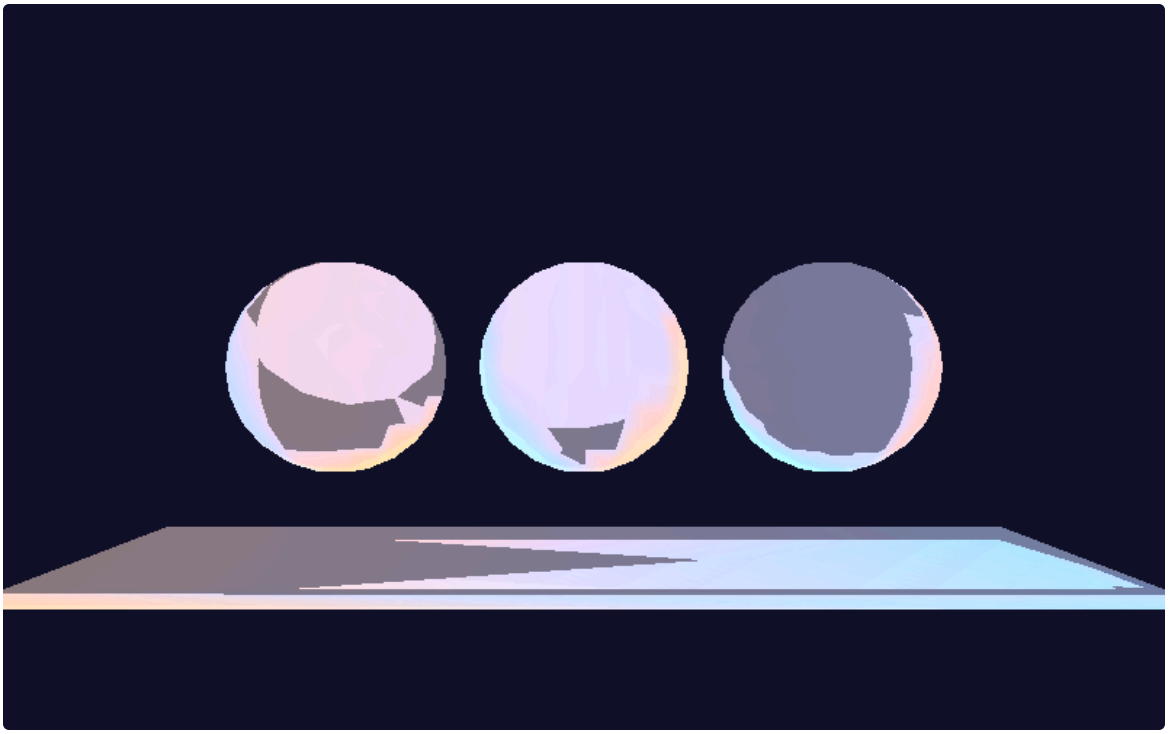


Figure: Warm and cool lights tint the white spheres differently on their two sides.

The balls are all pure white (`235, 235, 235`), yet they pick up pink on one side and blue on the other. That is what coloured lighting does.

Patterns made of lines: `vtex`

Solids are not the only things you can place in space. The `vtex_*` family draws **vector textures**: patterns made of lines, laid on a flat sheet that you can orient any way you like. They are cheap to draw and wonderful for walls, floors and decorative art. Examples are `vtex_grid`, `vtex_rings`, `vtex_star`, `vtex_flower`, `vtex_spiral`, `vtex_lotus`, `vtex_chakra`, `vtex_yantra` and `vtex_pagoda`.

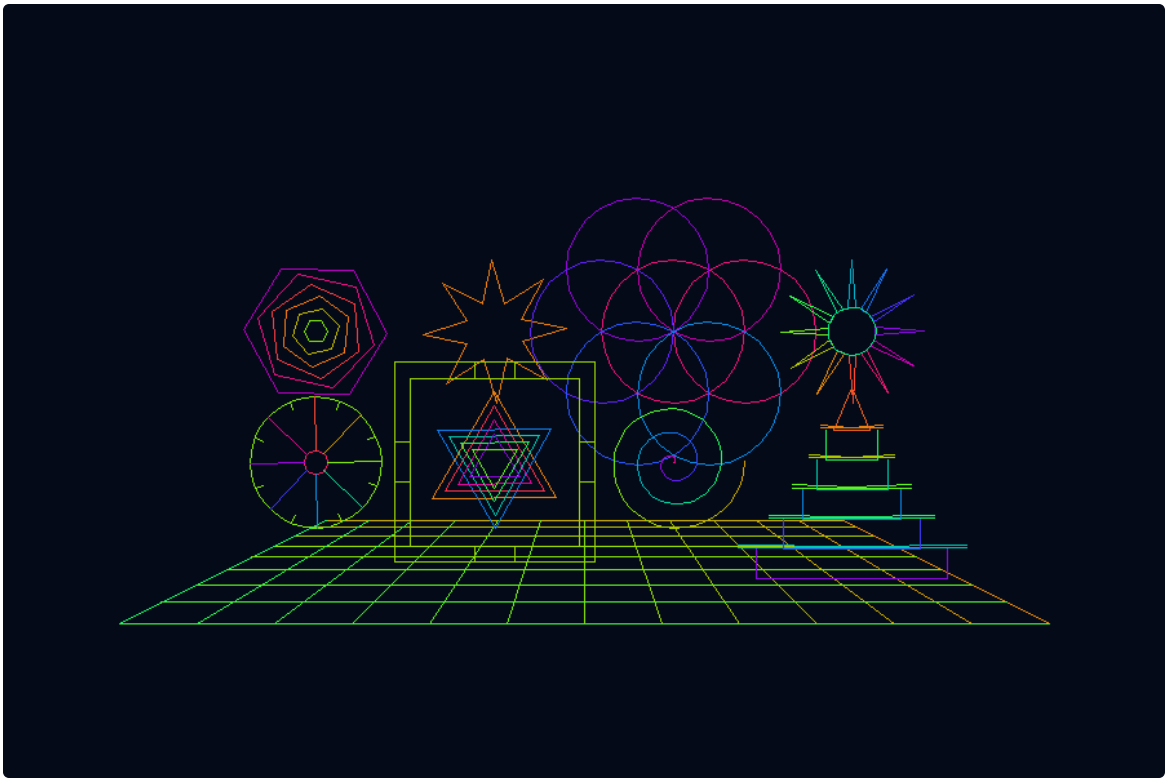


Figure: Eight vtex patterns above a vtex_grid floor.

Every vtex call begins with nine numbers that describe the **sheet**:

```
FRAGMENT
vtex_flower(cx, cy, cz,   ux, uy, uz,   vx, vy, vz,   ...pattern numbers...,   fr, hue)
```

- `cx, cy, cz` is the centre of the pattern.
- `ux, uy, uz` is the direction that the sheet's "right" points.
- `vx, vy, vz` is the direction of its "up".
- At the end, `fr` is a frame counter that you pass in, which animates the colours, and `hue` shifts the colour so that neighbouring patterns look different.

Three combinations cover most rooms:

Sheet	U (right)	V (up)
floor or ceiling	`1, 0, 0`	`0, 0, 1`
upright wall facing you	`1, 0, 0`	`0, -1, 0`

Sheet	U (right)	V (up)
upright wall to your left or right	`0, 0, 1`	`0, -1, 0`

Notice `0, -1, 0` for "up": because y points down, the way to say "up" is a *negative* y.

> **NOTE**
>
> vtex colours come from a rainbow that depends on `fr` and `hue`. They do not use your pen colour from `set_color`.

A spinning mandala

Now for the fun part. We can **rotate a sheet** by changing U and V. If `a` is an angle, then `U = (cos a, sin a, 0)` and `V = (sin a, -cos a, 0)` are a sheet turned by that angle. (U and V must stay at right angles to each other. If you get the sign of one wrong, the pattern looks squashed.) Layer several patterns turning at different speeds, and you get a mandala:

GUI

```
# A spinning mandala made of vector textures.
bind start = do {
  open_window(640, 640, "Mandala")
  bind t = 0
  while window_is_open() {
    fill(6, 4, 16)
    set_camera(1, 0, 1, 0)

    # Turn the pattern by changing its "right" (U) and "up" (V) directions.
    bind a = t * 0.01
    bind ux = cos(a)
    bind uy = sin(a)
    bind vx = sin(a)
    bind vy = 0 - cos(a)
    vtex_rings(0, 0, 0,  ux, uy, 0,  vx, vy, 0,  7, 8, 2.2, 0.2,  t, 0)
    vtex_star(0, 0, 0,  ux, uy, 0,  vx, vy, 0,  12, 2.0, 1.2, 0.0,  t, 1)

    # A second layer turns the other way.
    bind b = 0 - t * 0.017
    bind ux = cos(b)
    bind uy = sin(b)
    bind vx = sin(b)
    bind vy = 0 - cos(b)
    vtex_lotus(0, 0, 0,  ux, uy, 0,  vx, vy, 0,  0.3, 1.5, 12,  t, 2)

    # The centre stays still.
    vtex_flower(0, 0, 0,  1, 0, 0,  0, -1, 0,  0.7, 18,  t, 3)

    present()
    bind t = t + 1
  }
}
```

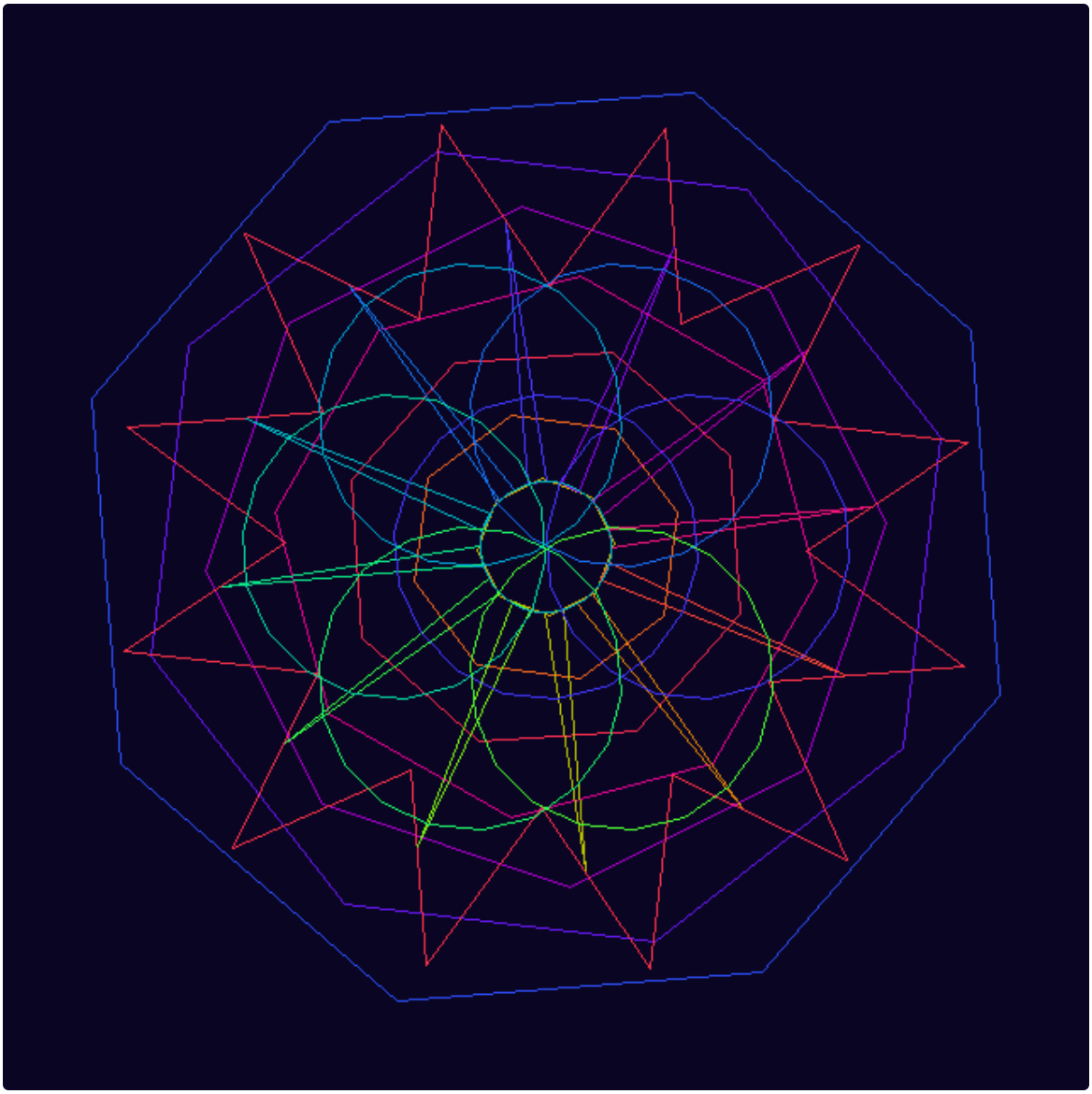


Figure: The mandala, a moment in its spin. The star and rings turn one way; the lotus turns the other way.

Note the structure. Inside the loop, we work out `a` from the counter `t`; from `a` we compute the four numbers `ux, uy, vx, vy`; we pass them to the first two patterns. Then we rebind the same four names for the opposite rotation `b` and pass them to the lotus. Re-using names like this is perfectly fine, since `bind` simply puts a name on a new value.

> **TRY IT**
>
> Change the `0.01` and `0.017` speeds. Make the lotus use `t` instead of `0 - t * 0.017` so that it turns the same way as the rings. Change the last number (the `hue`) of each pattern to see different colour combinations. Raise or lower the `12` in `vtex_star` to change the number of points.

Taking screenshots

Programs can photograph themselves. The call `screenshot("tag")`, placed *after* `present()`, saves the current frame as a PNG file in a folder called `screenshots` in the directory where you launched the program. The file name contains the time, your tag and the window size, for example `ss_1793232922_room_960x600.png`.

You can ask for one at any time, for instance when the player presses a key. The room program below uses `if key_pressed("p") { screenshot("room") }`: press P and you get a picture. Every figure in this chapter was made that way.

You can also make a program photograph itself automatically and then quit. This is handy when you want to produce a picture from a script, and it is also the best way to *test* a graphical program without watching it:

GUI

```
# A window that closes itself after 60 frames, then reports.
bind start = do {
  open_window(480, 300, "Counting frames")
  bind n = 0
  while window_is_open() {
    fill(10, 20, 50)
    set_color(255, 255, 255)
    sphere(0, 0, 3, 1, 1, 1, 0, n * 0.05, 0, 0)
    present()
    bind n = n + 1
    if n >= 60 {
      quit()
    }
  }
  println("I drew ", n, " frames")
  println("frame_count says ", frame_count())
}
```

OUTPUT

```
I drew 60 frames
frame_count says 60
```

The program counts frames, closes the window with `quit()` after 60, and then prints a report. The code after the loop runs once the window has closed, so a windowed program can leave text output in the terminal, too. `frame_count()` is a built-in counter that agrees with our own.

TIP

There is no built-in "delta time" function in this version of Ling. If you need to know how many seconds passed between frames, call `time_now()` (seconds since the program started) and subtract the value you saved on the previous frame.

Finishing touch: a room you can walk around

Now we combine everything: a floor and ceiling of grids, four walls with decorations, four pillars, lights, a spinning centrepiece, and keyboard control. Read it slowly. It is longer than anything so far, but it is built from pieces you know.

```
GUI
# A room you can walk around in.
# Arrow keys: left/right turn, up/down walk.  P takes a screenshot.  Escape quits.

fn pillar(x, z) {
  set_color(200, 190, 170)
  cylinder(x, 0, z, 0.5, 3, 0.5, 0, 0, 0, 0)
  set_color(150, 140, 120)
  cube(x, 3, z, 0.8, 0.12, 0.8, 0, 0, 0, 0)
  cube(x, -3, z, 0.8, 0.12, 0.8, 0, 0, 0, 0)
}

fn walls(t) {
  # floor and ceiling (+y is DOWN, so the floor is y = 3)
  vtex_grid(0, 3, 0,  1, 0, 0,  0, 0, 1,  16, 16, 1, 1,  t, 0)
  vtex_grid(0, -3, 0, 1, 0, 0,  0, 0, 1,  16, 16, 1, 1,  t, 2)
  # four walls, upright: V = (0, -1, 0)
  vtex_grid(0, 0, 8,   1, 0, 0,  0, -1, 0,  16, 6, 1, 1,  t, 1)
  vtex_grid(0, 0, -8,  1, 0, 0,  0, -1, 0,  16, 6, 1, 1,  t, 1)
  vtex_grid(-8, 0, 0,  0, 0, 1,  0, -1, 0,  16, 6, 1, 1,  t, 3)
  vtex_grid(8, 0, 0,   0, 0, 1,  0, -1, 0,  16, 6, 1, 1,  t, 3)
  # a wall decoration on each wall
  vtex_flower(0, 0, 7.9,   1, 0, 0,  0, -1, 0,  1.6, 24,  t, 0)
  vtex_rings(0, 0, -7.9,   1, 0, 0,  0, -1, 0,  6, 6, 1.8, 0.2,  t, 1)
  vtex_star(-7.9, 0, 0,    0, 0, 1,  0, -1, 0,  10, 1.8, 0.8, 0.0,  t, 2)
  vtex_lotus(7.9, 0, 0,    0, 0, 1,  0, -1, 0,  0.4, 1.7, 12,  t, 3)
}

bind start = do {
  open_window(960, 600, "My 3D room")
  set_shade_mode(1)
  set_rim(0, 0, 0, 0)
  set_shadow_color(30, 30, 50)
  set_ambient(0.3)

  bind px = 0
  bind pz = -5
  bind yaw = 0
```

```
bind t = 0

while window_is_open() {
  # --- input: update the camera numbers ---
  if key_down("left") {
    bind yaw = yaw - 0.03
  }
  if key_down("right") {
    bind yaw = yaw + 0.03
  }
  if key_down("up") {
    bind px = px + sin(yaw) * 0.1
    bind pz = pz + cos(yaw) * 0.1
  }
  if key_down("down") {
    bind px = px - sin(yaw) * 0.1
    bind pz = pz - cos(yaw) * 0.1
  }

  # --- draw ---
  fill(8, 8, 20)
  set_camera(cos(yaw), sin(yaw), 1, 0)
  set_camera_pos(px, 0, pz)
  set_zdist(0.5)

  clear_lights()
  add_light(0, -2.5, 0, 1.0, 0.85, 0.6, 1.0, 14)

  walls(t * 0.3)
  pillar(-6, -6)
  pillar(6, -6)
  pillar(-6, 6)
  pillar(6, 6)

  # the centrepiece: a bobbing, spinning torus and a sphere in the middle
  set_color(240, 200, 80)
  torus(0, 1.5 + sin(t * 0.05) * 0.3, 0, 0.7, 0.7, 0.7, 1.2, t * 0.03, 0)
  set_color(90, 160, 255)
  sphere(0, 1.5 + sin(t * 0.05) * 0.3, 0, 0.3, 0.3, 0.3, 0, 0, 0)
```

```
    present()
    if key_pressed("p") {
      screenshot("room")
    }
    bind t = t + 1
  }
}
```

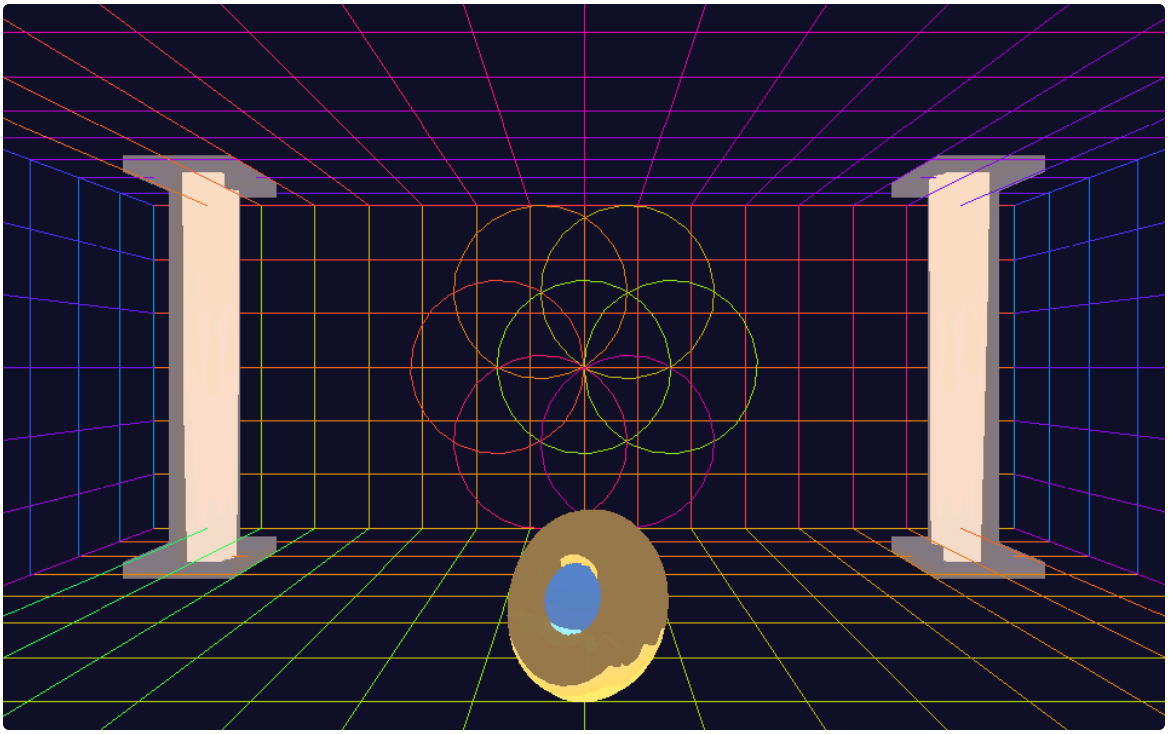


Figure: The room seen from its starting position. The flower on the far wall is vtex_flower, the floor and walls are vtex_grid, the pillars are cylinders, and the bobbing centrepiece is a torus around a sphere.

How it is organized:

- `pillar(x, z)` and `walls(t)` are **functions** that draw parts of the room. Because they are called from inside the frame loop, they draw anew every frame. Remember from Chapter 4 that functions see their parameters and top-level names. That is why we pass `t` in explicitly.
- Before the loop, `px`, `pz` and `yaw` hold the position and heading of the camera. These are updated by the key tests in the loop.

- `key_down("left")` is true while the left arrow is held. Turning changes `yaw`; walking moves `px` and `pz` by a small step in the facing direction `(sin(yaw), cos(yaw))`.
- The input section comes *before* drawing, so every frame uses the freshest camera numbers.
- The lamp sits at `(0, -2.5, 0)`: remember that negative y means up, so it hangs near the ceiling.
- The floor is at `y = 3` and the ceiling at `y = -3`, matching the pillars, which are 3 units tall each side of the centre.

> **WARNING**
>
> The 3D world has no walls in the sense of collisions. You can walk right through the grids and look at the room from outside. The room is only drawn, not solid. (Chapter 9 and later editions cover physics.)

Walls that you have walked through, or are standing right behind, can show stray lines. That is a limit of the simple perspective used here, not a mistake in your code.

Where to go from here

You have used maybe a tenth of the graphics toolbox. In the Reference you will find many more shapes, `tex_*` pixel textures such as the Mandelbrot fractal, post-processing effects such as bloom and outlines, and text and user-interface drawing. Here is the same room kind of scene with some of that toolbox applied:

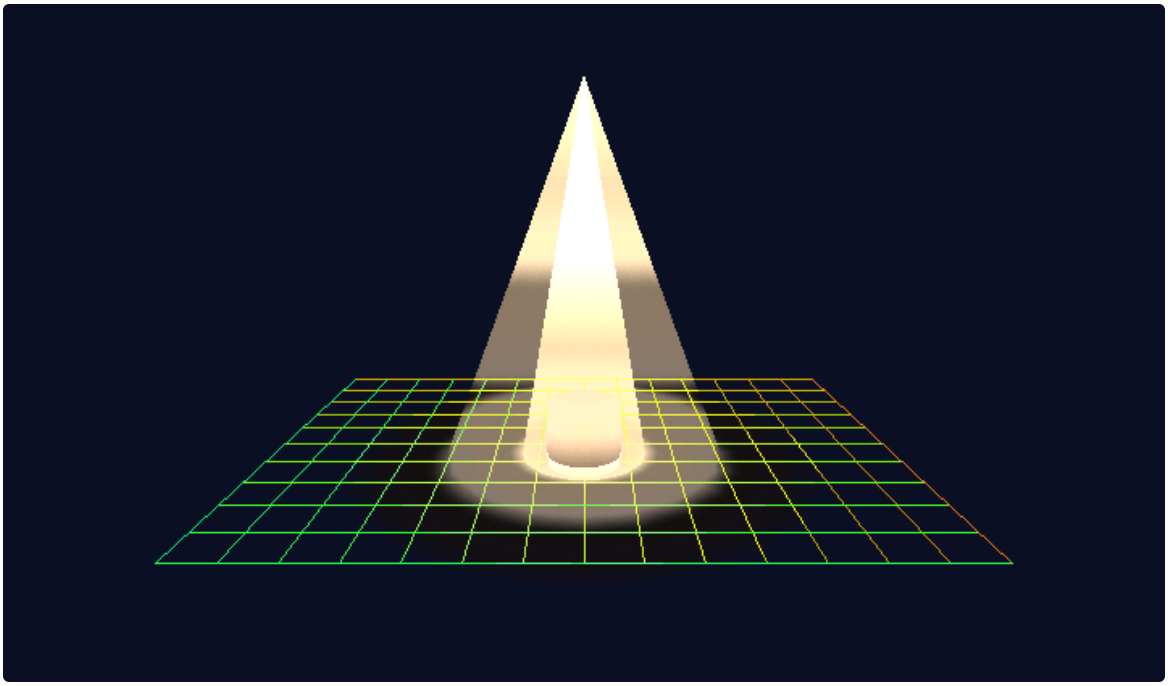


Figure: Light pools and light beams give a scene atmosphere. Effects like these are covered in the Intermediate edition.

Chapter summary

- A graphical program is a loop: open the window once, then each frame: `fill`, draw, `present()`; stop when `window_is_open()` is false (Escape or close).
- `present()` is the only call that shows pixels. Nothing appears without it.
- World axes: x right, y **down**, z away from you. Up means negative y.
- Shapes share one call format: centre, size, rotation, mode. `cube`, `sphere`, `cylinder`, `cone`, `torus`, `pyramid`, and many more.
- Colours are `0` to `255`, not clamped. Use `set_shade_mode(1)`, `set_rim(0, 0, 0, 0)`, `set_shadow_color(...)` and `set_ambient(...)` for faithful colours.
- Lights: `clear_lights()` then `add_light(x, y, z, r, g, b, intensity, radius)` each frame.
- Camera: `set_camera(cos a, sin a, 1, 0)` takes cosines and sines; `set_camera_pos`, `set_zdist`.
- `vtex_*` patterns are drawn on a sheet defined by a centre and U and V directions; floor, wall and side-wall sheets have standard values.

- `screenshot("tag")` after `present()` saves a PNG; `quit()` ends the program. Use a frame counter with `quit()` to test GUI programs.
- Unknown function names are silently ignored, so check the spelling when a call seems to do nothing.

Exercises

1. Make the first program's cube grow and shrink, by using `1 + 0.5 * sin(angle)` as its size.
2. Add a floor to the "row of shapes" program with a `vtex_grid` at `y = 1.5`, using the floor sheet (`U = 1,0,0`, `V = 0,0,1`).
3. Change the room's wall pattern colours by editing the last number (`hue`) of each `vtex_grid` call. Also try changing `t * 0.3` in `walls(t * 0.3)` to `0` and see what happens to the colours.
4. Add a fifth pillar in the middle of the room, and add a second light in a different colour.
5. (Challenge) Make the mandala react to the mouse: pass `mouse_x() * 0.001` instead of `t * 0.01` as the rotation angle.

Answers

1. In `g1`, replace the three size arguments by a name: bind `s = 1 + 0.5 * sin(angle)` inside the loop and call `cube(0, 0, 3, s, s, s, angle, angle * 0.7, 0, 0)`. Because `angle` only grows by 0.02 per frame, the breathing is slow; use `angle * 5` inside the `sin` for a faster pulse.
2. Add `vtex_grid(0, 1.5, 4, 1, 0, 0, 0, 0, 1, 12, 6, 1, 1, t * 30, 0)` before `present()`. The grid is centred at `(0, 1.5, 4)`, 12 columns by 6 rows, one unit per cell.
3. The `hue` just rotates the colour phase; each value 0, 1, 2, 3 gives a different colour set. With `walls(0)` the colours are frozen, because `fr` never changes.
4. `pillar(0, 0)` draws a pillar at the centre; it hides the centrepiece, so move that or use `pillar(3, 0)`. A second light is one more `add_light(...)` call right after the first, for example `add_light(0, -2.5, 4, 0.3, 0.5, 1.0, 1.0, 14)`.
5. `mouse_x()` returns the pointer's x position in pixels. Multiplying by `0.001` converts it to a small angle, so moving the pointer across a 640-pixel window turns the mandala about 0.64 radians (roughly 37 degrees).

CHAPTER 7

Drawing and Games

What you'll learn. In the last chapter you worked in 3D. Here you will go back to the flat page and draw pictures with circles, triangles, gradients and arcs. Then you will make your pictures respond to the keyboard and the mouse, and put everything together into a complete little game that you can play and change.

> NOTE
>
> Programs in this chapter open a window and are labelled **GUI**. Press **Escape** to close any of them.

The 2D canvas

A 2D drawing is like painting on graph paper. The window is a grid of pixels, and every drawing call tells Ling which pixels to colour. Two things are different from maths class:

- The origin `(0, 0)` is the **top-left** corner of the window.
- **y grows downward.** A point at `y = 400` is lower on the screen than a point at `y = 100`.

So in an 800 by 500 window, the centre is `(400, 250)`, the top-right is `(800, 0)` and the bottom-right is `(800, 500)`. This is the same "y points down" rule you met in 3D, and it is how nearly every screen works.

You draw with a **pen**. `set_color(r, g, b)` chooses the pen colour, and every shape you draw after that uses it, until you change it. This is the opposite of putting the colour in each call, and it makes drawing code short.

Here are the main tools:

Call	Draws
`fill(r, g, b)`	clears the whole window to one colour
`draw_disc(cx, cy, r)`	a filled circle
`draw_circle(cx, cy, r)`	a circle outline
`triangle(x0, y0, x1, y1, x2, y2)`	a filled triangle
`draw_line(x0, y0, x1, y1)`	a thin line

Call	Draws
`draw_arc(cx, cy, r, a0, a1, segments)`	part of a circle outline, angles in radians, clockwise
`draw_poly([x0, y0, x1, y1, ...])`	a filled polygon from a list of points
`grad_rect(x, y, w, h, r0,g0,b0, r1,g1,b1, dir)`	a rectangle fading between two colours
`set_color_hsl(h, s, l)`	pen colour by hue 0-360, saturation 0-100, lightness 0-100

You might wonder where the rectangle is. There is no plain filled-rectangle call. But a gradient from a colour to

the same colour

is just a flat rectangle, so we can write our own. That is the perfect use for a function (Chapter 4):

FRAGMENT

```
fn rect(x, y, w, h, r, g, b) {
  grad_rect(x, y, w, h, r, g, b, r, g, b, 1)
}
```

Now `rect(0, 380, 800, 120, 50, 110, 60)` paints a green strip. Building your own small tools on top of what the language provides is a big part of programming.

TIP

Draw in layers, from the back to the front. Later shapes paint over earlier ones, exactly like painting on a canvas. Sky first, mountains next, trees last.

A landscape

Here is a complete scene. Read it from the top and imagine the picture growing layer by layer.

GUI

```
# A little 2D landscape.  The pen colour applies to everything drawn after set_color.

# There is no rectangle call, but a gradient with two equal colours is a flat
rectangle.
fn rect(x, y, w, h, r, g, b) {
  grad_rect(x, y, w, h, r, g, b, r, g, b, 1)
}

fn tree(x, ground) {
  rect(x - 6, ground - 50, 12, 50, 110, 70, 40)
  set_color(40, 150, 70)
  draw_disc(x, ground - 70, 32)
  draw_disc(x - 20, ground - 50, 24)
  draw_disc(x + 20, ground - 50, 24)
}

bind start = do {
  open_window(800, 500, "Landscape")
  bind t = 0
  while window_is_open() {
    # sky: a gradient from deep blue (top) to pink (horizon)
    grad_rect(0, 0, 800, 380, 30, 40, 110, 250, 150, 140, 1)

    # sun
    set_color(255, 220, 120)
    draw_disc(600, 300, 60)

    # mountains
    set_color(70, 60, 110)
    triangle(0, 380, 150, 190, 300, 380)
    triangle(200, 380, 380, 150, 560, 380)
    set_color(95, 85, 140)
    triangle(450, 380, 620, 220, 800, 380)

    # ground
    rect(0, 380, 800, 120, 50, 110, 60)
```

```
    tree(120, 440)
    tree(330, 460)
    tree(690, 450)

    # a bird made of two arcs, flapping with the frame counter
    set_color(20, 20, 30)
    bind flap = sin(t * 0.15) * 6
    draw_arc(150, 120 - flap, 18, 3.4, 6.0, 16)
    draw_arc(186, 120 - flap, 18, 3.4, 6.0, 16)

    present()
    bind t = t + 1
  }
}
```

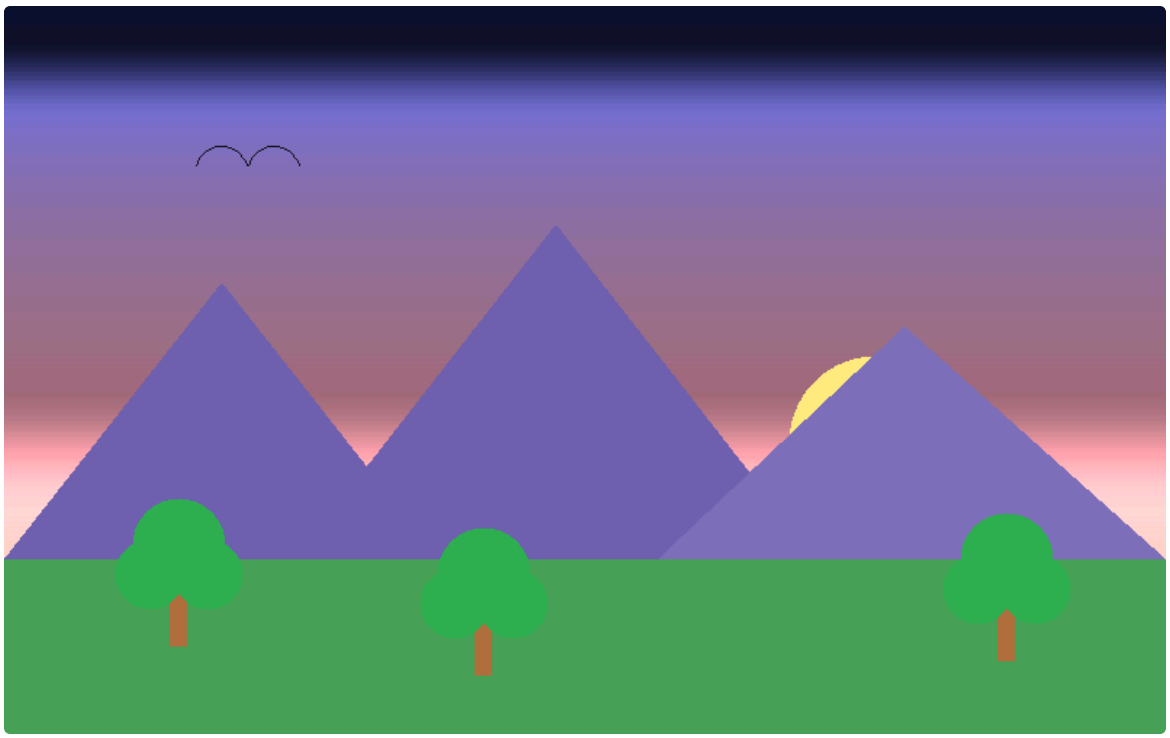


Figure: The landscape. The sky is a gradient, the mountains are triangles, the trees are discs and a rectangle, and the bird is two arcs.

Some details worth noting:

- The sky is one `grad_rect` from dark blue at the top to pink at the horizon (the last argument `1` means top-to-bottom; `0` would be left-to-right).
- `tree(x, ground)` is a function: the trunk and leaves are placed relative to `x` and `ground`, so we can plant as many trees as we like by calling it again.
- The bird's wings are arcs. Angles are in radians and grow *clockwise* because y points down: `0` is to the right and `3.14159` is to the left. The arcs from `3.4` to `6.0` make the upper curve, like a "v" with curved wings.
- The wing flap uses `sin(t * 0.15) * 6`, a number that swings smoothly between -6 and 6, which makes the bird move up and down. Sine waves are the animator's best friend.

> **TRY IT**
>
> Make the sun set. Bind a name `sun_y = 300 + t * 0.2` and use it as the sun's y. Add a few more trees with different x positions, or make a second bird. Try `set_color_hsl` for the leaves.

Listening to the keyboard

A game must react to its player. In Ling, you **poll** the input: once per frame, you ask "is this key down right now?" There are no event handlers or callbacks. That keeps things simple.

- `key_down("left")` is true as long as the left arrow is *held*.
- `key_pressed("space")` is true only on the single frame when space *goes down*. Use it for things that happen once per press: shooting, jumping, pausing.

Keys are named in lower case: the letters `"a"` to `"z"`, the digits, `"left"`, `"right"`, `"up"`, `"down"`, `"space"`, `"enter"`, `"escape"`, `"lshift"`, `"f1"` and many others. An unknown name is just false.

Here is the classic first interactive program: a ball you can move.

```
GUI
# Move a ball with the arrow keys. Hold Shift to go faster.
bind start = do {
  open_window(640, 400, "Move the ball")
  bind x = 320
  bind y = 200
  while window_is_open() {
    bind speed = 3
    if key_down("lshift") {
      bind speed = 8
    }
    if key_down("left") {
      bind x = x - speed
    }
    if key_down("right") {
      bind x = x + speed
    }
    if key_down("up") {
      bind y = y - speed
    }
    if key_down("down") {
      bind y = y + speed
    }
    bind x = clamp(x, 20, 620)
    bind y = clamp(y, 20, 380)

    fill(10, 10, 30)
    set_color(255, 180, 60)
    draw_disc(x, y, 20)
    present()
  }
}
```

The ingredients of nearly every game are visible here:

1. **State**: the numbers `x` and `y`, which describe where the ball is. They live outside the loop so they survive from frame to frame.
2. **Input**: the `if key_down(...)` lines change the state.
3. **Draw**: the picture is created from the state.

Note `bind speed = 3` at the start of each frame, followed by a Shift test that raises it to `8`. Also note `clamp(x, 20, 620)`: it forces the ball to stay inside the window.

> WARNING
>
> Keep the order **input, update, draw**. If you draw first and update afterwards, your picture is one frame behind the state.

Listening to the mouse

`mouse_x()` and `mouse_y()` give the pointer's position in pixels. `mouse_down()` is true while the left button is held, and `mouse_down_right()` while the right one is held.

GUI

```
# Paint with the mouse: hold the left button to draw, right button to clear.
bind start = do {
  open_window(640, 400, "Mouse paint")
  fill(245, 240, 225)
  bind hue = 0
  while window_is_open() {
    if mouse_down() {
      set_color_hsl(hue, 80, 50)
      draw_disc(mouse_x(), mouse_y(), 10)
      bind hue = hue + 3
      if hue >= 360 {
        bind hue = 0
      }
    }
    if mouse_down_right() {
      fill(245, 240, 225)
    }
    present()
  }
}
```

Hold the left button and drag to paint with a changing rainbow colour. We do not call `fill` inside the loop, so the old paint is not erased each frame: the window keeps what we painted. (A right-click calls `fill` once to wipe it.) That is the exception to the rule from the last chapter: in a drawing program that accumulates, you clear only when you want to.

> **NOTE**
>
> If you try the paint program and notice gaps in a fast stroke, that is because we draw only one disc per frame, at the pointer's current position. A smoother brush would draw a line from the previous position to the new one. That makes a good exercise.

Making a game

Now let us build **Star Catcher**. Stars fall from the sky; you slide a basket left and right to catch them. Every catch scores a point and makes the next star fall faster. Miss three and the game ends. Press Space to try again.

Before typing any code, a game designer lists three things:

- **State**, the numbers that describe the world at this instant: the basket's x, the star's x and y, the fall speed, the score, the lives left.
- **Rules**, how the state changes: the basket obeys the keys, the star falls, and when it reaches the bottom either you catch it (it is close enough to the basket) or you lose a life.
- **Picture**, how the state is drawn: a gradient sky, a star at `(star_x, star_y)`, a basket, and the score text.

Each frame the program does:

input, update, draw.

Here is the complete game:

GUI

```
# STAR CATCHER
# Move the basket with the left and right arrow keys (or A and D).
# Catch the falling stars. Miss three and the game is over. Space restarts.

bind autoplay = false      # set to true to watch the computer play

fn rect(x, y, w, h, r, g, b) {
  grad_rect(x, y, w, h, r, g, b, r, g, b, 1)
}

# A five-pointed star, drawn as a ten-cornered polygon.
fn star(cx, cy, size, spin) {
  bind pts = list_new()
  bind i = 0
  while i < 10 {
    bind a = spin + i * 0.6283185
    bind r = size
    if i % 2 == 1 {
      bind r = size * 0.45
    }
    bind pts = list_push(pts, cx + cos(a) * r)
    bind pts = list_push(pts, cy + sin(a) * r)
    bind i = i + 1
  }
  draw_poly(pts)
}

bind start = do {
  open_window(640, 480, "Star Catcher")

  bind basket_x = 320
  bind star_x = 100 + rand() * 440
  bind star_y = 0
  bind fall = 3
  bind score = 0
  bind lives = 3
  bind t = 0
```

```
while window_is_open() {
  if lives > 0 {
    # ---- input ----
    if key_down("left") || key_down("a") {
      bind basket_x = basket_x - 7
    }
    if key_down("right") || key_down("d") {
      bind basket_x = basket_x + 7
    }
    if autoplay {
      bind basket_x = star_x
    }
    bind basket_x = clamp(basket_x, 50, 590)

    # ---- update ----
    bind star_y = star_y + fall
    if star_y > 440 {
      # the star reached the basket's height: caught or missed?
      if abs(star_x - basket_x) < 55 {
        bind score = score + 1
        bind fall = fall + 0.25
      } else {
        bind lives = lives - 1
      }
      bind star_x = 50 + rand() * 540
      bind star_y = 0
    }
  } else {
    if key_pressed("space") {
      bind lives = 3
      bind score = 0
      bind fall = 3
      bind star_y = 0
    }
  }

  # ---- draw ----
  grad_rect(0, 0, 640, 480, 15, 15, 50, 40, 25, 80, 1)
```

```
    set_color(255, 220, 80)
    star(star_x, star_y, 24, t * 0.05)

    set_color(255, 255, 255)
    rect(basket_x - 50, 445, 100, 14, 90, 190, 255)
    rect(basket_x - 50, 445, 100, 4, 200, 235, 255)

    set_color(255, 255, 255)
    ui_text(16, 14, 22, "SCORE " + to_str(score))
    ui_text(500, 14, 22, "LIVES " + to_str(lives))
    if lives <= 0 {
      set_color(255, 110, 110)
      ui_text(170, 200, 36, "GAME OVER")
      set_color(255, 255, 255)
      ui_text(170, 260, 20, "PRESS SPACE")
    }

    present()
    bind t = t + 1
  }
  println("Final score: ", score)
}
```

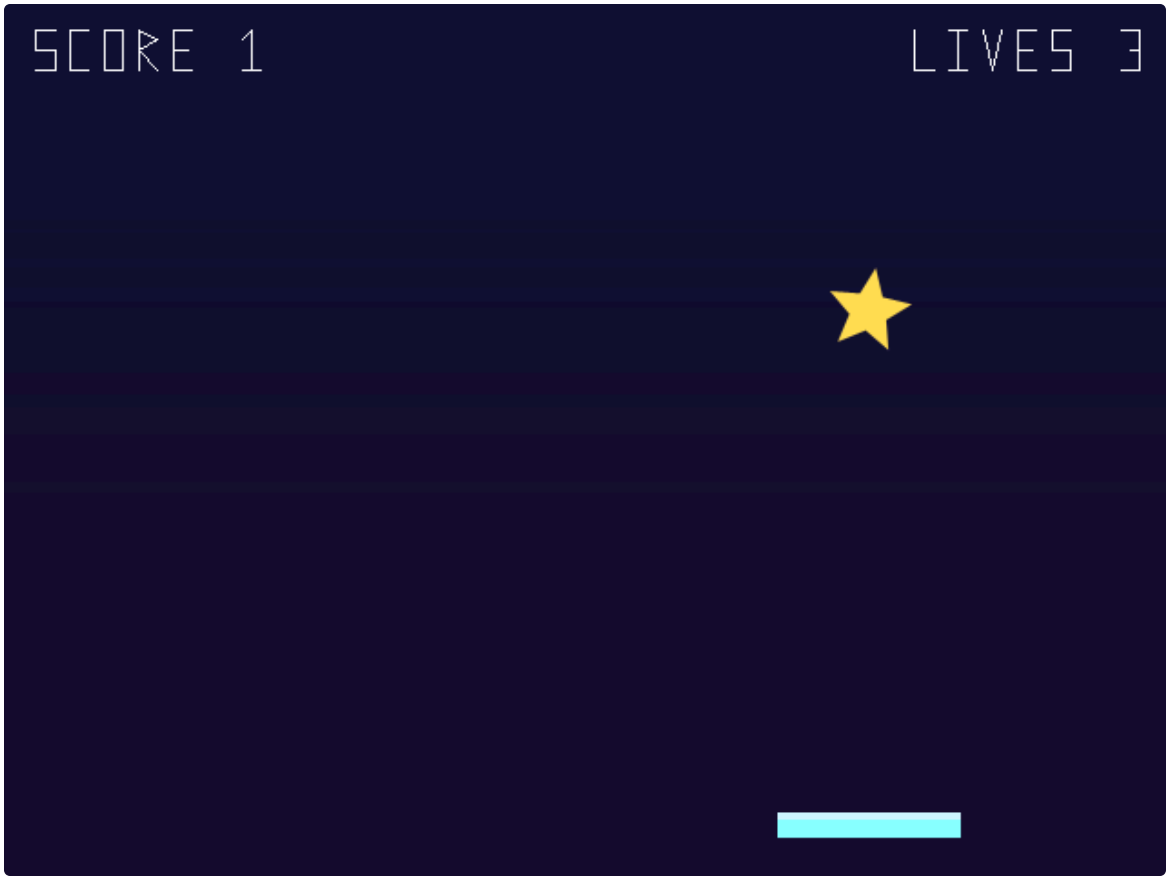


Figure: Star Catcher in play. The star spins as it falls; the basket is at the bottom.

How the game works

The star. `star(cx, cy, size, spin)` builds a list of ten corner coordinates, alternating between a big radius (the points) and a smaller one (the notches), and hands it to `draw_poly`. A list is built with a `while` loop (the for-loop trap again), and the corners come from cosine and sine, which place points around a circle. Passing a spin angle makes the star rotate.

The state machine. Look at the structure `if lives > 0 { ... } else { ... }`. While you have lives, the game is running; when they are gone, only the restart test runs. That is the simplest form of a *game state*: "playing" versus "game over". Bigger games add more states such as "menu" and "paused".

Catching. When the star's y passes 440 (the height of the basket), we compare the horizontal distance `abs(star_x - basket_x)` with 55, which is half the basket's width plus a little forgiveness. If it is smaller, you caught it. Either way, the star restarts at the top with a new

random x. `rand()` gives a random number from 0 up to (but not including) 1, so `50 + rand() * 540` is a random position between 50 and 590.

Difficulty. `bind fall = fall + 0.25` makes every next star a little quicker. That tiny line turns a toy into a game: it creates tension.

Text. `ui_text(x, y, size, text)` draws text in a built-in stroke font that needs no font file. It draws capital letters, and the colour is the pen colour. We join the label to the number with `+` and `to_str`.

The ending. When the window is closed, the loop ends, and the line after it, `println("Final score: ", score)`, prints in the terminal.

> TIP
>
> Constants like `55` and `440` are called *magic numbers*, because a reader cannot tell what they mean. For your own bigger games, give them names at the top of the file: `bind basket_half = 50`. The code is easier to read and to change.

Testing a game without playing it

How do you know a game works if you cannot sit and play it for every change? Programmers write a

test mode

. Notice the line near the top:

```
FRAGMENT
bind autoplay = false      # set to true to watch the computer play
```

and the two lines in the loop that use it:

```
FRAGMENT
if autoplay {
  bind basket_x = star_x
}
```

When `autoplay` is `true`, the basket simply follows the star, so the computer catches everything. You can watch a minute of the game without touching the keyboard. For automated checks, you can also add a frame counter that calls `quit()` after some number of frames. We used both while writing this book: the game ran about fifteen seconds with no key presses (so the player lost all three lives, tested the game-over state), and also in autoplay mode (tested the catch). That is how we know every line of this program runs.

Ideas for improvement

- Add a second kind of falling object, a bomb, which costs a life if you catch it.
- Add a high score: keep `best` and update it when the game ends.
- Use `mouse_x()` for steering instead of the keys.
- Change the sky colour every five points.
- Draw the lives as little stars instead of a number.

> **TRY IT**
>
> Make the game easier by raising the half-width of the basket check from 55 to 80 (and make the basket wider to match). Then make it harder by starting `fall` at 5.

Chapter summary

- The 2D canvas has its origin at the top-left; x grows right, y grows **down**.
- `set_color` chooses the pen; `draw_disc`, `draw_circle`, `triangle`, `draw_line`, `draw_arc`, `draw_poly` and `grad_rect` draw in it. There is no rectangle call; write your own `rect` helper around `grad_rect`.
- Draw back to front; later shapes cover earlier ones.
- Input is polled each frame: `key_down` (held), `key_pressed` (once per press), `mouse_x()`, `mouse_y()`, `mouse_down()`.
- Structure every game as **state** (names outside the loop), then each frame **input, update, draw**.
- Use `clamp` to keep things in bounds, `rand()` for randomness, and `abs` for distances.
- Use `ui_text` for text when you have no font file.
- Build lists with `while`. Use a `lives > 0` check as a simple "playing / game over" state.
- Give yourself a test mode (an `autoplay` flag, or a frame counter with `quit()`).

Exercises

1. Draw a simple face: a big yellow disc, two small dark discs for eyes, and an arc for the mouth.
2. Make the moving ball bounce instead of stopping: keep a speed `vx` and `vy`, move the ball by them each frame, and when it hits a wall reverse the sign of that speed.
3. Change the paint program so that the colour changes with the pointer's x position instead of time.

4. Add a high score to Star Catcher: remember the best score during the session and show it on screen with `ui_text`.
5. (Challenge) Add a bomb to Star Catcher: a second falling object, drawn as a dark circle, that costs a life if caught. Reuse the same pattern as the star.

Answers

1. `ling-gui bind start = do { open_window(400, 400, "Face") while window_is_open() { fill(30, 30, 60) set_color(255, 220, 60) draw_disc(200, 200, 120) set_color(40, 40, 40) draw_disc(160, 170, 14) draw_disc(240, 170, 14) draw_arc(200, 210, 60, 0.4, 2.74, 24) present() } }` The mouth is an arc from 0.4 to 2.74 radians, the lower part of a circle because angles run clockwise.
2. `ling-gui bind start = do { open_window(640, 400, "Bounce") bind x = 100 bind y = 100 bind vx = 4 bind vy = 3 while window_is_open() { bind x = x + vx bind y = y + vy if x < 20 || x > 620 { bind vx = 0 - vx } if y < 20 || y > 380 { bind vy = 0 - vy } fill(10, 10, 30) set_color(255, 120, 80) draw_disc(x, y, 20) present() } }`
3. Replace `bind hue = hue + 3` by `bind hue = mouse_x() / 640 * 360`, and remove the wrap-around test.
4. Bind `best = 0` before the loop. When a miss takes `lives` to zero, `if score > best { bind best = score }`. Draw it with `ui_text(16, 44, 22, "BEST " + to_str(best))`.
5. Give the bomb its own `bomb_x` and `bomb_y`; make it fall each frame; when `bomb_y > 440` and `abs(bomb_x - basket_x) < 55`, subtract a life; then reset it to the top. Draw it with `draw_disc(bomb_x, bomb_y, 14)` in dark grey.

CHAPTER 8

Sound and Music

What you'll learn. Ling can make sound with nothing but numbers: no recordings, no downloads. In this chapter you will learn how pitch becomes a number, play notes from your keyboard and mouse, compose a melody as a list, turn it into a real sound file, and have Ling listen to its own music and tell you about it. You will finish with a tiny melody player.

> NOTE
>
> Programs that play sound are labelled **GUI**, because they open a small window. They need working speakers or headphones, and you will hear nothing if your volume is muted. The text panels in this chapter show what the programs print, not what they sound like.

Sound is a number

When you pluck a guitar string it vibrates, and the vibration moves the air. The number of vibrations per second is the **frequency**, measured in **hertz** (Hz). A higher frequency is a higher note. Concert A, the note an orchestra tunes to, is 440 vibrations per second: 440 Hz. A computer can create a sound by producing a wave that wobbles at the right frequency. It is just arithmetic.

Musicians do not talk about hertz; they say "A", "C sharp", or "middle C". Computers use yet another numbering, called **MIDI numbers**, where every key on a piano has a whole number: middle C is **60**, the next key up (C sharp) is 61, then D is 62, and so on. A4, the concert A, is 69. Adding 12 to a MIDI number raises the note by one **octave**, which doubles its frequency.

Ling has three functions to move between these ideas:

- `music_hz(pitch)` turns a MIDI number (or a name such as `"C4"` or `"A#3"`) into a frequency in hertz.
- `music_note_name(hz)` turns a frequency into the name of the nearest note.
- Both accept ordinary numbers, so you can do arithmetic on pitches.

Let us print a table: a C major scale.

```
bind start = do {
  println("A4 is ", music_hz("A4"), " Hz")
  println("middle C (60) is ", round(music_hz(60) * 100) / 100, " Hz")
  println("C5 is ", round(music_hz("C5") * 100) / 100, " Hz")
  println("440 Hz is the note ", music_note_name(440))
  println("300 Hz is nearest to ", music_note_name(300))

  # one octave of a C major scale
  bind scale = [60, 62, 64, 65, 67, 69, 71, 72]
  bind i = 0
  while i < len(scale) {
    bind m = list_get(scale, i)
    println(m, "  ", music_note_name(music_hz(m)), "  ", round(music_hz(m)), " Hz")
    bind i = i + 1
  }
}
```

```
OUTPUT
A4 is 440 Hz
middle C (60) is 261.63 Hz
C5 is 523.25 Hz
440 Hz is the note A4
300 Hz is nearest to D4
60  C4  262 Hz
62  D4  294 Hz
64  E4  330 Hz
65  F4  349 Hz
67  G4  392 Hz
69  A4  440 Hz
71  B4  494 Hz
72  C5  523 Hz
```

The pitch numbers 60, 62, 64, 65, 67, 69, 71, 72 are the white keys of a piano from middle C up to the next C. Notice the pattern of gaps: 2, 2, 1, 2, 2, 2, 1. That pattern *is* a major scale, and it sounds "happy" no matter which key you start from. Start from 57 instead of 60 and you get an A major scale.

> TIP
>
> `music_hz(69)` is 440. Every MIDI number up or down multiplies the frequency by the twelfth root of two, about 1.0595. Twelve steps make exactly 2, one octave. You never have to remember any of this, because `music_hz` does it for you.

Your first sounds

A one-shot blip

The easiest sound is `audio_blip`. You give it a frequency, a duration in seconds, a *waveform*, and a volume:

FRAGMENT

```
audio_blip(440, 0.5, "triangle", 0.3)
```

The waveform decides the character of the sound:

Name	Sounds like
`"sine"`	pure and smooth, like a flute or a whistle
`"triangle"`	soft and hollow
`"square"`	buzzy and retro, like an old video game
`"saw"`	bright and harsh, like strings or brass
`"noise"`	hiss, useful for drums and wind

An unknown name falls back to sine, so a typo gives you a sine wave rather than an error.

A keyboard piano

Let us combine this with last chapter's keyboard polling. The home-row keys A to K play a scale:

GUI

```
# A computer-keyboard piano. Keys A S D F G H J play C D E F G A B (and K the high C).
fn key_note(i) {
  return list_get([60, 62, 64, 65, 67, 69, 71, 72], i)
}

bind start = do {
  open_window(640, 260, "Keyboard piano")
  bind keys = ["a", "s", "d", "f", "g", "h", "j", "k"]
  bind lit = -1
  bind fade = 0

  while window_is_open() {
    # input: one blip each time a key goes down
    bind i = 0
    while i < len(keys) {
      if key_pressed(list_get(keys, i)) {
        audio_blip(music_hz(key_note(i)), 0.5, "triangle", 0.3)
        bind lit = i
        bind fade = 12
      }
      bind i = i + 1
    }

    # draw: eight white keys; the one just played glows
    fill(20, 20, 35)
    bind i = 0
    while i < len(keys) {
      bind x = 20 + i * 75
      if i == lit && fade > 0 {
        grad_rect(x, 40, 70, 180, 255, 220, 90, 255, 180, 60, 1)
      } else {
        grad_rect(x, 40, 70, 180, 245, 245, 245, 205, 205, 215, 1)
      }
      set_color(40, 40, 60)
      ui_text(x + 25, 190, 20, str_upper(list_get(keys, i)))
      bind i = i + 1
    }
```

```
    set_color(255, 255, 255)
    ui_text(20, 10, 18, "PLAY WITH A S D F G H J K")
    if fade > 0 {
      bind fade = fade - 1
    }
    present()
  }
}
```

When you press a key, `key_pressed` is true for exactly one frame, so each press triggers exactly one blip. We look up the pitch with a little function, `key_note(i)`, find its frequency with `music_hz`, and play it. The "lit" key glows for twelve frames. Rows of keys are drawn with a `while` loop.

> **TRY IT**
>
> Change `"triangle"` to `"square"` and listen. Change the numbers in `key_note` to a different scale. A minor pentatonic starting on A is `[57, 60, 62, 64, 67, 69, 72, 74]`. Does anything you play sound bad? (Hard to do with a pentatonic scale, which is why guitarists love it.)

A theremin

The theremin, invented around 1920, is an instrument you play without touching it. The player's hand moves in the air near an antenna and the pitch changes. We can build a version with the mouse:

```
GUI
# A theremin: the mouse controls pitch (left to right) and loudness (bottom to top).
bind start = do {
  open_window(640, 400, "Theremin")
  audio_volume(0.5)
  while window_is_open() {
    bind freq = 110 * pow(2, mouse_x() / 640 * 3)     # three octaves, from 110 Hz up
    bind loud = (1 - mouse_y() / 400) * 0.25
    audio_tone(0, 0, 0, 3, 0, freq, loud, 5, 0.01)

    fill(10, 10, 30)
    set_color(120, 220, 255)
    draw_disc(mouse_x(), mouse_y(), 12 + loud * 80)
    set_color(255, 255, 255)
    ui_text(16, 14, 18, music_note_name(freq) + "  " + to_str(round(freq)) + " HZ")
    present()
  }
  audio_tone(0, 0, 0, 3, 0, 220, 0, 5, 0.01)
}
```

This uses a different function, `audio_tone`, which creates a **continuous** sound, a tone that keeps going until you silence it. It has nine arguments:

```
FRAGMENT
audio_tone(slot, x, y, z, w, freq, amp, lfo_rate, lfo_depth)
```

- `slot` is the number of the tone. Calling with the same slot again *retunes* the same tone instead of starting another, so calling it every frame is the way to make a sound that follows your mouse.
- `x, y, z` places the sound in 3D space, relative to the camera, so the sound pans left and right and gets quieter if it is far away. We put it directly ahead at `z = 3`.
- `w` is an extra "fourth dimension" parameter for strange timbres. Zero turns it off.
- `freq` is the frequency in hertz and `amp` the loudness.
- `lfo_rate` and `lfo_depth` add a gentle vibrato: a wobble in the pitch, `lfo_rate` times a second, of a size `lfo_depth` as a fraction of the pitch.

The pitch formula is worth a close look: `110 * pow(2, mouse_x() / 640 * 3)`. At the left edge `mouse_x()` is 0, so `pow(2, 0)` is 1 and the frequency is 110 Hz. At the right edge, it is `pow(2, 3) = 8`, so 880 Hz. Each doubling is an octave, so the window spans three octaves. Because the exponent changes smoothly with the pointer, equal mouse movements make equal *musical* steps. That is how real instruments work.

There is no way to delete a tone, so when the loop ends, the program sets the amplitude to 0 to silence it.

> WARNING
>
> Keep the volume low while you experiment. `audio_volume(0.5)` sets the master volume. A continuous tone at full volume through headphones is unpleasant.

Composing with lists

You already know how to store a sequence of numbers: a list. A melody is exactly that: a list of **pitches** and a matching list of **durations**. We measure durations in **beats**, not seconds, because beats are independent of the tempo. A tempo of 120 beats per minute means each beat lasts half a second; a tempo of 60 means one second. Change the tempo, and the same tune plays faster or slower, just as in a score.

This is the "parallel lists" idea from Chapter 5 doing musical work.

Rendering a song to a file

Ling includes the engine of a full music studio, called a **DAW** (digital audio workstation), which you can script with the `daw_*` functions. You build a *project* with *tracks*, put *clips* on the tracks, and put *notes* in the clips. Then you **bounce** the project, which means rendering it to an ordinary sound file. All this happens in memory; it needs no sound card, and no recordings.

Here is the whole thing for "Twinkle, Twinkle, Little Star":

```
# Compose a melody as data, render it to a sound file, then analyse the file.
# No sound files are needed: everything is made from numbers.

bind start = do {
  daw_new()
  daw_set_bpm(120)
  bind lead = daw_track_add("synth")
  bind clip = daw_clip_add_midi(lead, 0, 16)

  # "Twinkle, Twinkle": MIDI pitches and how long each lasts, in beats
  bind pitches = [60, 60, 67, 67, 69, 69, 67, 65, 65, 64, 64, 62, 62, 60]
  bind lengths = [1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 2]

  bind when = 0
  bind i = 0
  while i < len(pitches) {
    bind p = list_get(pitches, i)
    bind d = list_get(lengths, i)
    daw_note_add(lead, clip, when, d * 0.9, p, 0.8)
    bind when = when + d
    bind i = i + 1
  }
  println("notes added: ", len(pitches))
  println("song length: ", when, " beats")

  bind ok = daw_bounce("twinkle.wav", 0, when, 0.5, 16, "wav", 0.5)
  println("rendered: ", ok)

  bind track = music_load("twinkle.wav")
  println("loaded track number ", track)
  println("lasts about ", round(music_duration(track)), " seconds")
  println("detected key: ", music_key(track))
  println("onsets found: ", len(music_onsets(track)))
}
```

```
OUTPUT
notes added: 14
song length: 16 beats
rendered: 1
loaded track number 0
lasts about 9 seconds
detected key: C# minor
onsets found: 13
```

Let us walk through it.

1. `daw_new()` starts an empty project. `daw_set_bpm(120)` sets 120 beats per minute.
2. `daw_track_add("synth")` adds a synthesizer track and returns its number. Other kinds include `"drums"`, `"pluck"` and `"chip"`.
3. `daw_clip_add_midi(lead, 0, 16)` puts an empty **clip** on the track, starting at beat 0 and lasting 16 beats. A clip is a container for notes.
4. The `while` loop adds each note. `daw_note_add(track, clip, start, length, key, velocity)` takes the start time and length in beats, the MIDI note number, and a velocity (how hard the key was hit, from 0 to 1). We keep a running `when` counter; each note starts when the previous one ended. We make notes 90 percent as long as their slot (`d * 0.9`), leaving a tiny gap so repeated notes do not blur together.
5. `daw_bounce` renders beats 0 to `when` to `twinkle.wav`. The `0.5` is a tail of half a second so that the last note can fade out; `16, "wav"` are the format; the final number only matters for compressed formats. It returns `1` for success.
6. The finished file is an ordinary WAV you can open in any media player. It appears in the folder where you ran the program.

Then something delightful happens: the program **listens to its own work**. `music_load` reads a sound file and gives it a handle. `music_duration` tells us the length in seconds (a 16-beat song at 120 BPM is 8 seconds, plus our 0.5 second tail), `music_key` estimates the musical key, and `music_onsets` finds the moments at which new notes begin.

> **NOTE**
>
> The analysis is an *estimate*. We composed in C major, but the analyser guessed C sharp minor: a close neighbour, since the C major tune only uses a few notes. A short melody is not much to go on. It found 13 onsets although we wrote 14 notes: when two identical notes follow one another, the analyser may not notice a new attack. Use these analysis tools for fun and for effects, not for proof.

Drums

Drums are a track of the kind `"drums"`. Instead of pitches, the note numbers choose which drum is hit: **36** is the bass drum, **38** the snare, **42** the closed hi-hat. Rhythm is about *where* in the bar the hits fall:

```
# A two-bar drum loop: kick on the beat, snare on 2 and 4, hi-hat on the off-beats.
bind start = do {
  daw_new()
  daw_set_bpm(100)
  bind drums = daw_track_add("drums")
  bind clip = daw_clip_add_midi(drums, 0, 8)

  bind beat = 0
  while beat < 8 {
    daw_note_add(drums, clip, beat, 0.25, 36, 0.9)
    if beat % 2 == 1 {
      daw_note_add(drums, clip, beat, 0.25, 38, 0.8)
    }
    daw_note_add(drums, clip, beat + 0.5, 0.25, 42, 0.5)
    bind beat = beat + 1
  }

  daw_bounce("drums.wav", 0, 8, 0.0, 16, "wav", 0.5)
  bind t = music_load("drums.wav")
  println("seconds: ", round(music_duration(t)))
  println("tempo: ", round(music_bpm(t)), " BPM")
  println("hits detected: ", len(music_onsets(t)))
}
```

```
OUTPUT
seconds: 5
tempo: 100 BPM
hits detected: 15
```

In every beat we place a kick (36) at the start and a hi-hat (42) halfway through (at `beat + 0.5`). On the odd beats we add a snare (38): that gives the backbeat of rock music. The analyser confidently reports the tempo we asked for. Note how it needs no idea of what drums are: it just notices that loud sounds repeat at a steady rate.

> TIP
>
> To hear what the DAW can do, ask a synth track for its preset names: `daw_inst_presets(track)` returns `[Init Saw, Sub Bass, Additive Organ, FM Bell, Hybrid Pad]` in this version. Pick one with `daw_inst_preset(track, "FM Bell")`.

Hearing your own song

The WAV file can be played with the `music_*` player: `music_load` it, then `music_play(track)`. While it plays, `music_pos()` tells you the position in seconds, so you can draw a progress bar. Here is a bass line and a bell melody, rendered and then played back:

GUI

```
# Make a song, save it as a WAV file, then play that file back with the music player.
bind start = do {
  daw_new()
  daw_set_bpm(110)
  bind bass = daw_track_add("synth")
  daw_inst_preset(bass, "Sub Bass")
  bind bclip = daw_clip_add_midi(bass, 0, 8)
  bind roots = [45, 41, 48, 43]
  bind i = 0
  while i < 4 {
    daw_note_add(bass, bclip, i * 2, 1.8, list_get(roots, i), 0.9)
    bind i = i + 1
  }

  bind lead = daw_track_add("synth")
  daw_inst_preset(lead, "FM Bell")
  bind lclip = daw_clip_add_midi(lead, 0, 8)
  bind tune = [72, 76, 79, 76, 72, 77, 81, 77, 76, 79, 83, 79, 74, 77, 79, 71]
  bind i = 0
  while i < len(tune) {
    daw_note_add(lead, lclip, i * 0.5, 0.45, list_get(tune, i), 0.7)
    bind i = i + 1
  }

  daw_bounce("song.wav", 0, 8, 1.0, 16, "wav", 0.5)
  bind track = music_load("song.wav")
  println("song length: ", round(music_duration(track)), " seconds")

  open_window(480, 200, "Now playing")
  music_play(track)
  while window_is_open() {
    fill(15, 25, 40)
    set_color(120, 255, 190)
    ui_text(20, 30, 26, "PLAYING SONG.WAV")
    bind frac = music_pos() / music_duration(track)
    grad_rect(20, 120, 440 * clamp(frac, 0, 1), 20, 120, 255, 190, 60, 160, 255, 0)
    present()
```

```
    if music_pos() >= music_duration(track) - 0.1 {
      quit()
    }
  }
}
```

```
OUTPUT
song length: 5 seconds
```

Several things are new here:

- Two tracks, each with a **preset** (`"Sub Bass"` and `"FM Bell"`) which changes the sound of the synthesizer.
- The bass notes `45, 41, 48, 43` are the chord roots A, F, C and G, each held for two beats; the bell plays eight notes per bar above.
- After rendering, the program opens a window and plays the finished file. The progress bar's width is `music_pos() / music_duration(track)`, the fraction played so far, multiplied by the full bar width. When the song is almost over, `quit()` closes the window.

> WARNING
>
> `music_play(track)` always restarts the track from the beginning, so do not call it every frame. There is `music_pause()` to pause (it keeps the position) and `music_stop()` to stop (it rewinds to the start). If you want to resume a paused track, call `music_play(-1)`.

> NOTE
>
> The DAW also has `daw_play()`, which plays the project live without bouncing first. It is audible, but it is meant for tools that edit music, and it is best avoided in short test scripts. Bouncing a file is simpler and does not depend on a sound card, which is why we use it throughout this chapter.

A tiny melody player

Last, a melody player that does not render anything. It plays each note at the right moment, using the clock. The melody is two lists at the top of the file: `pitches` (with `0` meaning a rest) and `beats`. It is "Ode to Joy".

GUI

```
# A tiny melody player. The tune is just two lists: pitches and durations.
# 0 means "rest". It plays once, shows each note as it sounds, then quits.

bind pitches = [64, 64, 65, 67, 67, 65, 64, 62, 60, 60, 62, 64, 64, 62, 62, 0]
bind beats   = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1.5, 0.5, 2, 1]
bind tempo = 132          # beats per minute

bind start = do {
  open_window(640, 300, "Melody player")
  bind spb = 60 / tempo                 # seconds per beat
  bind begun = time_now()
  bind next_at = 0                      # when the next note is due (seconds)
  bind index = 0
  bind shown = "..."

  while window_is_open() {
    bind now = time_now() - begun
    if index < len(pitches) && now >= next_at {
      bind p = list_get(pitches, index)
      bind d = list_get(beats, index) * spb
      if p > 0 {
        audio_blip(music_hz(p), d * 0.9, "triangle", 0.3)
        bind shown = music_note_name(music_hz(p))
      } else {
        bind shown = "rest"
      }
      bind next_at = next_at + d
      bind index = index + 1
    }
    if index >= len(pitches) && now > next_at + 0.5 {
      quit()
    }

    fill(25, 15, 45)
    set_color(255, 200, 90)
    ui_text(40, 70, 90, shown)
    set_color(120, 120, 180)
```

```
    ui_text(40, 200, 20, "NOTE " + to_str(index) + " OF " + to_str(len(pitches)))
    present()
  }
  println("Played ", index, " notes")
}
```

```
OUTPUT
Played 16 notes
```

The important idea is the **scheduler**. Each frame we compute `now`, the seconds since the start. Whenever `now` reaches `next_at`, the time at which the next note is due, we play that note and push `next_at` forward by the note's duration. In this way the music's rhythm comes from the clock, not from the frame rate: if the computer is slower or faster, the tune keeps the correct tempo. (The notes are only started on frame boundaries, so each can be late by a fraction of a frame, about a sixtieth of a second. That is quite good enough for a tune; a precise sequencer would use the DAW.)

`spb` is "seconds per beat", `60 / tempo`. The tune plays at 132 beats per minute. Change `tempo` to 90 for a calmer performance, or 180 for a frantic one.

At the end the program prints how many notes it played. As the last note is played, a half second of silence passes before the window closes, so the final note is not cut off.

> **TRY IT**
>
> Replace the melody with your own tune. Write the notes of "Happy Birthday" (it starts with 60, 60, 62, 60, 65, 64) with matching beats, and put the tempo at 120. A rest between phrases is a `0` in the pitches list.

Chapter summary

- Sound is vibration; the **frequency** in hertz sets the pitch. A4 is 440 Hz; an octave doubles the frequency.
- MIDI numbers number the piano keys: 60 is middle C, 69 is A4. `music_hz(pitch)` and `music_note_name(hz)` convert between numbers, names and frequencies.
- `audio_blip(freq, seconds, wave, volume)` plays a short sound; waveforms are `"sine"`, `"triangle"`, `"square"`, `"saw"` and `"noise"`.
- `audio_tone(slot, x, y, z, w, freq, amp, lfo_rate, lfo_depth)` is a continuous tone, retuned by calling it again with the same slot; set the amplitude to 0 to silence it.

- A melody is two parallel lists: pitches and durations in beats.
- The DAW builtins build a project: `daw_new`, `daw_set_bpm`, `daw_track_add`, `daw_clip_add_midi`, `daw_note_add`, and `daw_bounce` to render a WAV file without any sound card or recordings.
- `music_load` reads the file back; `music_duration`, `music_key`, `music_bpm` and `music_onsets` analyse it; `music_play`, `music_pause`, `music_stop` and `music_pos` play and track it.
- Drum notes select drums: 36 kick, 38 snare, 42 closed hi-hat.
- To play notes on time, compare the clock (`time_now()`) with the time the next note is due.
- Analysis results are estimates.

Exercises

1. Use `music_hz` to print the frequency of every note from middle C (60) up to the C an octave above (72), twelve steps in all. What do you notice about the first and last frequencies?
2. Change the keyboard piano so that pressing the key `"z"` plays a drum-like sound using `audio_blip(120, 0.15, "noise", 0.4)`.
3. Change the theremin so that vertical position controls the vibrato depth (the last parameter) instead of the loudness. Keep the loudness constant at 0.15.
4. Write a program that composes a four-beat drum pattern with a kick on every beat, bounces it to `mydrums.wav`, loads it, and prints the number of onsets detected.
5. Add a second voice to the melody player: play each note a fifth higher (that is 7 MIDI steps) at the same time as the main note, with a lower volume. Hint: one more `audio_blip` inside the same `if`.

Answers

1. A loop from 60 to 72 calling `println(m, " ", music_hz(m))` gives 261.6 and about 523.3. The last frequency is exactly double the first: that is what an octave is.
2. Add `if key_pressed("z") { audio_blip(120, 0.15, "noise", 0.4) }` inside the frame loop, before drawing.
3. Replace the last argument of `audio_tone` with `mouse_y() / 400 * 0.05` and use `0.15` for `amp`.
4. Reuse the drum program above with `daw_set_bpm(120)`, a clip of 4 beats and a single `daw_note_add(drums, clip, beat, 0.25, 36, 0.9)` per beat. Bounce with `daw_bounce("mydrums.wav", 0, 4, 0.0, 16, "wav", 0.5)` and print `len(music_onsets(t))`. The count may be slightly different from 4: the analyser is an estimate.

5. Inside `if p > 0 { ... }`, after the existing `audio_blip`, add `audio_blip(music_hz(p + 7), d * 0.9, "triangle", 0.15)`.

CHAPTER 9

Your First Project

What you'll learn. Until now each program has been a single file. Real programs grow, so in this last beginner chapter you will learn to create a proper project with `lingfu`, split a program across several files with `use`, and package it so friends can run it. You will also learn how to hunt down mistakes, including the sneaky ones that Ling does not report by default, and you will get a troubleshooting checklist to keep.

From a file to a project

A single `.ling` file is perfect for experiments. But a game with graphics, sound, levels and fonts has many files, and you need somewhere to keep them together, along with a *description* of the whole thing: its name, its version, where it starts.

That is what a **project** is: a folder with a small description file, called the **manifest**, and your source files. The tool for managing projects is `lingfu` (灵符, "spirit token"). You installed it together with `ling` in Chapter 1.

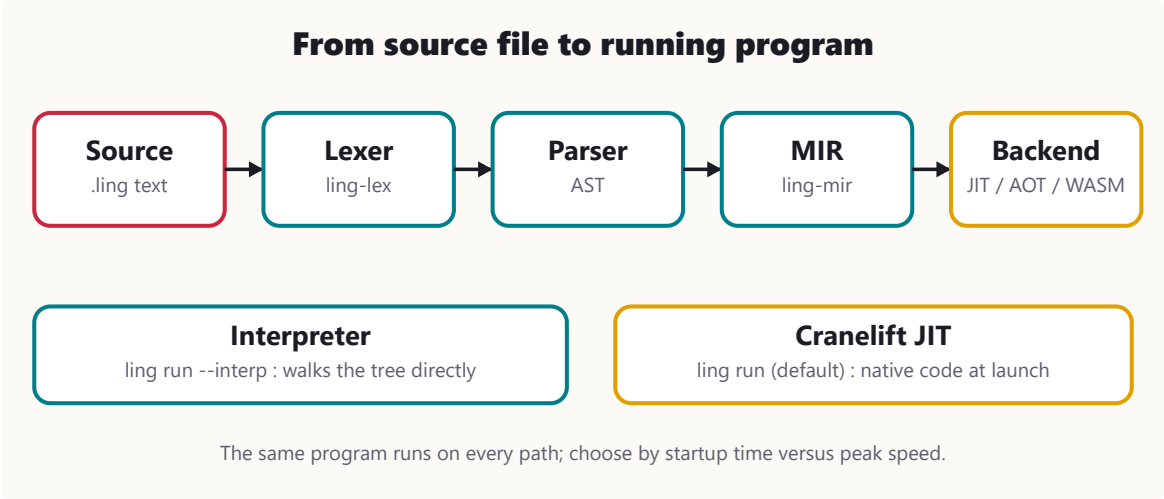


Figure: From source to program. Your files are read by ling, which runs them directly, or packages them for other platforms.

Creating a project

Open a terminal in the folder where you keep your work and type:

```
OUTPUT
$ lingfu new demo
Creating project: demo

Spirit Realm created!
  cd demo
  lingfu build
  lingfu run
```

(The real tool decorates its messages with small symbols; we have left them out.) `lingfu` has created a folder named `demo` containing:

```
OUTPUT
demo/
  Ling.toml          the manifest
  src/start.ling     your program starts here
  tests/test.ling    a starter test
  lexicons/en.ling   language data
  README.md
  .gitignore
```

Let us run it:

```
OUTPUT
$ cd demo
$ lingfu run
Running .../demo/src/start.ling...
Welcome to demo Spirit Realm
Spirit Level: 铁
```

Three words in a row, `lingfu`, `run`, and nothing about files. `lingfu run` finds the manifest, finds the entry program, and runs it with `ling run` underneath. You can still run any file directly with `ling run src/start.ling`: both paths end up in the same place.

> NOTE
>
> `lingfu new` can also create projects in other human languages: `lingfu new demo --lang zh` makes a project whose folders and files have Chinese names (灵符.toml , 灵源/启.灵). Supported choices are `en` , `zh` , `ja` , `ko` , `th` and `ru` .

The manifest

Open `Ling.toml` in your editor:

```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" }
```

TOML is a simple configuration format: `[sections]` in square brackets, and `key = value` lines. The `[spirit]` section holds what matters most: the `name` of the project, its `kind` (`bin` is an ordinary program; others include `game` , `ui` and `lib`), and its `version` .

The `entry =` gotcha

There is one thing in the generated manifest that will trip you up when you get to packaging. The program is in `src/start.ling` , but the packaging command `lingfu build` (which uses `ling build` internally) assumes by default that your program is called `main.ling` in the project's top folder. If you try to build the project as it was generated, you get this:

```
OUTPUT
$ lingfu build
Building Ling project...
Entry: .../demo/src/start.ling
-> AOT-compiling to a native executable...
error: entry '.../demo/main.ling' not found
Error: build failed
```

The fix is one line. Tell the builder where the program starts, in the `[spirit]` section:

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

We tried this and the entry error is gone. (`lingfu run` and `lingfu test` do not need the line, because they search `src/start.ling` themselves. That is why the problem only shows up at the last step.)

> **WARNING**
>
> The part of Ling that packages programs reads the manifest line by line instead of with a full TOML reader. Write every setting on **one line**, in the form `key = value` with a space before the `=`. Lists, like `targets = ["win", "lin", "web"]`, must also be on a single line.

Splitting a program across files

A program with five hundred lines in one file is hard to read. Split it into files, each with a job. The keyword that connects files is `use`.

Our example is a temperature converter. The two conversion functions go in their own file, as a *library*: a file that holds functions but has no `bind start`:

FRAGMENT

```
# convert.ling — a library file: functions only, no start.
fn c_to_f(c) {
  return c * 9 / 5 + 32
}

fn f_to_c(f) {
  return (f - 32) * 5 / 9
}
```

The main program, which sits in the same folder, borrows them:

```
# main.ling — the program. It borrows two functions from convert.ling.
use "convert.ling" as conv

bind start = do {
  println("0 C = ", conv::c_to_f(0), " F")
  println("100 C = ", conv::c_to_f(100), " F")
  println("-40 F = ", conv::f_to_c(-40), " C")
}
```

OUTPUT

```
0 C = 32 F
100 C = 212 F
-40 F = -40 C
```

`use "convert.ling" as conv` loads the other file, *relative to the file that says* `use`, and gives it the nickname `conv`. After that, `conv::c_to_f(0)` calls the function `c_to_f` from that file. The double colon `::` reads as "inside". Nicknames prevent confusion if two files happen to contain functions with the same name.

You may leave out the `as`: then the functions are called by their plain names, and you may also leave off `.ling`:

```
FRAGMENT
use "convert"

bind start = do {
  println(c_to_f(37))
}
```

which prints `98.6`. Both forms work in a plain `ling run main.ling` and also inside a `lingfu` project, where you would place `convert.ling` in the `src` folder next to `start.ling`. (We tried it both ways.)

> WARNING
> Share **functions** between files, not values. A top-level `bind` in a library file (such as `bind tax = 0.2`) is *not* usable from the importing file, and not even from the library's own functions, so such programs misbehave (the default engine gives you `()`; the interpreter gives you an "undefined" error). If you need a shared value, write a tiny function that returns it: `fn tax() { return 0.2 }`.

How to divide a program

A useful way to decide what goes where:

- **One file per idea.** `physics.ling`, `drawing.ling`, `levels.ling`.
- **Small files.** If you scroll more than a couple of screens, it is time to split.
- **The entry file is short.** Its `start` calls the pieces in order. Someone opening it should be able to understand the whole program in a minute.
- **Name by what it does,** not how: `scoring.ling`, not `stuff2.ling`.

Testing

A **test** is a small program that checks that something works. `lingfu test` runs the file `tests/test.ling` (and any other `.ling` file in `tests/`) and reports success if they all run without failing:

```
OUTPUT
$ lingfu test
Testing Ling project...
  test: .../demo/tests/test.ling
All mirrors are clear
Ling tests passed for demo
All tests passed!
```

A test is an ordinary program: it needs a `bind test = do { ... }` (any `bind` of a `do` block serves as the entry). It "fails" if it stops with a runtime error. Ling has no assertion function, so to make a test that fails on purpose, call a function that fails under the interpreter. That is a bit awkward, so for now you can think of tests as a safety net for "does this program still run to the end without crashing?".

Shipping your program

You have made something. How do you give it to a friend who does not have Ling? You **build** it: the build process packages your program, and the Ling runtime that drives it, into something a person can double-click.

```
OUTPUT
$ ling build main.ling --platform win
$ ling build main.ling --platform web,lin,mac
$ lingfu build --platform all
```

The `--platform` option chooses the target systems: `win` (Windows), `lin` (Linux), `mac` (macOS), `web` (a page for a web browser), or `all` for the four of them. The results are written to a folder called `dist`, with one sub-folder per platform. Some other useful options:

Option	What it does
`--out folder`	put the results somewhere other than `dist`
`--icon file`	use your own icon (SVG, PNG or ICO) for the application
`--pack`	embed pictures, fonts and sounds *inside* the program, giving a single file
`--aot`	compile to native machine code instead of embedding the interpreter

If your program uses files such as pictures or sound, list them in the manifest so that they travel along. Since the packager reads the manifest line by line, a list of patterns goes on **one line**:

```
includes = ["music/*.wav", "font/*.otf"]
```

> WARNING
>
> Native builds create a small Rust project behind the scenes and compile it, so you must have a **Rust toolchain** installed (the installer from Chapter 1 sets one up), and the first build can take several minutes. The command-line options and the manifest rules in this section come from the documented behaviour and the program's source. In our test environment we confirmed that the entry-file problem above is real and that the fix works, but the compile step itself stopped with a linker error that came from the sandbox machine we used, so **we could not produce a finished executable ourselves.** If your build fails on a missing file or a linker message, check that Rust is correctly installed first.

Finding mistakes (debugging)

Every programmer spends a large part of their time hunting for the reason a program does something unexpected. The hunt is a skill, and it can be learned. Here are the methods, from the simplest.

1. Print what you think you know

The oldest trick is to add `println` lines that show the values at the point where things go wrong:

```
FRAGMENT
println("DEBUG x = ", x, "  y = ", y)
```

Run the program and compare what is printed with what you expected. The first place where reality differs from your expectation is where the bug lives. Remove the debug lines when you are done, or comment them out with `#`.

2. Run it on the interpreter

This is the most useful Ling-specific advice in the book. Ling has two engines that run your programs:

- The default engine, the **JIT** (just-in-time compiler), is **fast**. It translates your program into machine code while it runs.
- The **interpreter** reads your program step by step. It is slower but **careful**: it reports many mistakes that the JIT silently lets pass.

You select the interpreter with `--interp`:

OUTPUT

```
$ ling run --interp myprogram.ling
```

Here is the best example. A misspelled function name:

```
fn area(w, h) {
  return w * h
}

bind start = do {
  bind a = area(3, 4)
  println("area: ", a)
  bind total = aera(5, 6) + a
  println("total: ", total)
}
```

OUTPUT

```
area: 12
total: ()
```

The program above has a typo: `aera` instead of `area`. On the default engine, the failed call does not stop the program. The call simply produces `()`, the "nothing" value, which spreads through the arithmetic, so the output says `total: ()`, and no error is shown. (We have shown the real output above.) The interpreter tells you exactly what is wrong:

```
OUTPUT
$ ling run --interp dbg1.ling
area: 12
error[runtime]: unknown function 'aera'
 --> dbg1.ling
  traceback (deepest call last):
    0: start
  hint: check the spelling, or `use` the module that defines it
```

> **TIP**
>
> Make `--interp` part of your routine. Whenever a program prints `()` where it should print something, or a sound is missing, or a function seems to do nothing, run it once with `--interp` before doing anything else. Many bugs are found in ten seconds this way.

The silent-failure rule applies to every built-in function too. Misspell `draw_disc` as `draw_dsic` in a graphics program and the shape simply never appears. The JIT also lets other things pass: for example a division by zero gives `()` rather than an error. And as you saw in Chapter 5, the two engines sometimes disagree about lists. If a program works on one engine and not on the other, your program has a bug; the interpreter is the reference for what Ling *means*.

3. Make it smaller

When a program of two hundred lines fails, copy it, then delete parts until the failure disappears. The last thing you deleted is the culprit. It is like finding which light bulb in a string is broken.

4. Explain it out loud

Describe your code, line by line, to a friend, a pet or a rubber duck. This is such a well-known method it has a name:

rubber duck debugging

. Nine times out of ten you find the mistake while explaining.

5. Read the error message, calmly

Ling's messages are short. They say what kind of error it was (`parse` or `runtime`), the file, and often a hint. For a parse error, look at the line *before* the one reported: a missing bracket or quote is noticed late.

Troubleshooting checklist

When something is wrong, work through the list from top to bottom:

Symptom	Likely cause and fix
`ling` is "not found"	Close and reopen the terminal; check that the folder `.cargo/bin` is on your `PATH`.
`error[parse]: unexpected token in expression: Eq`	You wrote an assignment like `x = x + 1`. Write `bind x = x + 1`.
`error[parse]` and the line looks fine	A missing `)`, `}` or closing quote earlier in the file.
`no entry point`	The file has no `bind start = do { ... }` (or you used `fn main()`).
Output shows `()`	A misspelled function, a missing value, or `1/0`. Run with `--interp`.
Text runs together: `sum3`	`println` adds no separators; put spaces in your text.
A loop total stays at 0	You accumulated inside a `for`. Use `while`.
A `for` over `0..len(xs)` does nothing	Range ends need parentheses: `0..(len(xs))`.
A list changed when you did not expect	Always rebind: `bind xs = list_set(xs, ...)`; copy lists you need to keep.
A window opens and shows nothing	You forgot `present()`, or you cleared after drawing.
A 3D shape is upside-down or missing	Remember y points down and z points away; check signs.
A graphics call does nothing	Misspelled name. Unknown builtins are silently ignored.
Colours look washed out and cyan	The default stylised shading; use `set_shade_mode(1)` and `set_rim(0, 0, 0, 0)`.
A number like `1.5e3` misbehaves	There is no scientific notation. Write `1500`.
No sound	Check the system volume; try `audio_blip(440, 0.5, "sine", 0.3)` alone.
`lingfu build` says `entry ... main.ling not found`	Add `entry = "src/start.ling"` to the manifest.
The window will not close	Press Escape, or close it with the window button; in a terminal press Ctrl+C.
A shared value from `use` is `()`	Share functions, not top-level values.

NOTE

Ling is a young language, and it has rough edges. A good deal of this book is *about* those rough edges so that they do not surprise you. When something behaves oddly, you may not have made a mistake. Check the Reference, which lists verified behaviour, and compare the two engines.

Where to go next

You have completed the Beginner edition. You can write programs with values, decisions, loops, functions, lists and text. You can draw in 2D and 3D, handle input, build a small game, make sound, and organize and ship a project. That is a great deal, and the same tools keep working as projects grow.

The next steps depend on what you enjoy:

- **Intermediate edition.** Data in depth (your own record types with `form` and `choose`, error handling with `ok` and `bad`), working with files, modules and bigger projects, user-interface applications, more game techniques (physics, collision, sprites, animation, dialogue boxes), packaging and a collection of cooking-book style recipes.
- **Advanced edition.** Concurrency (threads and channels), networking and web servers, cryptography, AI and language-model features, compiler internals, and embedding Ling in Rust programs.
- **The Reference** is always useful. Every built-in function has an entry with its parameters and a verified example.

Meanwhile, here are four project ideas of exactly the right size to practise on:

1. A **quiz game** that asks questions from a list and keeps the score.
2. A **3D gallery**: a room with several frames on the walls (use the `vtex` patterns as paintings) and a camera that you can walk around.
3. A **drum machine**: a grid of keys that toggles drum hits, bounced to a WAV file.
4. A **dodge game**: like Star Catcher, but the player moves in two directions and avoids falling hazards.

TRY IT

Pick one idea and write a plan before you write code: list the state, the rules and the picture (as we did for Star Catcher). Then build the smallest version that works, run it, and add one feature at a time. That is how programs grow, and it is how everyone learns.

Chapter summary

- A project is a folder with a manifest (`Ling.toml`) and sources. `lingfu new name` creates one; `lingfu run` and `lingfu test` run it.
- **Gotcha:** the generated project keeps its program in `src/start.ling`, so add `entry = "src/start.ling"` to `[spirit]` before using `lingfu build`.
- Manifest settings are read line by line; keep each on one line, with a space before `=`.
- Use `use "file.ling" as name` to split code into files; call `name::function(...)`. Share functions, not top-level values.
- `ling build file.ling --platform win,lin,mac,web` (or `all`) packages a program into `dist`; `--pack` embeds resources. You need a Rust toolchain.
- **Debugging:** print values; run with `ling run --interp` to expose silent mistakes such as misspelled function names; shrink the program; explain it aloud.
- The default engine silently ignores unknown functions (returning `()`); the interpreter reports them.
- Keep a troubleshooting checklist and use the Reference.

Exercises

1. Create a project with `lingfu new myfirst`. Change its `start.ling` to print your name, and run it with `lingfu run`.
2. In that project, add a library file `src/greetings.ling` with a function `hello(name)` that returns `"Hello, " + name`, and use it from `start.ling` with `use "greetings.ling" as g`.
3. Write a program that deliberately misspells a function name. Run it with both engines and write down the difference.
4. Add `entry = "src/start.ling"` to your manifest. What would have happened without it when you ran `lingfu build`?
5. Pick one of the bugs in the troubleshooting table and write a tiny program that shows it. Then fix the program.

Answers

1. `lingfu run` prints whatever your `println` shows; the project creates `src/start.ling`, which contains `bind start = do { ... }`.
2. `greetings.ling`: `fn hello(name) { return "Hello, " + name }`. `start.ling`: `use "greetings.ling" as g` on the first line, and in `start`, `println(g::hello("Mina"))`. The output is `Hello, Mina`.

3. For example `println(sqrt_of(9))`. The default engine prints `()`; the interpreter stops with `unknown function 'sqrt_of'`.
4. `lingfu build` would have stopped with `entry '.../main.ling' not found`, because the builder looks for `main.ling` by default.
5. For example the accumulate-in-`for` bug: `bind total = 0`, `for i in 1..5 { bind total = total + i }`, `println(total)` prints `0`. The fix is the same loop written with `while`.

PART

Reference

Every keyword and the essential builtins, with examples.

CHAPTER 10

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 后	起動	생성	สร้าง	Reserved word only
`ok`	好 可	良い	좋아	ตกลง	Active
`bad`	坏 误	悪い	나쁨	ผิดพลาด	Active
`none`	无	なし	없음	ไม่มี	Unreliable
`as`	为	として	로서	เป็น	Active (aliasing)
`where`	条件	但し	단	โดยที่	Skipped
`true` / `false`	真 / 假	真 / 偽	참 / 거짓	จริง / เท็จ	Active (literals)

Keyword reference

`bind`

Introduces a binding: `bind name = expression`. It is the only way to create or change a variable (there is no separate assignment statement), so re-binding an existing name is how you "update" it.

- **Status:** Active
- **Aliases:** zh: `令`, `灵符`; ja: `束縛`, `バ`; ko: `바인드`, `묶`; th: `ผูก` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew, Hindi)
- **Note:** At top level, `bind` also defines global values and the program entry point (`bind start = do { ... }`).
- **Note:** Any keyword may be used as a binding name, e.g. `bind change = 5`.

```
bind start = do {
  bind price = 40
  bind price = price + 2
  println(price)
}
```

```
OUTPUT
42
```

`do`

Block expression: `do { statements }` runs its statements in order and evaluates to the value of the last one. A bare `{ ... }` in expression position is the same thing. The program entry point is a `do` block bound to `start`.

- **Status:** Active
- **Aliases:** zh: `执`; ja: `執`, `実行`; ko: `실행`; th: `ทำ` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew, Hindi)
- **Note:** Under the interpreter, bindings made inside a `do` block are discarded when the block ends. The default JIT backend lets them leak into the enclosing scope. Do not depend on either.

```
bind start = do {
  bind total = do {
    bind a = 20
    bind b = 22
    a + b
  }
  println(total)
}
```

```
OUTPUT
42
```

`fn`

Declares a named function: `fn name(parameters) { body }`. The value of the last expression in the body is the result; `return` exits early. Functions are declared at top level (or inside a `mod`), not inside other functions.

- **Status:** Active
- **Aliases:** zh: `函`; ja: `関`, `関数`; ko: `함수`; th: `ฟังก์ชัน` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)
- **Note:** Parameter and return type annotations (`a: number`, `-> number`) are parsed and ignored. Generic parameters (`<T>`) are skipped.
- **Note:** A function sees top-level `bind` values read-only, but not the local variables of its caller.
- **Note:** `fn` before a `|...|` is not how closures are written; use `|x| expression` (see **Closures** in the syntax chapter).

```
fn area(w, h) { w * h }
fn describe(n) {
  if n > 100 { return "big" }
  "small"
}
bind start = do {
  println(area(6, 7))
  println(describe(500))
  println(describe(5))
}
```

```
OUTPUT
42
big
small
```

`if`

Conditional expression: `if condition { ... }`, optionally followed by `else if ... { }` and `else { }`. It evaluates to the value of the branch taken (`()` when no branch is taken and there is no `else`). Braces are mandatory; the condition needs no parentheses.

- **Status:** Active
- **Aliases:** zh: `若`, `如`; ja: `もし`; ko: `만약`, `조건`; th: `ถ้า` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew, Hindi)
- **Note:** Use real booleans in conditions. Truthiness of non-booleans differs between backends: the JIT treats `""` and `[]` as true, the interpreter as false.
- **Note:** Bindings made inside an `if` body remain visible after it (the body shares the enclosing scope).

```
bind start = do {
  bind n = 15
  bind kind = if n % 15 == 0 { "fizzbuzz" } else if n % 3 == 0 { "fizz" } else {
"other" }
  println(kind)
}
```

```
OUTPUT
fizzbuzz
```

`else`

Introduces the alternative branch of an `if`, or chains with another `if` as `else if`.

- **Status:** Active
- **Aliases:** zh: `否则`, `否`; ja: `他`; ko: `아니면`; th: `มิฉะนั้น` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew, Hindi)

```
bind start = do {
  bind t = 3
  if t > 5 { println("hot") } else { println("mild") }
}
```

```
OUTPUT
mild
```

`while`

Loop while a condition holds: `while condition { body }`. The body runs in the **enclosing** scope, so `bind counter = counter + 1` inside the loop updates the outer `counter`. `while` is therefore the loop to use when you need to accumulate a result.

- **Status:** Active
- **Aliases:** zh: `循`, `当`, `一方`; ja: `間`; ko: `동안`, `반복`; th: `ขณะที่` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew, Hindi)
- **Note:** There is no working `break` or `continue` (see `stop` and `again`). To leave early, make the condition false or `return` from a function.

```
bind start = do {
  bind i = 1
  bind product = 1
  while i <= 5 {
    bind product = product * i
    bind i = i + 1
  }
  println(product)
}
```

```
OUTPUT
120
```

`for`

Iterates over a range, list or string: `for name in iterable { body }`. Ranges are written `lo..hi` and exclude `hi`. Strings iterate by character under the interpreter.

- **Status:** Active
- **Aliases:** zh: `历`; ja: `繰`, `ために`; ko: `위해`; th: `สำหรับ` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew, Hindi)
- **Note: Each iteration gets a fresh copy of the scope**: `bind total = total + i` inside a `for` body does not change the outer `total`. Use `while` to accumulate, or build results with `list_push` on the JIT backend.
- **Note:** Iterating a string yields empty values under the default JIT backend; iterate a list or a range instead.

```
bind start = do {
  for i in 1..4 { println(i * i) }
  for name in ["ada", "lin"] { println(str_upper(name)) }
}
```

```
OUTPUT
1
4
9
ADA
LIN
```

`in`

Separates the loop variable from the sequence in a `for` statement.

- **Status:** Active
- **Aliases:** zh: `于`; ja: `の中`; ko: `안에`; th: `ใน` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)

```
bind start = do {
  for x in [10, 20, 30] { println(x) }
}
```

```
OUTPUT
10
20
30
```

`match`

Pattern matching: `match value { pattern => expression, ... }`. The first arm whose pattern fits is evaluated. Patterns: `_` (anything), a literal (number, string, `true`/`false`), a bare name (matches anything and binds it), `ok(p)` / `bad(p)`, and enum variants `Variant(p1, p2)` (a payload-less variant is written `Variant()`).

- **Status:** Active

- **Aliases:** zh: `配`; ja: `一致`; ko: `매치`; th: `จับคู่` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew)
- **Note:** If no arm matches, the whole `match` evaluates to `()`; there is no exhaustiveness check.
- **Note:** There are no guards (`if` conditions) and no patterns for lists, ranges or `form` records.

```
choose Shape { Circle(r), Rect(w, h), Dot }
fn area(s) {
  match s {
    Circle(r) => 3 * r * r,
    Rect(w, h) => w * h,
    Dot() => 0
  }
}
bind start = do {
  println(area(Circle(2)))
  println(area(Rect(3, 4)))
  println(area(Dot()))
  println(match 7 { 1 => "one", 7 => "seven", _ => "other" })
}
```

```
OUTPUT
12
12
0
seven
```

`return`

Leaves the current function immediately with a value: `return expression` (or bare `return` for `()`).

- **Status:** Active
- **Aliases:** zh: `归`; ja: `戻る`, `帰る`; ko: `반환`, `귀환`; th: `คืน` (further spellings exist in: French/German/Spanish/Portuguese, Russian, Arabic/Persian/Urdu, Hebrew, Hindi)

- **Note:** A function without `return` yields its last expression. A `return` directly inside `start` ends the program silently on the JIT backend and is a runtime error (`unexpected top-level return`) on the interpreter.

```
fn first_even(xs) {
  for x in xs {
    if x % 2 == 0 { return x }
  }
  return 0 - 1
}
bind start = do {
  println(first_even([3, 5, 8, 9]))
  println(first_even([1, 3]))
}
```

```
OUTPUT
8
-1
```

`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
```

`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
```

`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 11

Syntax, Types and Semantics

This chapter describes the Ling language as it actually behaves. Everything here was checked against the lexer and parser in `src/` and by running programs. Where the README or the glossary promise something the implementation does not do, the difference is called out.

Ling currently has two execution engines. `ling run file.ling` uses the Cranelift JIT backend by default; `ling run --interp file.ling` uses the tree-walking interpreter, which is the semantic reference. They agree on well-behaved programs but differ in several corners, summarised in the last section of this chapter. Examples are run on both engines and any difference is shown.

Program structure

A Ling file is a sequence of **items** at top level: `bind`, `fn`, `mod`, `form`, `choose`, `use`, `type`. Statements such as `println(...)` are **not** allowed at top level; they must live inside a function or the entry point.

The entry point

Execution starts at a top-level binding whose value is a `do` block. The conventional spelling is:

```
OUTPUT
bind start = do {
  ...
}
```

The runtime looks for a binding named `start`, `main`, `启`, `เริ่ม`, `시작`, `начать`, `начало`, `inicio`, `comenzar`, `début`, `commencer`, `démarrer`, `anfang`, `starten`, `início`, `শুরু`, `شروع`, `ابدأ` or `התחל` (checked in that order). If none exists, the first top-level binding whose value is a `do { }` block is used. If there is no such binding the program is a library and `ling run` reports `no entry point`.

`fn main() { ... }`, as shown in some README examples, does **not** work: `main` must be a `bind` of a `do` block. The error is `no entry point — need bind start = do {...}`.

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

```
OUTPUT
Hello, Ling!
```

```
bind main = do {
  println("main works too")
}
```

```
OUTPUT
main works too
```

Comments and whitespace

A comment runs from `//` or `#` to the end of the line. Whitespace, including newlines, separates tokens; statements need no terminator, and an optional `;` may follow any statement. There are no block comments.

```
# a hash comment
// a slash comment
bind start = do {
  println("a") // trailing comment
  println("b"); println("c")
}
```

```
OUTPUT
a
b
c
```

Identifiers

An identifier starts with a letter or underscore (any Unicode letter, including CJK, Thai, Cyrillic, Arabic) and continues with letters, digits and underscores. Identifiers are case-sensitive. Any keyword may be used as an identifier.

```
bind start = do {
  bind 名前 = "Ling"
  bind ชื่อ = 3
  println(名前)
  println(ชื่อ + 1)
}
```

```
OUTPUT
Ling
4
```

Literals

Literal	Form	Notes
Number	`42`, `3.14`	Decimal digits with an optional single fractional part. There is **no** hexadecimal, exponent (`1e3`), digit separator (`1_000`), or leading/trailing-dot (`.5`, `3.`) form. All numbers are 64-bit floats.
Text	`"hello"`	Double quotes only; may span several lines. Escapes: `\n` `\t` `\r` `\"` `\\` `\0`. Any other `\x` is kept literally, backslash included. No interpolation: `"{x}"` is just those characters.
Boolean	`true`, `false`	Also `真` / `假`, `참` / `거짓`, `จริง` / `เท็จ`, ...
Unit	`()`	The "nothing" value; produced by functions with no result. Prints as `()`.
List	`[1, "two", 3.0]`	Heterogeneous, zero-indexed.
Range	`0..5`	Half-open: 0, 1, 2, 3, 4. Used with `for`.
Function value	`|x| x + 1`	Closure; see below.

There is no character literal (`'a'` is a syntax error), no map/dictionary literal and no tuple literal.

```
bind start = do {
  println("line1\nline2")
  println("tab\there")
  println("quote: \"q\" backslash: \\")
  println(1.5 + 1)
  println([1, "two", 3.5, true])
  println(())
}
```

```
OUTPUT
line1
line2
tab	here
quote: "q" backslash: \
2.5
[1, two, 3.5, true]
()
```

Numbers

Because every number is a 64-bit float, integers are exact up to 2^53 and division always produces a float result.

```
bind start = do {
  println(7 / 2)
  println(10 / 5)
  println(7 % 3)
  println(0.1 + 0.2)
  println(1000000 * 1000000)
}
```

```
OUTPUT
3.5
2
1
0.30000000000000004
1000000000000
```

Division by zero is not an error: under the interpreter `1 / 0` is `inf` and `0 / 0` is `NaN`; the JIT backend prints `()` for `1 / 0`.

Types

Ling is dynamically typed. A value is one of:

Type	Examples	Created by
number	`3`, `2.5`	numeric literals, arithmetic
text	`"abc"`	string literals, `+`, string builtins
bool	`true`	literals, comparisons
unit	`()`	functions without a result, `()`
list	`[1, 2]`	list literals, `list_new`, ranges (conceptually)
result	`ok(5)`, `bad("e")`	the `ok` / `bad` constructors
function	`|x| x`	closure expressions
record	`Point { x: 1, y: 2 }` as printed	a `form` name called like a function
variant	`Circle(2)`	a `choose` variant called like a function

There is **no map/dictionary type**, no set, no tuple, no character type and no null: the builtins `list_*` operate on lists, and keys must be emulated with parallel lists. The type names `number`, `text`, `bool`, `list`, `map`, `tuple` in the glossary and lexicon files are documentation only. Type annotations in code (`x: number`, `-> number`) are parsed and **ignored**.

```
fn describe(a: number, b: text) -> text {
  b + to_str(a)
}
bind start = do {
  println(describe(5, "n="))
}
```

```
OUTPUT
n=5
```

Operators

From the tightest to the loosest binding:

Level	Operators	Associativity	Notes
1	`.` `()` `[]` `::` `..`	left	member/method, call, index, path, range
2	`-` (prefix) `!` `&` `wait` `own` `lend` `share` `move` `copy`	prefix	`-x` is `0 - x`; `!x` is `x == false`; `&x` is just `x`; the ownership words are no-ops
3	`*` `/` `%`	left	
4	`+` `-`	left	`+` concatenates text (the other operand is converted to text)
5	`==` `!=` `<` `>` `<=` `>=`	left	`1 < 2 == true` parses as `(1 < 2) == true`
6	`&&`	left	
7	`||` and a single `|`	left	

Notes that matter in practice:

- **Unary minus works** (`-x`, `-3 + 5`, `2 * -x`) but cannot be doubled (`- - x` is a syntax error). Writing `0 - x` is always safe and is what older documentation recommends.
- **The upper bound of a range must be a single name, number or parenthesised expression.** `0..n` and `0..(n + 1)` work; `0..n + 1` is a syntax error, and `0..len(xs)` does not do what it looks like (the call is applied to the range). Write `0..(len(xs))` or bind the length first.
- **`&&` and `||` do not short-circuit** on the interpreter: both operands are always evaluated, so do not rely on `x != 0 && 10 / x > 1` to protect the right-hand side.
- Comparison chains are left-nested: `1 == 1 == 1` is `(1 == 1) == 1`, which is `false`.
- `+` on a number and text converts: `"a" + 1` is `a1` and `1 + "a"` is `1a`. Arithmetic between a number and a boolean is a runtime error on the interpreter.
- `%` on floats works (`7.5 % 2` is `1.5`).
- Text compares with `==`, `!=`; ordering (`<`) on text is not reliable (JIT gives `false`, the interpreter errors). Lists, records and variants are never equal under `==` (`[1, 2] == [1, 2]` is `false`); compare elements or fields individually. Numbers, text and booleans compare by value.

```
bind start = do {
  println(2 + 3 * 4)
  println((2 + 3) * 4)
  println(10 - 3 - 2)
  println(-3 + 5)
  println(2 * -4)
  println(1 < 2 && 2 < 3)
  println(!true || 1 == 1)
  println("n=" + 5)
  println([1, 2] == [1, 2])
}
```

```
OUTPUT
14
20
5
2
-8
true
true
n=5
false
```

Bindings, scope and shadowing

`bind name = expression` creates a binding. There is **no assignment statement** (`x = 5` is a syntax error); to change a value, bind the name again. A later `bind` of the same name replaces the earlier one (shadowing).

```
bind start = do {
  bind x = 1
  bind x = x + 10
  bind x = x * 2
  println(x)
}
```

```
OUTPUT
22
```

Where a re-binding is visible depends on the construct that contains it:

Construct	Bindings made inside it
`if` / `else` body	stay visible afterwards (shared scope)
`while` body	stay visible afterwards, so loops can accumulate
`for` body	**discarded** at the end of each iteration (fresh copy per iteration)
`match` arm	discarded
function body	local to the call
`do { }` block	discarded by the interpreter, **leaked** into the enclosing scope by the JIT

The consequence that surprises most newcomers: a `for` loop cannot accumulate into an outer variable. Use `while`.

```
bind start = do {
  bind total = 0
  for i in 1..5 { bind total = total + i }
  println(total)
  bind t2 = 0
  bind i = 1
  while i < 5 {
    bind t2 = t2 + i
    bind i = i + 1
  }
  println(t2)
}
```

```
OUTPUT
0
10
```

Top-level `bind` values are global constants. Functions can read them, but each function call starts with only its parameters and those globals; the local variables of the caller are not visible.

```
bind RATE = 3
fn scaled(n) { n * RATE }
bind start = do {
  bind RATE = 100
  println(scaled(2))
}
```

```
OUTPUT
6
```

Functions

`fn name(a, b) { ... }`. The result is the last expression, or the value of `return`. Functions may be recursive and may be declared in any order (an `fn` below `start` is visible to it). Functions cannot be nested inside other functions. Calling a function with too few arguments leaves the missing parameters undefined; using one is a runtime error on the interpreter and yields `()` on the JIT.

```
fn fib(n) {
  if n < 2 { return n }
  fib(n - 1) + fib(n - 2)
}
fn greet(name) { "Hello, " + name + "!" }
bind start = do {
  println(fib(15))
  println(greet("Ling"))
}
```

```
OUTPUT
610
Hello, Ling!
```

A function that is called with an unknown name fails with `unknown function` on the interpreter, but the JIT backend quietly evaluates the call to `()`. Check spelling carefully; use `ling run --interp` to get a diagnostic.

Control flow

if / else

`if` is an expression. Braces are required. See the `if` keyword entry for truthiness caveats.

```
bind start = do {
  bind n = 7
  bind label = if n < 0 { "negative" } else if n == 0 { "zero" } else { "positive" }
  println(label)
}
```

```
OUTPUT
positive
```

while

```
bind start = do {
  bind n = 27
  bind steps = 0
  while n != 1 {
    bind n = if n % 2 == 0 { n / 2 } else { 3 * n + 1 }
    bind steps = steps + 1
  }
  println(steps)
}
```

```
OUTPUT
111
```

for and early exit

Loops over a range, a list (or, on the interpreter, a string). There is no `break` / `continue` (`stop` and `again` are reserved words that do nothing); leave a loop early by `return`ing from a function.

```
fn index_of(xs, target) {
  bind i = 0
  while i < len(xs) {
    if xs[i] == target { return i }
    bind i = i + 1
  }
  0 - 1
}
bind start = do {
  println(index_of(["a", "b", "c"], "b"))
  println(index_of(["a"], "z"))
}
```

```
OUTPUT
1
-1
```

The example uses `while` with an explicit index because a counter updated inside a `for` body would be discarded after each iteration.

---

match

See the `match` keyword entry. Patterns are tried top to bottom.

```
fn classify(n) {
  match n {
    0 => "zero",
    1 => "one",
    other => "many (" + to_str(other) + ")"
  }
}
bind start = do {
  println(classify(0))
  println(classify(1))
  println(classify(9))
}
```

```
OUTPUT
zero
one
many (9)
```

Records and enumerations

`form` defines a record and `choose` a tagged union. Both are used by calling the declared name as a constructor.

```
form Player { name, hp, }
choose Cmd { Move(dx, dy), Say(text), Quit }
fn run(c) {
  match c {
    Move(dx, dy) => "move " + to_str(dx) + "," + to_str(dy),
    Say(t) => "say " + t,
    Quit() => "bye"
  }
}
bind start = do {
  bind p = Player("Lin", 30)
  println(p.name + " has " + to_str(p.hp))
  println(run(Move(1, 0 - 1)))
  println(run(Say("hi")))
  println(run(Quit()))
}
```

```
OUTPUT
Lin has 30
move 1,-1
say hi
bye
```

Constructing with the wrong number of fields is an error on the interpreter (`P expects 2 field(s), got 1`) and silently yields `()` on the JIT backend.

Closures

A closure is written `|params| expression`, for example `|x| x * 2` or `|a, b| a + b`. A closure with no parameters must be written `| | expression` (with a space) because `||` lexes as the logical-or operator. Closures capture the variables visible where they are created.

Closures are the least reliable part of the language. Calling a closure held in a variable (`bind f = |x| x * 2` then `f(3)`) fails on the interpreter (`unknown function 'f'`, because a call by bare name only looks up declared `fn` items) and gives `()` or a correct answer on the JIT depending on the surrounding code. Passing a closure to a function that calls it fails on the JIT. What is dependable:

- Calling a closure expression directly: `(|x| x + 1)(5)`.
- Handing a closure to the builtins that take one (`try`, `list_map`, `list_filter`, `list_find`) **on the interpreter**.

Prefer a named `fn` whenever you need a function value of your own.

INTERPRETER ONLY

```
bind start = do {
  println((|x| x + 1)(5))
  println(list_map([1, 2, 3], |x| x * 10))
}
```

OUTPUT

```
6
[10, 20, 30]
```

The default JIT backend prints this instead:

OUTPUT

```
6
()
```

Modules and imports

Inline modules. `mod name { fn ... }` groups functions under a namespace called with `name::function(...)`.

File imports. `use "path"` loads another file; `use "path" as alias` namespaces it. The path is relative to the importing file and the `.ling` extension may be left off. A file used as a library has no entry point; the entry file is the one passed to `ling run`.

```
mod text_tools {
  fn shout(s) { str_upper(s) + "!" }
}
use "mathutil.ling" as mu
bind start = do {
  println(text_tools::shout("hey"))
  println(mu::cube(4))
}
```

```
OUTPUT
HEY!
64
```

where `mathutil.ling` contains:

```
FRAGMENT
fn cube(n) { n * n * n }
```

Imported files share the limitation noted for `mod`: values bound at the top level of an imported file are not usable; export functions instead.

Ownership words

Ling's design documents describe an ownership system (`own`, `lend`, `share`, `move`, `copy`, `change`). In the shipped runtime these words are accepted as prefix operators on expressions (and before parameter names) and then **ignored**: there is no borrow checker, use-after-move is not detected, and `change` is not even an operator. Memory is managed automatically. They are safe to write as documentation of intent. See the keyword entries.

Concurrency and `async`

`async` and `wait` are accepted syntax with no scheduling behind them: an `async fn` runs synchronously when called, and `wait` returns the value on the interpreter but `()` on the JIT. `spawn` is not implemented. Genuine parallelism is provided by the `thread_*`, `chan_*` and `shared_*` builtins, where each thread is an isolated interpreter that communicates by channels; see the builtins chapter.

Error handling

There are no exceptions and no `try` / `catch` syntax. Recoverable failures are values: a function returns `ok(value)` or `bad(reason)` and callers unpack the result with `match`. Many builtins follow a softer convention instead - they return `0`, `false`, `""` or `[]` on failure (for example `to_number`, `list_get`, `file_exists`). A fatal problem raises a **runtime error** that ends the program with a message and a traceback; `try(closure)` (interpreter only) converts such an error into `bad(message)`.

```
fn parse_pair(text) {
  bind parts = split(text, ",")
  if len(parts) != 2 { return bad("expected two values") }
  ok(to_number(parts[0]) + to_number(parts[1]))
}
bind start = do {
  println(match parse_pair("3,4") { ok(v) => v, bad(e) => e })
  println(match parse_pair("3") { ok(v) => v, bad(e) => e })
}
```

```
OUTPUT
7
expected two values
```

Runtime errors look like this (interpreter):

```
INTERPRETER ONLY
bind start = do {
  println(read_file("no_such_file.txt"))
}
```

```
OUTPUT
error[runtime]: read_file 'no_such_file.txt': The system cannot find the file
specified. (os error 2)
 --> example.ling
  traceback (deepest call last):
    0: start
```

The default JIT backend prints this instead:

```
OUTPUT
()
```

Writing Ling in other languages

Because the lexer maps each keyword spelling to one token, a file may be written entirely in Chinese, Japanese, Korean, Thai and other supported languages, and may mix them. Builtin names have translations too (for example `印` for `print`; see each builtin entry). `ling` prints `[detected language: ...]` on standard error for non-English files. Use `lingfu normalize <lang>` to rewrite a whole project into another language.

```
函 加倍(n) { 归 n * 2 }
令 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" < "b"`	`false`	runtime error
`read_file` of a missing file	`()`	runtime error

Situation	JIT (default)	Interpreter (`--interp`)
Wrong constructor arity	`()`	runtime error
Top-level `return` in `start`	ends program quietly	runtime error

A program that avoids these corners behaves identically on both engines.

CHAPTER 12

Core Builtins Reference

Every function in this chapter is called by name, for example `sqrt(16)`. Parameters are positional; a missing trailing argument usually takes a documented default instead of raising an error. Each entry lists the spellings that also work in Chinese (zh), Japanese (ja), Korean (ko) and Thai (th); the runtime recognises further aliases in French, German, Russian, Arabic, Persian, Hebrew and Urdu which are not listed here.

Examples marked `ling` were run with the default `ling run` backend and the output shown is real. Where the tree-walking interpreter (`ling run --interp`) prints something different the difference is shown under the example. Examples marked `ling-interp` only work on the interpreter, and `ling-gui` examples were checked against the source but not executed. Blocks marked `ling-module` are library files, not programs.

Graphics, audio, networking, cryptography, physics and AI builtins are documented in their own chapters.

Output and formatting

Console output. `print` and `println` are the same builtin: both write every argument converted to text, **joined with no separator**, followed by a newline. Use `format` or `+` to build spaced output.

Name	Summary
`print(value, ...)`	Writes its arguments to standard output, then a newline
`println(value, ...)`	Identical to `print`: writes the arguments joined without a separator and ends with a newline
`print_color(color, value, ...)`	Prints a line wrapped in a bold bright ANSI colour escape sequence
`print_file(text)`	Writes text to standard output **without** a trailing newline
`format(template, value, ...)`	Builds a string by substituting each `{}` in `template` with the next argument
`format::join(list)`	Concatenates every element of a list as text, with no separator

`print(value, ...)`

Writes its arguments to standard output, then a newline.

- **Aliases:** English: `println`; zh: `印`, `打印`; ja: `印刷`; ko: `출력`; th: `พิมพ์` (further languages exist)
- **Parameters:** Any number of values of any type. Each is converted to text; there is no separator between them.
- **Returns:** `()` (unit).
- **Note:** `print` appends a newline exactly like `println`; there is no newline-free variant (see `print_file` for that).
- **Note:** Lists print as `[1, 2, 3]`, results as `Ok(v)` / `Err(e)`, whole numbers without a decimal point (`3`, not `3.0`).

```
bind start = do {
  print("Hello, Ling!")
  print(42)
  print(3.5)
  print(true)
  print([1, 2, 3])
  print("a", "b", 1 + 2)
}
```

```
OUTPUT
Hello, Ling!
42
3.5
true
[1, 2, 3]
ab3
```

`println(value, ...)`

Identical to `print`: writes the arguments joined without a separator and ends with a newline.

- **Aliases:** English: `print`; zh: `印`, `打印`; ja: `印刷`; ko: `출력`; th: `พิมพ์` (further languages exist)
- **Parameters:** Any number of values.
- **Returns:** `()`.

- **Note:** Gotcha: `println("sum", 1 + 2)` prints `sum3`, not `sum 3`. Put the space inside a string (`"sum "`) or use `format`.

```ling
bind start = do {
  println("sum", 1 + 2)
  println("sum ", 1 + 2)
  println("a", "b", "c")
}
```

```
OUTPUT
sum3
sum 3
abc
```

`format(template, value, ...)`

Builds a string by substituting each `{}` in `template` with the next argument.

- **Aliases:** zh: `格式`; ja: `フォーマット`; ko: `서식`; th: `รูปแบบ` (further languages exist)
- **Parameters:** `template` - text containing `{}` placeholders. `{:.N}` formats the next argument (when it is a number) with N decimal places. Remaining arguments fill the placeholders in order.
- **Returns:** Text.
- **Note:** A `{}` with no matching argument is dropped (becomes empty). Extra arguments are ignored.
- **Note:** Only `{}` and `{:.N}` are supported; `{:5}`, `{:>8}`, `{0}` and `{name}` are not. A literal brace cannot be escaped.
- **Note:** If the first argument is not text, it is simply converted to text and returned.

```ling
bind start = do {
  println(format("x={} y={}", 1, "two"))
  println(format("pi is about {:.3}", 3.14159265))
  println(format("{} and {}", "only one"))
  println(format("[{}]", [1, 2]))
}
```

```
OUTPUT
x=1 y=two
pi is about 3.142
only one and
[[1, 2]]
```

`ok(value)`

Wraps a value as a success result.

- **Aliases:** zh: `好`; ja: `良し`; ko: `좋아`; th: `โอเค` (further languages exist)
- **Parameters:** `value` - any value (default `()`).
- **Returns:** A result that prints as `Ok(value)`.
- **Note:** Use `match r { ok(v) => ..., bad(e) => ... }` to unpack. Calling any method on a result returns the inner value (`ok(5).anything` is `5`), which is rarely what you want; prefer `match`.

```
fn half(n) {
  if n % 2 == 0 { ok(n / 2) } else { bad("odd number") }
}
bind start = do {
  println(half(10))
  println(match half(10) { ok(v) => v, bad(e) => e })
}
```

```
OUTPUT
Ok(5)
5
```

`bad(reason)`

Wraps a value as a failure result.

- **Aliases:** English: `err`; zh: `坏`; ja: `悪い`; ko: `나쁨`; th: `ผิด` (further languages exist)
- **Parameters:** `reason` - any value, usually a message string.
- **Returns:** A result that prints as `Err(reason)`.

- **Note:** The English builtin also answers to `err`. Note the printed form is `Err(...)`, while the pattern and constructor are spelled `bad`.

```
fn half(n) {
  if n % 2 == 0 { ok(n / 2) } else { bad("odd number") }
}
bind start = do {
  println(half(7))
  println(match half(7) { ok(v) => "value", bad(e) => e })
}
```

```
OUTPUT
Err(odd number)
odd number
```

`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.

```
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.

```ling
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
```

`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
```

`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
```

`len(value)`

Number of characters in a string, or elements in a list.

- **Aliases:** English: `str_len`; zh: `长度`; ja: `長さ`; ko: `길이`; th: `ความยาว` (further languages exist)
- **Parameters:** `value` - text or list. Anything else gives `0`.

- **Returns:** Number.
- **Note:** The English builtin is `str_len`; `len` is its alias. The method form `"abc".len()` counts **bytes** (`"é".len()` is 2) while `len("é")` counts characters (1).

```
bind start = do {
  println(len("hello"))
  println(len("héllo"))
  println(len([10, 20, 30]))
  println(len(42))
}
```

```
OUTPUT
5
5
3
0
```

`to_str(value)`

Converts any value to its text form.

- **Aliases:** English: `str`, `num_str`; th: `แปลงสตริง` (further languages exist)
- **Parameters:** `value` - any value.
- **Returns:** Text.

```
bind start = do {
  println(to_str(12) + "!")
  println(str(3.5))
  println(to_str([1, 2]))
  println(len(to_str(1000)))
}
```

```
OUTPUT
12!
3.5
[1, 2]
4
```

`to_number(text)`

Parses text as a decimal number.

- **Aliases:** zh: `转数字` (further languages exist)
- **Parameters:** `text` - leading and trailing whitespace is ignored.
- **Returns:** Number; `0` **if the text is not a number** (no error).
- **Note:** Cannot distinguish `"0"` from invalid input. Hex, exponents and underscores are not recognised; see `hex_to_number`.

```
bind start = do {
  println(to_number("42") + 1)
  println(to_number("3.5"))
  println(to_number("abc"))
  println(to_number("  7  "))
}
```

```
OUTPUT
43
3.5
0
7
```

`split(text, separator)`

Splits text at every occurrence of a separator.

- **Aliases:** English: `str_split`; th: `แยก` (further languages exist)
- **Parameters:** `text`; `separator` (default newline; an empty separator also means newline).
- **Returns:** List of strings.
- **Note:** Splitting never trims; consecutive separators produce empty strings.

```
bind start = do {
  println(split("a,b,c", ","))
  println(len(split("a,,b", ",")))
  println(split("line1\nline2"))
}
```

```
OUTPUT
[a, b, c]
3
[line1, line2]
```

`trim(text)`

Removes whitespace from both ends.

- **Aliases:** English: `str_trim`; th: `ตัดช่องว่าง` (further languages exist)
- **Parameters:** `text`.
- **Returns:** Text.

```
bind start = do {
  println("[" + trim("  hi  ") + "]")
}
```

```
OUTPUT
[hi]
```

`starts_with(text, prefix)`

True if `text` begins with `prefix`.

- **Aliases:** English: `str_starts_with`; th: `เริ่มด้วย` (further languages exist)
- **Parameters:** `text`, `prefix`.
- **Returns:** Boolean.

```
bind start = do {
  println(starts_with("hello", "he"))
  println(starts_with("hello", "lo"))
}
```

```
OUTPUT
true
false
```

`ends_with(text, suffix)`

True if `text` ends with `suffix`.

- **Aliases:** English: `str_ends_with`; th: `ลงท้ายด้วย` (further languages exist)
- **Parameters:** `text`, `suffix`.
- **Returns:** Boolean.

```
bind start = do {
  println(ends_with("report.pdf", ".pdf"))
}
```

```
OUTPUT
true
```

`str_replace(text, from, to)`

Replaces **every** occurrence of `from` with `to`.

- **Aliases:** th: `แทนสตริง` (further languages exist)
- **Parameters:** `text`, `from`, `to`.
- **Returns:** Text.

```
bind start = do {
  println(str_replace("a-b-c", "-", "+"))
}
```

```
OUTPUT
a+b+c
```

`str_find(text, needle)`

Position of the first occurrence of `needle`.

- **Aliases:** th: `หาในสตริง` (further languages exist)
- **Parameters:** `text`, `needle`.
- **Returns:** Character index (0-based), or `-1` if absent.

```ling
bind start = do {
  println(str_find("hello", "l"))
  println(str_find("hello", "z"))
}
```

```
OUTPUT
2
-1
```

`substr(text, start, length)`

Extracts a slice by character position.

- **Aliases:** English: `str_slice`; th: `ส่วนสตริง` (further languages exist)
- **Parameters:** `text`; `start` (0-based); `length` (optional; default: to the end).
- **Returns:** Text. Out-of-range values are clamped, never an error.

```ling
bind start = do {
  println(substr("hello", 1, 3))
  println(substr("hello", 2))
  println(substr("hello", 10, 2) + "|")
}
```

```
OUTPUT
ell
llo
|
```

`str_repeat(text, count)`

Repeats a string.

- **Aliases:** th: `ทำซ้ำสตริง` (further languages exist)
- **Parameters:** `text`; `count` (default 1).
- **Returns:** Text.

```
bind start = do {
  println(str_repeat("ab", 3))
  println(str_repeat("=", 10))
}
```

```
OUTPUT
ababab
==========
```

`str_upper(text)`

Converts to upper case.

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

```
bind start = do {
  println(str_upper("hello"))
}
```

```
OUTPUT
HELLO
```

`str_lower(text)`

Converts to lower case.

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

```
bind start = do {
  println(str_lower("HeLLo"))
}
```

```
OUTPUT
hello
```

`.len()`

Length of a string (in **bytes**) or list (elements).

- **Aliases:** zh: `长`, `长度`; ja: `長さ`; ko: `길이`; th: `ความยาว`
- **Parameters:** receiver: text or list.
- **Returns:** Number.
- **Note:** For text, use the builtin `len(text)` when you want characters.

```
bind start = do {
  println([1, 2, 3].len())
  println("abc".len())
  println("é".len())
  println(len("é"))
}
```

```
OUTPUT
3
3
2
1
```

`.push(value)`

Returns a copy of the list with `value` appended.

- **Aliases:** zh: `推`, `添加`; ja: `追加`; ko: `추가`; th: `เพิ่ม`
- **Parameters:** receiver: list; `value`.
- **Returns:** New list.
- **Note:** The original list is unchanged under the interpreter, so write `bind xs = xs.push(4)`. Under the default JIT backend the original may be updated in place as well; do not rely on either.

```ling
bind start = do {
  bind xs = [1, 2]
  bind xs = xs.push(3)
  println(xs)
}
```

```
OUTPUT
[1, 2, 3]
```

`.contains(text)`

True if a string contains a substring.

- **Aliases:** zh: `包含`
- **Parameters:** receiver: text; `text`.
- **Returns:** Boolean (false for non-string argument).

```ling
bind start = do {
  println("hello".contains("ell"))
  println("hello".contains("z"))
}
```

```
OUTPUT
true
false
```

`.is_empty()`

True if a string has no characters.

- **Aliases:** zh: `是空`
- **Parameters:** receiver: text.
- **Returns:** Boolean.

```
bind start = do {
  println("".is_empty())
  println("x".is_empty())
}
```

```
OUTPUT
true
false
```

`.name`

Reads a field of a `form` (struct) value.

- **Aliases:** none (the field name is whatever you declared)
- **Parameters:** receiver: struct value; the field name follows the dot.
- **Returns:** The field value. Error `no field ...` if it does not exist.

```
form Point { x, y }
bind start = do {
  bind p = Point(3, 4)
  println(p.x)
  println(p.y)
}
```

```
OUTPUT
3
4
```

Lists

Lists are ordered, mixed-type sequences written `[1, "two", 3.0]`. They behave as values: functions return a new list instead of mutating. Because rebinding is the only form of assignment, the idiom is `bind xs = list_push(xs, item)`. Indexing is `xs[i]` (zero-based); reading past the end is a runtime error under the interpreter.

Name	Summary
`list_new()`	Creates an empty list
`list_push(list, value)`	Returns the list with `value` appended
`list_get(list, index)`	Reads an element by zero-based index
`list_set(list, index, value)`	Returns the list with the element at `index` replaced
`list_max(list, default)`	Largest number in a list
`list_min(list, default)`	Smallest number in a list
`list_join(list, separator)`	Joins the elements into one string
`list_map(list, function)`	Applies a function to every element
`list_filter(list, predicate)`	Keeps the elements for which the predicate returns `true`
`list_find(list, predicate)`	First element for which the predicate returns `true`

`list_new()`

Creates an empty list.

- **Aliases:** English: `Vec::with_capacity(n)`, `Vec::from(list)`; zh: `新建列表`; ja: `新規リスト`; ko: `새목록`; th: `รายการใหม่` (further languages exist)
- **Parameters:** none.
- **Returns:** Empty list.

```
bind start = do {
  bind xs = list_new()
  println(xs)
  println(len(xs))
}
```

```
OUTPUT
[]
0
```

`list_push(list, value)`

Returns the list with `value` appended.

- **Aliases:** zh: `列表添加`; ja: `リスト追加`; ko: `목록추가`; th: `เพิ่มรายการ` (further languages exist)

- **Parameters:** `list` ; `value` .
- **Returns:** The extended list.
- **Note:** Always rebind the result: `bind xs = list_push(xs, 5)` . The interpreter optimises exactly this pattern to run in place.
- **Note:** If the first argument is not a list, a new one-element list is returned.

```
bind start = do {
  bind xs = list_new()
  bind xs = list_push(xs, 10)
  bind xs = list_push(xs, "twenty")
  println(xs)
}
```

```
OUTPUT
[10, twenty]
```

`list_get(list, index)`

Reads an element by zero-based index.

- **Aliases:** zh: `取元素` , `要素取得` ; ko: `요소가져오기` ; th: `รับรายการ` (further languages exist)
- **Parameters:** `list` ; `index` (default 0).
- **Returns:** The element; **an empty string `""` if the index is out of range** (no error).
- **Note:** `xs[i]` is the stricter form: it is a runtime error when out of range under the interpreter.

```
bind start = do {
  bind xs = [10, 20, 30]
  println(list_get(xs, 1))
  println("[" + list_get(xs, 9) + "]")
}
```

```
OUTPUT
20
[]
```

`list_set(list, index, value)`

Returns the list with the element at `index` replaced.

- **Aliases:** zh: `设元素`, `要素設定`; ko: `요소설정`; th: `ตั้งรายการ` (further languages exist)
- **Parameters:** `list`; `index`; `value`.
- **Returns:** Updated list. An out-of-range index leaves it unchanged.
- **Note:** Rebind the result: `bind xs = list_set(xs, 0, 99)`.

```
bind start = do {
  bind xs = [1, 2, 3]
  bind xs = list_set(xs, 0, 99)
  println(xs)
}
```

```
OUTPUT
[99, 2, 3]
```

`list_join(list, separator)`

Joins the elements into one string.

- **Aliases:** English: `join`; zh: `连接`, `連結`; ko: `연결`; th: `รวมรายการ` (further languages exist)
- **Parameters:** `list`; `separator` (default empty).
- **Returns:** Text.

```
bind start = do {
  println(list_join(["a", "b", "c"], "-"))
  println(join([1, 2, 3], ", "))
}
```

```
OUTPUT
a-b-c
1, 2, 3
```

`read_file(path)`

Reads an entire text file.

- **Aliases:** th: `อ่านไฟล์` (further languages exist)
- **Parameters:** `path` - file path.
- **Returns:** Text (UTF-8). A missing or unreadable file is a runtime error (`read_file 'path': ...`).
- **Note:** Under the default JIT backend a failed read yields `()` instead of an error, so test with `file_exists` first.

```
bind start = do {
  write_file("note.txt", "line one\nline two")
  println(read_file("note.txt"))
  println(len(split(read_file("note.txt"), "\n")))
  remove_file("note.txt")
}
```

```
OUTPUT
line one
line two
2
```

`write_file(path, text)`

Writes text to a file, creating parent folders and replacing existing content.

- **Aliases:** th: `เขียนไฟล์` (further languages exist)
- **Parameters:** `path`; `text`.
- **Returns:** `()`. Failure to write is a runtime error.

```
bind start = do {
  write_file("out/data.txt", "hello")
  println(file_exists("out/data.txt"))
  remove_dir_all("out")
}
```

```
OUTPUT
true
```

`file_exists(path)`

True if the path exists (file or folder).

- **Aliases:** zh: `文件存在` (further languages exist)
- **Parameters:** `path`.
- **Returns:** Boolean.

```
bind start = do {
  println(file_exists("definitely_missing.txt"))
}
```

```
OUTPUT
false
```

`is_file(path)`

True if the path is an existing regular file.

- **Aliases:** th: `เป็นไฟล์` (further languages exist)
- **Parameters:** `path`.
- **Returns:** Boolean.

```
bind start = do {
  write_file("t.txt", "x")
  println(is_file("t.txt"))
  println(is_file("."))
  remove_file("t.txt")
}
```

```
OUTPUT
true
false
```

`is_dir(path)`

True if the path is an existing directory.

- **Aliases:** th: `เป็นไดเรกทอรี` (further languages exist)
- **Parameters:** `path`.
- **Returns:** Boolean.

```
bind start = do {
  println(is_dir("."))
  println(is_dir("missing_dir"))
}
```

```
OUTPUT
true
false
```

`sleep(milliseconds)`

Pauses the program.

- **Aliases:** English: `sleep_ms`; zh: `睡眠`, `流水:睡眠`; ja: `眠る`, `スリープ`; ko: `잠자기`, `잠`; th: `หยุด`, `นอน` (further languages exist)
- **Parameters:** `milliseconds` - number.
- **Returns:** `()`.

```
bind start = do {
  sleep(50)
  println("done")
}
```

```
OUTPUT
done
```

`rand()`

Pseudo-random number in the range 0 up to (not including) 1.

- **Aliases:** zh: `随机`; ja: `乱数`; ko: `난수`; th: `สุ่ม` (further languages exist)
- **Parameters:** none (arguments are ignored).
- **Returns:** Number.
- **Note:** There is no seeding function and no integer form: scale and floor it, e.g. `floor(rand() * 6) + 1` for a die.
- **Note:** Not suitable for security; use `random_hex` for unpredictable bytes.

```
bind start = do {
  bind r = rand()
  println(r >= 0 && r < 1)
  bind die = floor(rand() * 6) + 1
  println(die >= 1 && die <= 6)
}
```

```
OUTPUT
true
true
```

`time_now()`

Seconds elapsed since the program started.

- **Aliases:** zh: `当前时间`; ja: `経過時間`; ko: `현재시간`; th: `เวลาปัจจุบัน` (further languages exist)
- **Parameters:** none.
- **Returns:** Number (fractional seconds).

```
bind start = do {
  bind t0 = time_now()
  sleep(20)
  println(time_now() - t0 >= 0.02)
}
```

```
OUTPUT
true
```

CHAPTER 13

Command Line Tools: `ling` and `lingfu`

Two programs ship with Ling:

- `ling` compiles, runs, visualises and packages individual programs.
- `lingfu` (灵符, "spirit token") is the project manager: it scaffolds projects, runs their tests, manages the `Ling.toml` manifest, publishes to a package registry and rewrites a project into another human language.

Both are installed to the same directory (`~/.cargo/bin` for a Cargo install). `lingfu` finds `ling` next to itself, then on `PATH`, and delegates to it for running and building. The transcripts below were produced with Ling 2030.1.53 (`ling`) and lingfu 2030.1.19.

Transcripts marked `console` are real terminal sessions; long absolute paths have been shortened.

`ling`

```
OUTPUT
ling run [--interp|--wasm|--O0|--O1|--O2|--O3] <file.ling>
ling <file.ling>                    (shorthand for: ling run <file.ling>)
ling build <file.ling|dir> [--out dir] [--platform list] [--icon file] [--pack] [--aot]
ling ast [path] [--technical|--artwork|--ling|--all] [--out dir]
ling visualize <file.ling>
ling convert <asset> [-o out.ling] [--no-compression]
```

Running `ling` with no arguments, or with anything it does not recognise, prints the usage text and exits with status 0. There is no `--version` or `--help` flag; the first line of the usage text carries the version (`ling 2030.1.53 – The Omniglot Systems Language`). Subcommands that other tools sometimes expect (`check`, `fmt`, `translate`, `test`) do **not** exist in `ling`; see the remarks under `lingfu check`.

A source file may use any of the extensions `.ling`, `.灵`, `.靈`, `.령` or `.ลิง`.

`ling run`

Runs a program: it parses the file, finds the entry point (`bind start = do { ... }`; see the syntax chapter) and executes it.

Flag	Effect
(none)	Cranelift JIT backend (the default). Falls back to the interpreter for the whole program if the JIT cannot compile it.
`--interp`	Tree-walking interpreter, the semantic reference. Gives the best diagnostics.
`--wasm` (alias `--web`)	Builds the program to WebAssembly, serves it on a local web server and opens the browser. Not verified here.
`--O0` `--O1` `--O2` `--O3`	JIT optimisation level (default `--O3`). Use a lower level if a very large program takes too long to compile.

The flags may appear before or after the file name; the first argument with a Ling extension is the program. Any further arguments stay on the command line and are visible to the program: `get_args()` returns the whole command line (including `ling`, `run` and the script path) and `cli_arg("port", "8080")` reads a `--port 99` option.

```
OUTPUT
$ ling run args.ling --port 99 extra
99
6
```

Behaviour worth knowing:

- Programs that call `http_serve` or `lingtp_serve` are automatically run on the interpreter, because the JIT does not yet support closure arguments.
- A file whose entry point is missing is treated as a library and fails with `error[runtime]: no entry point`, exit status 1.
- For a non-English source file, `ling` writes `[detected language: Chinese (...)]` (or Thai, Japanese, ...) to **standard error** before running.
- A missing file gives `[ling] error: file does not exist: <path>` and status 1; no file at all prints `Usage: ling run [--wasm|--interp|--O0|--O1|--O2|--O3] <file.ling>` and status 1.
- Parse errors are reported as `error[parse]: ...` with status 1. Runtime errors are `error[runtime]: ...` followed by a `traceback`.

```
OUTPUT
$ cat hello.ling
fn sq(n) { n * n }
bind start = do {
  println(sq(7))
}
$ ling run hello.ling
49
$ ling hello.ling
49
$ ling run --interp --O0 hello.ling
49
$ ling run nofile.ling
[ling] error: file does not exist: nofile.ling
```

Commands at a glance

Command	English aliases	Purpose
`new <name>`	`create`	Create a project folder
`init`	`initialize`	Scaffold a project in the current folder
`build`	`compile`, `make`	Build a native executable
`run`	`execute`, `start`	Run the project
`test`	`verify`	Run the project's tests
`check`	`lint`, `validate`	Check sources (see warning)
`clean`	`purge`, `clear`	Delete build output
`doc`	`docs`, `document`	Build documentation
`fmt`	`format`, `style`	Format sources (see warning)
`tree`	`structure`, `ls`	Print the project tree
`bench`	`benchmark`, `measure`	Run `cargo bench`
`update`	`upgrade`, `refresh`	Run `cargo update`
`add <package>`	`install`, `include`	Install a package from the registry
`search <query>`	`find`, `seek`	Search the registry
`publish`	`release`, `share`	Publish the project
`login <api-key>`	`sign-in`, `auth`	Save registry credentials

Command	English aliases	Purpose
`logout`	`sign-out`, `deauth`	Forget credentials
`translate <file>`	`trans`, `convert`	Translate keywords between languages (see warning)
`manifest`	`show`, `info`	Print the manifest
`wizard`	`guide`, `create-wizard`	Interactive project creator
`normalize <lang>`	`norm`, `localize`	Rewrite the project into another language
`seed`	-	Integrity fingerprint of the project
`ast`, `convert`	-	Forwarded to `ling ast` / `ling convert`

Options common to several commands: `--lang` / `-l <code>` (scaffolding language: `en`, `zh`, `ja`, `ko`, `th`, `ru`), `--type` / `-t <kind>`, `--root-lang <code>`, `--crates`, `--path <dir>`, `--dry-run` / `-n`.

Localised command names. Each command also has aliases in Chinese, Japanese, Korean, Thai, Russian, Arabic, Hindi, Vietnamese, Greek, Hebrew, German, French, Spanish, Turkish and Polish (for example `试` / `测` for `test` in Chinese, `ทดสอบ` in Thai). These are recognised **only when the executable is invoked under its localised name** - `灵符.exe`, `靈符`, `영부`, `ลิงฟู`, `линфу`, `لينغفو`, `लिंगफू`, `linhphu`, `λινγφυ`, `לינגפו`, `lingfu-de`, `lingfu-fr`, `lingfu-es`, `lingfu-tr`, `lingfu-pl`. Under the normal name `lingfu` only the English words work, and messages are English:

```
OUTPUT
$ lingfu 试
unknown command: 试
  lingfu --help
$ ./灵符.exe 试          # same program, copied under its Chinese name
正在试灵...
  试验: ...\tests\test.ling
...
```

`lingfu new`

```
OUTPUT
lingfu new <name> [--type bin|lib|game|ui|ai|crypto|web|polyglot] [--lang
en|zh|ja|ko|th|ru] [--root-lang <code>]
```

Creates `<name>/` containing a manifest, a source folder with an entry program, a tests folder, a lexicon folder, `README.md`, `.gitignore` and `ling.ignore`. `--type` also accepts localised words (for example `lib` = `共修` = `라이브러리`). `--lang` chooses the language of the generated files **and** of the folder names; `--root-lang` chooses the language of the `core` dependency.

```
OUTPUT
$ lingfu new demo
~ Ling Manifest - Spirit Token Descends ~
Creating project: demo

Spirit Realm created!
  cd demo
  lingfu build
  lingfu run
$ find demo -type f
demo/.gitignore
demo/lexicons/en.ling
demo/ling.ignore
demo/Ling.toml
demo/README.md
demo/src/start.ling
demo/tests/test.ling
```

Folder and file names depend on the scaffold language:

Language	Manifest	Sources	Entry	Tests	Test file	Build	Lock	Lexicons
English	`Ling.toml`	`src`	`start.ling`	`tests`	`test.ling`	`build`	`.lock`	`lexicons`
Chinese	`灵符.toml`	`灵源`	`启.灵`	`灵镜`	`试.灵`	`灵碑`	`灵符锁`	`灵言`
Japanese	`霊符.toml`	`霊源`	`啓.霊`	`霊鏡`	`試.霊`	`霊碑`	`霊符錠`	`霊語`
Korean	`영부.toml`	`령원`	`시작.령`	`령경`	`시험.령`	`령비`	`령잠금`	`령어`
Thai	`ลิงฟู.toml`	`ต้นกำเนิด`	`เริ่ม.ลิง`	`ทดสอบ`	`ทดสอบ.ลิง`	`สร้าง`	`ล็อก`	`คำศัพท์`

(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

```ling
# src/start.ling - gateway to the Spirit Realm

bind start = do {
    print("Welcome to demo Spirit Realm")
    print("Spirit Level: 铁")

    # Your Ling code awakens here
}
```

The `lingfu` manifest uses the tables `[spirit]` (`name`, `kind`, `version`, and optionally `entry`), `[author]`, `[realm]` (`language`, `level`, `source`), `[root]` (the `core` dependency) and `[depends]`. Two tools read it with different parsers, which matters:

- `lingfu run` parses the file as real TOML and honours `entry` in `[spirit]` (or `[package]`).
- `ling build` reads `name`, `version`, `kind`, `entry`, `icon`, `targets`, `pack`, `includes`, … line by line (see the key table above) and defaults `entry` to `main.ling`.

Because the generated project keeps its program in `src/start.ling`, **add** `entry = "src/start.ling"` **under** `[spirit]` before running `lingfu build`; without it the build stops with `entry '...\main.ling' not found`. `lingfu run` and `lingfu test` work out of the box because they search `src/start.ling` themselves.

`lingfu init`

`lingfu init [--type ...] [--lang ...]` does the same scaffolding as `new`, but inside the current directory, using its name as the project name.

`lingfu run`

Runs the project's entry program with `ling run`. The entry is the first existing file among: the manifest's `[spirit] entry`, `<src>/<entry file>`, `<root>/<entry file>`, `<src>/main.ling`, `<root>/main.ling`. If there is no manifest, it looks in the current directory for a `.ling` file (`main.ling`, `src/main.ling`, `src/start.ling`, ...); as a last resort it runs `cargo run`. Extra arguments are passed to `ling run`.

```
OUTPUT
$ cd demo
$ lingfu run
Running ...\demo\src\start.ling...
Welcome to demo Spirit Realm
Spirit Level: 铁
```

`lingfu test`

Runs `<src>/<test file>` if it exists (for English projects `src/test.ling`), then every `.ling` file below the tests folder, each with `ling run`. The first failing run (non-zero exit status) stops the sequence with `tests failed`. A project with no test files but a `Cargo.toml` falls back to `cargo test`; with no manifest at all it runs `cargo test`.

A test is an ordinary program: it needs an entry point and fails by exiting with an error. The generated test program binds `test` (any `bind` of a `do` block counts as the entry).

```
OUTPUT
$ lingfu test
Testing Ling project...
  test: ...\demo\tests\test.ling
All mirrors are clear
Ling tests passed for demo
All tests passed!
```

Because there is no assertion builtin, make a test fail deliberately with a runtime error, for example by calling `read_file` on a missing path or an unknown function under `--interp`.

`lingfu clean`

Deletes the project's build folder and lock folder (`build` and `.lock`, or their localised names) and, when a `Cargo.toml` or `target/` exists, runs `cargo clean`.

```
OUTPUT
$ lingfu clean
  removed ...\demo\build
  removed ...\demo\.lock
Clean!
```

`lingfu tree`, `lingfu manifest`

`lingfu tree` prints the project's directory tree. `lingfu manifest` prints the path of the manifest followed by its contents (or the `Cargo.toml` if no Ling manifest exists).

```
OUTPUT
$ lingfu tree
Project tree:
demo
├── .gitignore
├── Ling.toml
├── README.md
├── lexicons
├── └── en.ling
├── ling.ignore
├── src
├── └── start.ling
└── tests
└── └── test.ling
```

The branch drawing is the tool's own: it prefixes every child with the parent's connector, so nested entries show two connectors on one line.

CHAPTER 14

Window, Frame Loop, Input and Camera

This chapter covers everything a graphical Ling program needs before it draws a single shape: opening a window, running the frame loop, painting in two dimensions, reading the keyboard, mouse and gamepad, and aiming the 3D camera. Shapes, textures and effects are in the next chapter.

Every entry below was checked against the interpreter's dispatch table (`src/runtime/mod.rs`). Where this book and the older draft pages in the repository disagree, this book follows the source. Entries marked "Aliases" list the Chinese (zh), Japanese (ja), Korean (ko) and Thai (th) names accepted by the interpreter in addition to the English name; the interpreter also accepts Arabic, Hebrew, Persian, Urdu, French, German and Russian names for many builtins, which are not listed here.

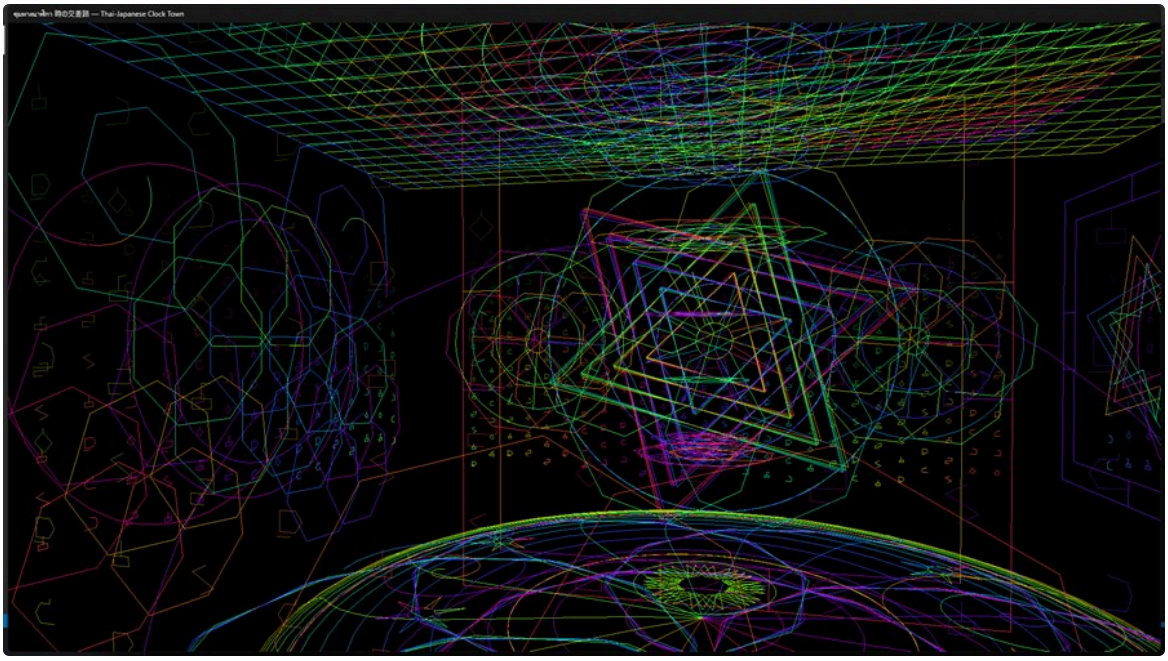


Figure 10.1. A scene built only from the builtins in this chapter and the next, drawn live in a Ling window. Source: `examples/3d-rooms/3d-thai-japanese-clocktown.ling` *in the Ling repository.*

The frame loop

A Ling graphics program has one shape. Open a window, then repeat four steps until the window closes: clear, draw, `present`, repeat.

```
GUI
bind start = do {
    open_window(800, 600, "Frame loop")
    bind frame = 0
    while window_is_open() {
        fill(10, 12, 30)                    # 1. clear to a background colour
        set_color(255, 200, 80)             # 2. draw
        draw_disc(400 + 200 * sin(frame * 0.05), 300, 40)
        present()                           # 3. flush and show
        bind frame = frame + 1
    }
}
```

Three facts about this loop are worth memorising.

1. `window_is_open()` becomes false when the user closes the window **or presses Escape**. You do not need to handle Escape yourself.
2. `present()` is the only call that puts pixels on screen. It also flushes everything queued by 3D builtins, runs the post-processing chain, and reads the mouse. A frame without `present()` is invisible.
3. There is no delta-time builtin. Use `time_now()` (seconds since the program started) and subtract the previous value, or use `frame_count()`.

Colour arguments are plain numbers from 0 to 255. They are **not clamped**: the interpreter packs `(r << 16) | (g << 8) | b`, so a value above 255 spills into the neighbouring channel. Negative numbers become 0.

Calling a function that does not exist (for example `draw_rect`, which Ling does not have) is silently ignored. If a drawing call seems to do nothing, check the spelling against this chapter first.

Coordinate conventions

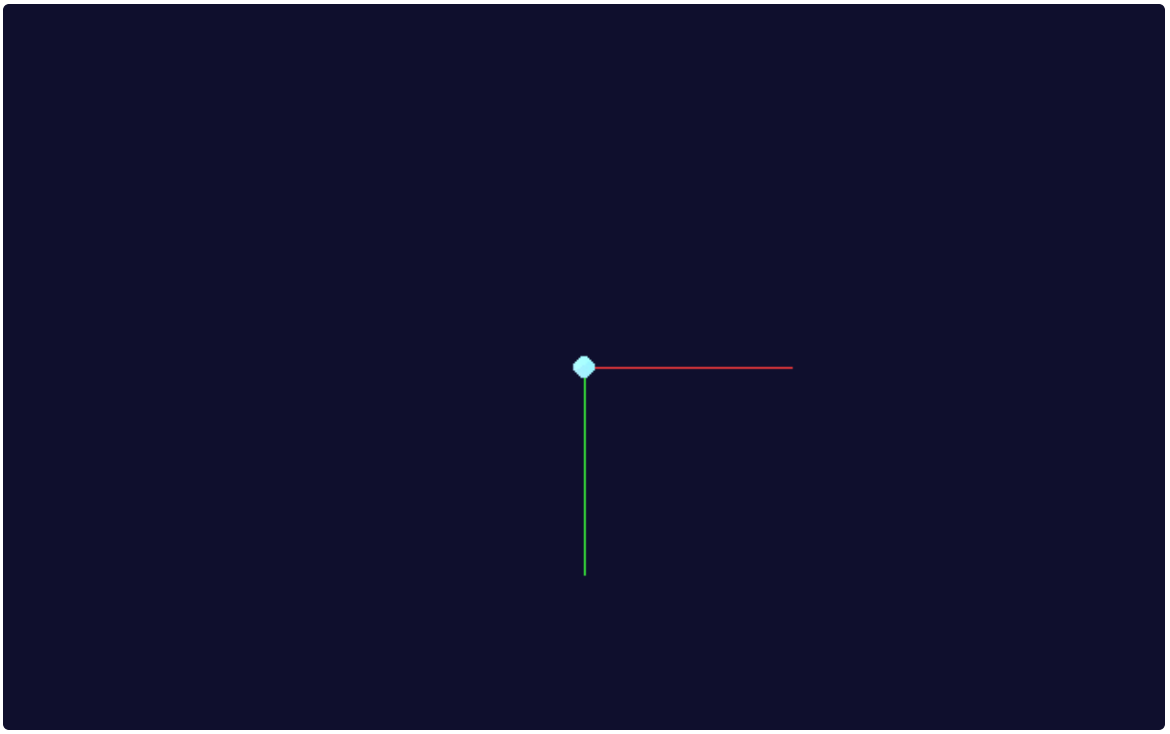


Figure 10.2. A red line along +x, a green line along +y and a blue line along +z, all starting at the white sphere. The green line points down the screen. The blue line points straight into the screen, so it is seen end-on.

Space	Origin	+x	+y	+z
Screen (2D)	top-left pixel	right	**down**	not used
World (3D)	camera pivot	right	**down**	away from the camera

Because +y is down, "above" is negative y. A ceiling lives at `y = -3`, a floor at `y = +3`. Angles in `draw_arc` also run clockwise on the screen for the same reason.

The default projection centres the view on the window, uses a focal length equal to the window height in pixels, and adds a fixed depth offset `zdist = 5` before the perspective divide. A point at `(x, y, z)` with the camera at the origin lands at

```
screen_x = width/2  + height * x / (z + zdist)
screen_y = height/2 + height * y / (z + zdist)
```

Summary: window and frame

Builtin	Purpose
`open_window(w, h, title, resizable)`	Open a window
`open_fullscreen(title, w, h)`	Open a borderless full-screen window
`open_fullscreen_scaled(title, scale)`	Full screen at 1/scale resolution
`window_is_open()`	Loop condition
`present()`	Finish the frame
`quit()`	Close the window from script
`get_width()`, `get_height()`	Framebuffer size
`set_fps(n)`, `set_vsync(on)`	Frame pacing
`monitor_width()`, `monitor_height()`, `monitor_refresh()`, `monitor_info()`	Physical display
`flush_3d()`, `flush_post()`	Mid-frame flush
`wait_window()`	Keep the last frame on screen
`screenshot(tag)`, `record_frame()`	Save the framebuffer
`frame_count()`, `time_now()`	Clocks

Window and frame entries

`open_window(w, h, title, resizable)`

Opens the program's window and allocates its framebuffer.

- **Aliases:** en `gfx_window`; zh `开窗`; ja `ウィンドウ開く`; ko `창열기`; th `เปิดหน้าต่าง`
- **Parameters:** `w` width in pixels (default 800); `h` height in pixels (default 600); `title` window title string (default `"Ling"`); `resizable` optional, `1` makes the window resizable and maximisable (default 0).
- **Returns:** nothing.
- **Notes:** The numeric arguments come first and the title third. When the window is resizable the framebuffer follows the client size from the next frame on, so read `get_width()` and `get_height()` each frame. Opening the window also resets the 3D projection to its defaults (centre of window, focal length equal to the height, `zdist` 5). If the window cannot be created the program stops with `cannot open window`.

```
GUI
open_window(1024, 640, "My game", 1)
```

`open_fullscreen(title, w, h)`

Opens a borderless window covering the whole primary monitor.

- **Aliases:** en `fullscreen`; zh `全屏`; ja `全画面`; ko `전체화면`; th `เปิดหน้าต่างเต็มจอ`
- **Parameters:** `title` window title (default `"Ling"`); `w`, `h` optional pixel size (default: the monitor's resolution).
- **Returns:** nothing.
- **Notes:** The title is the **first** argument here, unlike `open_window`. On Windows the window is topmost and keeps keyboard focus; `examples/geometry/shapes_demo.ling` opens its window this way and pairs it with `capture_mouse()`.

```
GUI
open_fullscreen("Cathedral")
```

`window_is_open()`

True while the window is open; false once it has been closed, `quit()` was called, or Escape was pressed.

- **Aliases:** en `gfx_is_open`, `is_open`; zh `窗开`; ja `開いている`; ko `창열림`; th `หน้าต่างเปิดอยู่`
- **Parameters:** none.
- **Returns:** boolean.
- **Notes:** Before any window is opened it returns false, so a loop guarded by it simply does not run.

```
GUI
while window_is_open() {
    fill(0, 0, 0)
    present()
}
```

`present()`

Ends the frame: draws every queued 3D primitive, applies post-processing, copies the framebuffer to the window and samples the mouse.

- **Aliases:** en `gfx_present`, `show`; zh `呈现`, `显`; ja `表示`; ko `표시`; th `แสดงผล`
- **Parameters:** none.
- **Returns:** nothing.
- **Notes:** Depth-sorted 3D geometry (`cube`, `vtex_*`, `draw_line_3d`, and the others in the next chapter) is only rasterised here, so those calls can be made in any order within the frame. Direct 2D calls (`draw_line`, `triangle`, `draw_disc`, `tex_*`) write to the framebuffer immediately, before the queue is flushed, so 3D geometry always lands on top of them. To draw 2D interface over a 3D scene use `flush_3d()` first. Click detection for the `ui_*` widgets depends on `present()` being called once per frame.

```
GUI
present()
```

`quit()`

Asks the window to close, exactly like pressing Escape. Useful for an on-screen exit button.

- **Aliases:** English only; `exit_game` and `close_window` are accepted synonyms.
- **Parameters:** none.
- **Returns:** nothing.

```
GUI
if ui_button(20, 20, 100, 32) == 1 { quit() }
```

`get_width()` and `get_height()`

The current framebuffer width and height in pixels.

- **Aliases, `get_width`:** zh `宽`; ja `幅取得`; ko `너비`; th `ความกว้าง`
- **Aliases, `get_height`:** zh `高`; ja `高取得`; ko `높이`; th `ความสูง`
- **Parameters:** none.
- **Returns:** number.

- **Notes:** For a fixed-size window these equal the arguments of `open_window`. For a resizable window they change when the user resizes it.

```ling
GUI
bind cx = get_width() / 2
bind cy = get_height() / 2
```

`time_now()`

Seconds elapsed since the program started, as a fractional number.

- **Aliases:** zh `当前时间`; ja `経過時間`; ko `현재시간`; th `เวลาปัจจุบัน`
- **Parameters:** none.
- **Returns:** number.
- **Notes:** The idiom for a frame-time delta is to remember the previous value. (`epoch_now()` returns wall-clock seconds since 1970.)

```ling
GUI
bind last = time_now()
while window_is_open() {
    bind now = time_now()
    bind dt = now - last
    bind last = now
    fill(0, 0, 0)
    present()
}
```

2D drawing

All the 2D builtins use screen pixels with the origin at the top-left and +y down. They draw in the current **pen colour**, which stays in effect until you change it. `present()` is not needed for the pixels to be written, only for them to be shown.

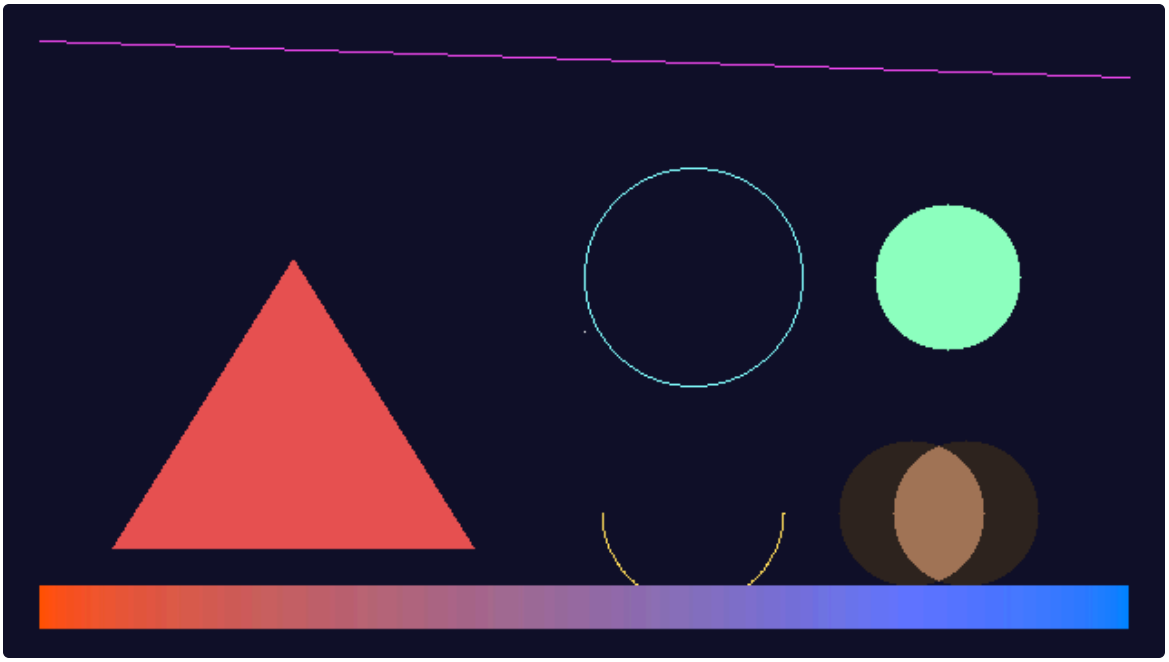


Figure 10.3. One frame of 2D drawing. The brown circles at the lower right overlap under `set_blend(1)`*, which adds colour, so the overlap is brighter. The yellow arc was drawn from angle 0 to pi and, because y points down, it is the lower half of its circle.*

Builtin	Purpose
`fill(r, g, b)`	Clear the framebuffer
`set_color(r, g, b)`, `set_color_hsl(h, s, l)`, `mix_color(...)`	Pen colour
`set_alpha(a)`, `set_blend(mode)`, `set_antialias(on)`	Pen state
`draw_pixel`, `draw_line`, `triangle`	Basic primitives
`draw_circle`, `draw_disc`, `draw_arc`, `draw_poly`	Curves and polygons
`grad_rect`, `grad_triangle`	Gradient fills
`shadow_blob`, `cast_shadow`, `shadow_params`	Soft shadows
`depth_triangle`, `depth_line`	Depth-sorted 2D
`set_frame_blur(a)`	Afterimage trails

`fill(r, g, b)`

Clears the whole framebuffer to a colour and resets the depth buffer for the new frame.

- **Aliases:** en `gfx_fill`, `clear`; zh `填`, `清`; ja `塗り潰し`, `消去`; ko `지우기`, `채우기`; th `เติม`

- **Parameters:** `r`, `g`, `b` each 0 to 255 (each defaults to 0, so `fill()` clears to black).
- **Returns:** nothing.
- **Notes:** Call it first in every frame. It does not change the pen colour.

```
GUI
fill(10, 12, 30)
```

`set_color(r, g, b)`

Sets the pen colour used by every drawing call that follows. 3D shapes also use it as their base colour.

- **Aliases:** en `gfx_color`, `color`; zh `设色`; ja `色設定`; ko `색설정`; th `สีดินสอ`
- **Parameters:** `r`, `g`, `b` each 0 to 255 (default 255).
- **Returns:** nothing.

```
GUI
set_color(255, 120, 60)
draw_disc(100, 100, 30)
```

`draw_pixel(x, y)`

Sets one pixel to the pen colour. Coordinates outside the window are ignored.

- **Aliases:** en `gfx_pixel`, `pixel`; zh `画点`; ja `点描く`; ko `점그리기`; th `วาดจุด`
- **Parameters:** `x`, `y` pixel coordinates (truncated to integers).
- **Returns:** nothing.

```
GUI
draw_pixel(320, 180)
```

`draw_line(x0, y0, x1, y1)`

A one-pixel line in the pen colour.

- **Aliases:** en `gfx_line`, `line`; zh `画线`; ja `線描く`; ko `선그리기`; th `วาดเส้น`
- **Parameters:** the two end points in pixels.

- **Returns:** nothing.
- **Notes:** Honours `set_antialias` and additive blend.

```
GUI
set_color(255, 255, 255)
draw_line(0, 0, get_width(), get_height())
```

`triangle(x0, y0, x1, y1, x2, y2)`

A filled triangle in the pen colour.

- **Aliases:** en `draw_triangle`, `gfx_triangle`; zh `三角形描画`, `画三角`; ko `삼각형그리기`; th `วาดสามเหลี่ยม`
- **Parameters:** three corner points in pixels.
- **Returns:** nothing.
- **Notes:** To build rectangles and other polygons from triangles use two or more calls, or use `draw_poly`.

```
GUI
set_color(230, 90, 90)
triangle(60, 300, 160, 140, 260, 300)
```

`draw_circle(cx, cy, r)`

A circle outline.

- **Aliases:** zh `円描画`, `画圆`; ko `원그리기`; th `วาดวงกลม`
- **Parameters:** centre `cx`, `cy` and radius `r` in pixels (radius default 10).
- **Returns:** nothing.

```
GUI
set_color(90, 200, 255)
draw_circle(380, 150, 60)
```

`draw_disc(cx, cy, r)`

A filled circle.

- **Aliases:** en `draw_filled_circle`; zh `画实心圆`; ja `塗りつぶし円`; ko `원채우기`; th `วาดวงกลมทึบ`
- **Parameters:** centre and radius as for `draw_circle`.
- **Returns:** nothing.
- **Notes:** Honours the blend mode, so with `set_blend(1)` it paints a glow.

```
GUI
set_color(120, 255, 160)
draw_disc(520, 150, 40)
```

`key_down(name)`

Is the named key currently held?

- **Aliases:** zh `按键`; ja `キー押す`; ko `키누름`; th `กดค้าง`
- **Parameters:** `name` a lower-case string (see the table below).
- **Returns:** boolean. An unknown name is simply false.
- **Notes:** Returns false while another window has focus (and for a short grace period after focus returns).

Group	Names
Letters, digits	`"a"` to `"z"`, `"0"` to `"9"`
Arrows and navigation	`"left"`, `"right"`, `"up"`, `"down"`, `"home"`, `"end"`, `"pageup"`, `"pagedown"`, `"insert"`, `"delete"`
Editing	`"space"`, `"enter"`, `"escape"`, `"tab"`, `"backspace"`, `"capslock"`
Modifiers	`"lshift"`, `"rshift"`, `"lctrl"`, `"rctrl"`, `"lalt"`, `"ralt"`
Function keys	`"f1"` to `"f12"`
Number pad	`"numpad0"` to `"numpad9"` (or `"kp0"` to `"kp9"`), `"numpad+"`, `"numpad-"`, `"numpad*"`, `"numpad/"`
Punctuation	`"minus"`, `"equals"`, `"comma"`, `"period"`, `"slash"`, `"semicolon"`, `"quote"`, `"lbracket"`, `"rbracket"`, `"backslash"`, `"grave"` (the symbols `-` `=` `,` `.` `/` `;` `'` `[` `]` `` ` `` also work)

```
GUI
bind x = 400
while window_is_open() {
    if key_down("left")  { bind x = x - 4 }
    if key_down("right") { bind x = x + 4 }
    fill(0, 0, 0)
    set_color(255, 255, 255)
    draw_disc(x, 300, 20)
    present()
}
```

`key_pressed(name)`

True only on the single frame in which the key went down, so holding the key does not repeat.

- **Aliases:** zh `键按` ; ja `キー押した` ; ko `키눌림` ; th `กดปุ่ม`
- **Parameters:** `name` as for `key_down`.
- **Returns:** boolean.
- **Notes:** Use it for toggles, menu selection and firing once per press. Pressing Start on a gamepad counts as `"enter"`.

```
GUI
bind paused = 0
while window_is_open() {
    if key_pressed("p") { bind paused = 1 - paused }
    fill(0, 0, 0)
    present()
}
```

`mouse_x()` and `mouse_y()`

The cursor position inside the window, in pixels, clamped to the window.

- **Aliases:** none (English names only)
- **Parameters:** none.
- **Returns:** number.

- **Notes:** Both return 0 until the window exists. `ui_hot` and the `ui_*` controls read the mouse themselves, so you rarely need these for interface work.

```
GUI
set_color(255, 255, 255)
draw_disc(mouse_x(), mouse_y(), 6)
```

`mouse_down()`, `mouse_down_right()`, `mouse_down_middle()`

True while the left, right or middle button is held.

- **Aliases,** `mouse_down_right`: th `เมาส์ขวา`
- **Aliases,** `mouse_down_middle`: th `เมาส์กลาง`
- **Parameters:** none.
- **Returns:** boolean.
- **Notes:** These are level checks. For a click edge, remember last frame's value.

```
GUI
bind was = 0
while window_is_open() {
    bind now = 0
    if mouse_down() { bind now = 1 }
    if now == 1 and was == 0 { println("click at " + str(mouse_x()) + "," +
str(mouse_y())) }
    bind was = now
    fill(0, 0, 0)
    present()
}
```

`set_camera(cry, sry, crx, srx)`

Sets the camera orientation from precomputed cosines and sines.

- **Aliases:** zh `设置摄像机`, `设镜`; ja `カメラ設定`; ko `카메라설정`; th `ตั้งกล้อง`
- **Parameters:** `cry`, `sry` cosine and sine of the yaw angle (defaults 1 and 0); `crx`, `srx` cosine and sine of the pitch angle (defaults 1 and 0).
- **Returns:** nothing.

- **Notes:** Pass the four numbers, not the angles. `set_camera(1, 0, 1, 0)` is the neutral view straight along +z.

```
bind start = do {
    bind a = 0.3
    set_camera(cos(a), sin(a), 1, 0)
    println("yaw+  : " + str(project_3d(0, 0, 5)))
    set_camera(1, 0, cos(a), sin(a))
    println("pitch+: " + str(project_3d(0, 0, 5)))
}
```

```
OUTPUT
yaw+  : [796.77392578125, 540, 4.776682376861572]
pitch+: [960, 376.7739562988281, 4.776682376861572]
```

With no window open the projection keeps its defaults for a 1920 by 1080 frame (centre 960, 540). A positive yaw moved the point left of centre; a positive pitch moved it up.

`set_camera_pos(x, y, z)`

Moves the camera pivot to a world position.

- **Aliases:** zh `镜坐标` ; ja `カメラ座標` ; ko `카메라좌표` ; th `ตั้งตำแหน่งกล้อง`
- **Parameters:** `x` , `y` , `z` in world units (default 0).
- **Returns:** nothing.
- **Notes:** The audio listener follows the camera position when audio is active. Remember y is down: to raise the camera, decrease y.

```
GUI
set_camera_pos(0, -2, -6)       # two units up, six back
```

A free-fly camera

The whole chapter in one program: mouse look, WASD walking and a spinning object to look at. The forward vector `(sin yaw, cos yaw)` keeps the walking direction aligned with the view.

GUI

```
bind start = do {
    open_window(900, 600, "Fly")
    capture_mouse()
    set_depth_test(1)
    set_ambient(0.3)
    bind yaw = 0.0
    bind pitch = 0.0
    bind px = 0.0
    bind py = 0.0
    bind pz = -4.0
    bind t = 0.0
    while window_is_open() {
        bind yaw = yaw + mouse_dx() * 0.003
        bind pitch = pitch + mouse_dy() * 0.003
        if pitch > 1.5 { bind pitch = 1.5 }
        if pitch < -1.5 { bind pitch = -1.5 }
        bind fx = sin(yaw) * 0.08
        bind fz = cos(yaw) * 0.08
        if key_down("w") { bind px = px + fx  bind pz = pz + fz }
        if key_down("s") { bind px = px - fx  bind pz = pz - fz }
        if key_down("d") { bind px = px + fz  bind pz = pz - fx }
        if key_down("a") { bind px = px - fz  bind pz = pz + fx }
        set_camera(cos(yaw), sin(yaw), cos(pitch), sin(pitch))
        set_camera_pos(px, py, pz)
        fill(8, 10, 24)
        clear_lights()
        add_light(-3, -6, 0, 1.0, 0.95, 0.85, 1.0, 0)
        vtex_grid(0, 2, 8,  1,0,0,  0,0,1,  20, 20, 1, 1,  t, 0)
        set_color(240, 150, 110)
        cube(0, 0, 8, 1,1,1, t * 0.5, t, 0, 0)
        present()
        bind t = t + 0.02
    }
}
```

Lighting set-up

Lights belong to the camera chapter because the same per-frame discipline applies: clear, add, draw. How the lights shade a surface is explained in the next chapter.

`set_ambient(level)`

Sets the ambient light that fills everything regardless of the lights.

- **Aliases:** zh `环境光` ; ja `環境光設定` ; ko `환경광설정` ; th `ตั้งแสงรอบข้าง`
- **Parameters:** `level` from 0 to 1 (default 0.15).
- **Returns:** nothing.
- **Notes:** The default shading model also tints unlit areas with a cool shadow colour; see `set_shadow_color` .

```
GUI
set_ambient(0.25)
```

`add_light(x, y, z, r, g, b, intensity, radius)`

Adds a coloured point light at a world position for this frame.

- **Aliases:** zh `加灯` ; ja `ライト追加` ; ko `조명추가` ; th `เพิ่มแสง`
- **Parameters:** `x` , `y` , `z` position (defaults 0, -3, 3). `r` , `g` , `b` colour, either 0 to 1 or 0 to 255 (if any channel exceeds 1.5 all three are treated as 0 to 255). `intensity` brightness multiplier (default 1). `radius` distance at which the light fades to nothing; 0 means no falloff (default 0).
- **Returns:** nothing.
- **Notes:** The argument order is position, **colour**, intensity, radius. Some older pages in the repository show intensity before colour; the source is as listed here. Lights accumulate until you call `clear_lights()` , so clear and re-add them each frame.

```
GUI
clear_lights()
add_light(0, -5, 4,  1.0, 0.85, 0.6,  1.2, 20)
```

`clear_lights()`

Removes every light.

- **Aliases:** zh `清灯` ; ja `ライト消去` ; ko `조명초기화` ; th `ล้างแสง`
- **Parameters:** none.
- **Returns:** nothing.

```
GUI
clear_lights()
```

CHAPTER 15

Shapes, Vector Textures, Pixel Textures and Effects

This chapter lists the builtins that put things into the 3D scene: the parametric solids (`cube`, `sphere`, `gear`, and about two dozen more), the low-level triangle and line calls, the `vtex_*` vector patterns, the `tex_*` pixel textures, and the shading, post-processing and light-volume effects that give the output its "anime holographic" look.

Everything here assumes the window, frame loop and camera of the previous chapter. In the short fragments, the names `n` and `frame` stand for a frame counter that the surrounding loop maintains. All snippets in this chapter need an open window; they are marked `ling-gui` and were checked against the interpreter's dispatch table. The pictures were produced by running the programs shown (or close variants) and calling `screenshot()`.

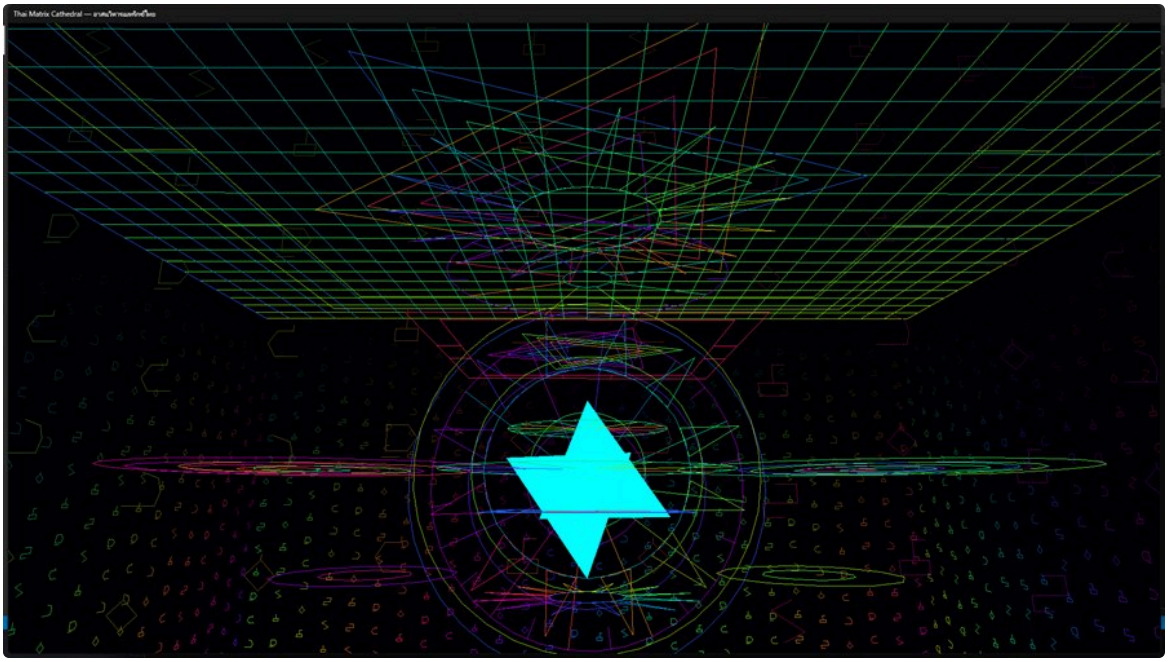


Figure 11.1. A room assembled from `vtex_*` patterns and a few solid shapes, drawn live. Source: `examples/3d-rooms/3d-room.ling` in the Ling repository.

Part 1. Primitive solids

Every solid shares one call shape:

```
name(cx, cy, cz,  sx, sy, sz,  rx, ry, rz,  mode,  e0, e1, e2)
```

Group	Meaning
`cx, cy, cz`	World position of the centre
`sx, sy, sz`	Scale per axis (default 1 each). Each shape is built about one unit in radius (the cube spans -1 to 1), so the scale is roughly the **half-extent** or radius
`rx, ry, rz`	Euler rotation in radians (default 0), applied in the order X, then Y, then Z, after scaling
`mode`	`0` filled (default), `1` wireframe, `2` both
`e0, e1, e2`	Shape-specific extras. `0` or omitted selects the default; a value that is not positive also selects the default

Only the position needs to be given; every other argument has a default. Trailing arguments may be dropped.

The pen colour (`set_color`) is the base colour for the fill and the colour of the wireframe. Solids are lit by `add_light` and `set_ambient`, shaded by the model described in Part 4, depth-sorted with the rest of the 3D queue, and drawn when `present()` is called.

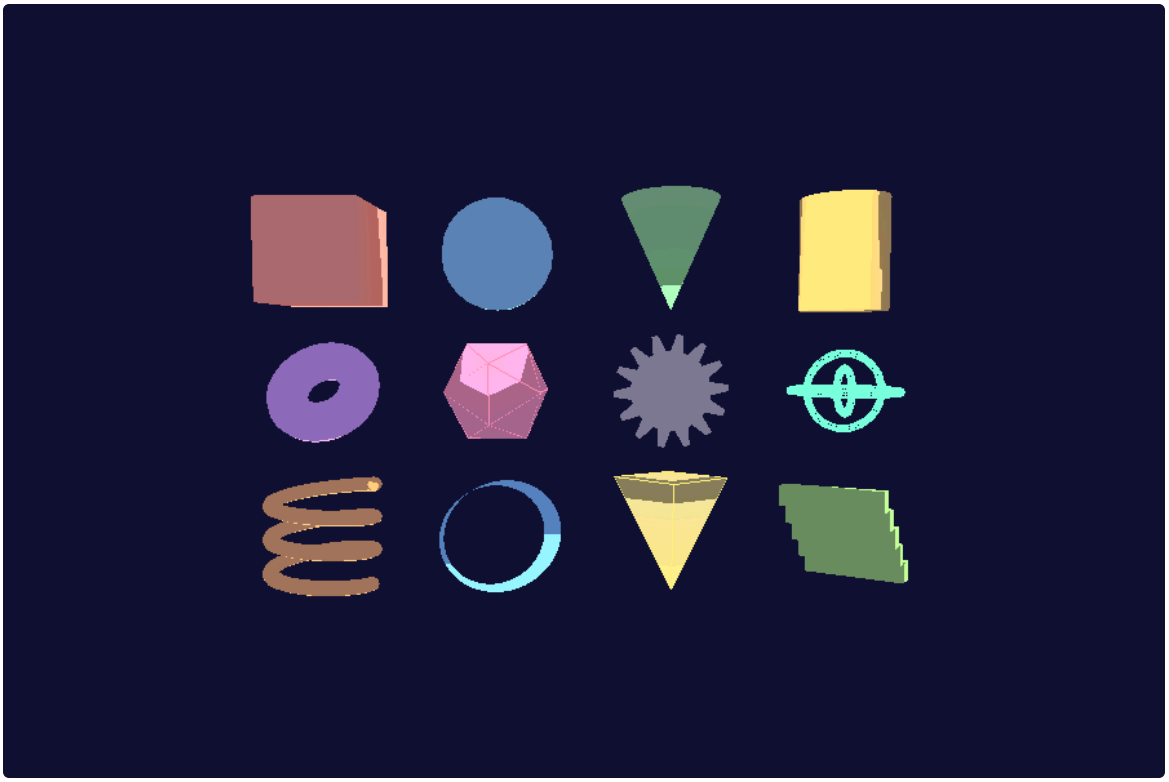


Figure 11.2. Twelve of the solids, lit by one warm light. Programs that want the pen colours to show through faithfully set `set_shade_mode(1)`, `set_rim(0, 0, 0, 0)` *and a dark* `set_shadow_color` *first; see Part 4.*

Remember the y axis points down (chapter 10). A `cone` has its apex at +**y**, so with the default camera it points down the screen; rotate by pi on x to flip it. A `gear` lies in the x-z plane, so rotate it by about 1.57 on x to face the camera.

```
GUI
bind start = do {
    open_window(900, 600, "Shapes")
    set_depth_test(1)
    set_ambient(0.25)
    set_shade_mode(1)
    set_rim(0, 0, 0, 0)
    set_shadow_color(30, 30, 50)
    bind n = 0
    while window_is_open() {
        fill(8, 10, 24)
        set_camera(1, 0, 1, 0)
        clear_lights()
        add_light(-2, -6, -2,  1.0, 0.95, 0.85,  0.6, 0)
        bind a = n * 0.03
        set_color(230, 120, 90)
        cube(-2.8, 0, 7.5,  0.8,0.8,0.8,  a, a * 0.7, 0,  0)
        set_color(120, 200, 255)
        sphere(0, 0, 7.5,  0.9,0.9,0.9,  0,0,0,  0)
        set_color(190, 190, 200)
        gear(2.8, 0, 7.5,  0.9,0.3,0.9,  1.57, a * 2, 0,  0,  14, 0.3)
        present()
        bind n = n + 1
    }
}
```

Summary: solids

Group	Builtins
Round	`cube`, `sphere`, `icosphere`, `dome`, `cylinder`, `cone`, `capsule`, `torus`
Prisms and pyramids	`pyramid`, `prism`, `frustum`
Dice	`tetrahedron`, `octahedron`, `dodecahedron`, `icosahedron`
Mechanical	`gear`, `gyro`, `arch`, `stairs`
Curves and surfaces	`helix`, `spring`, `star_prism`, `capsule_chain`, `mobius`

`cube(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode)`

An axis-aligned box. Non-uniform scales give any rectangular block.

- **Aliases:** en `box`; zh `方块`; ja `箱`; ko `상자`, `정육면체`; th `กล่อง`, `ลูกบาศก์`; zh/ja `立方体`
- **Parameters:** the common signature. No extras.
- **Returns:** nothing.
- **Notes:** The edge length is twice the scale: `cube(0,0,8, 1,1,1, ...)` is a 2-unit cube.

```
GUI
set_color(230, 120, 90)
cube(0, 0, 8,  1, 1, 1,  0, 0.6, 0,  0)
```

`sphere(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode, segments, rings)`

A UV sphere; unequal scales make an ellipsoid.

- **Aliases:** zh `球体`; ja `球`; ko `구`; th `ทรงกลม`
- **Parameters:** the common signature; `segments` around the equator (default 16, range 3 to 128); `rings` pole to pole (default 12, range 2 to 128).
- **Returns:** nothing.

```
GUI
set_color(120, 200, 255)
sphere(0, 0, 8,  1, 1, 1,  0, 0, 0,  0,  32, 24)
```

`cylinder(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode, segments)`

A capped tube. The height runs along local y from -1 to 1, so it is `2 * sy` tall.

- **Aliases:** zh `圆柱`; ja `円柱`; ko `원기둥`; th `ทรงกระบอก`
- **Parameters:** the common signature; `segments` (default 24, range 3 to 256).
- **Returns:** nothing.
- **Notes:** For a flat disc or plate use a small `sy`, for example 0.05.

```
GUI
cylinder(0, 0, 8,  0.8, 1.2, 0.8,  0, 0, 0,  0)
```

`cone(cx,cy,cz, sx,sy,sz, rx,ry,rz, mode, segments)`

A capped cone with its apex at +y and base at -y.

- **Aliases:** zh `圆锥` ; ja `円錐` ; ko `원뿔` ; th `กรวย`
- **Parameters:** the common signature; `segments` (default 24).
- **Returns:** nothing.
- **Notes:** Because +y is down the screen, an unrotated cone points downward. Use `rx = 3.14159` to point it up.

```
GUI
cone(0, 0, 8,  0.8, 1, 0.8,  3.14159, 0, 0,  0)    # apex up
```

`draw_line_3d(ax,ay,az, bx,by,bz)`

A depth-sorted line between two world points, clipped at the near plane.

- **Aliases:** en `line3d` ; zh `画3D线` ; ja `3D線描く` ; ko `3D선그리기` ; th `วาดเส้น3มิติ`
- **Parameters:** two world-space points.
- **Returns:** nothing.
- **Notes:** Shared edges between solids are de-duplicated automatically. See `set_antialias` for smooth lines.

```
GUI
set_color(255, 255, 255)
draw_line_3d(-2, 0, 8,  2, 0, 8)
```

Summary: vtex

Builtin	Pattern
`vtex_grid`	Rectangular grid
`vtex_rings`	Concentric polygons with twist
`vtex_star`	Spinning star polygon
`vtex_spiral`	Archimedean spiral
`vtex_flower`	Flower of Life
`vtex_halftone`	Cross-hatch dither fill

Builtin	Pattern
`vtex_tessellated`	Wavy triangle mesh
`vtex_hyperbolic_uv`	Poincaré-disc ceiling grid
`vtex_letter_rain`	Thai-letter matrix rain
`vtex_lotus`	Lotus petals
`vtex_chakra`	Dhamma wheel
`vtex_yantra`	Interlocked triangles in a square
`vtex_spiked_cog`	Spiked gear
`vtex_torii`	Torii gate silhouette
`vtex_pagoda`	Tiered pagoda silhouette

In the entries below, `PLANE` stands for the nine plane arguments `cx,cy,cz, ux,uy,uz, vx,vy,vz`.

`vtex_grid(PLANE, cols, rows, cell_w, cell_h, fr, hue)`

A rectangular grid of lines, centred on the plane origin.

- **Aliases:** zh `纹格`; ja `格子模様`; ko `격자무늬`; th `ลายตาราง`
- **Parameters:** `cols`, `rows` cell counts (default 10, 10); `cell_w`, `cell_h` cell size in world units (default 1, 1); `fr`, `hue` (default 0).
- **Returns:** nothing.
- **Notes:** The grid spans `cols * cell_w` by `rows * cell_h`. Colour varies along the lines.

```
GUI
vtex_grid(0, 3, 9,  1,0,0,  0,0,1,  12, 8, 1, 1,  frame, 0)
```

How a solid is lit

For each vertex the shader mixes a base colour (your pen colour) with:

1. an **ambient** term, `set_ambient`, blended with a **shadow tint** (`set_shadow_color`) wherever the surface faces away from every light;
2. each light's contribution: pen colour times a soft step ramp of the angle to the light, times the light colour, intensity and distance falloff;
3. a **rim glow** (`set_rim`), brighter where the surface turns away from the viewer; and

4. in the default `holo` mode, a faint iridescent sheen that depends on the surface normal.

The colour is interpolated across each triangle, then quantised into a few brightness **bands** per pixel. The result is smooth shading with crisp toon edges.

The defaults are deliberately stylised: a cool indigo shadow tint, a cyan rim at strength 0.6 and the holo sheen. They make unstyled scenes look cyan and washed out. For colours that match your pen faithfully, as in Figure 11.2, set `set_shade_mode(1)`, `set_rim(0, 0, 0, 0)` and a dark neutral `set_shadow_color`.

Summary: tex

Builtin	Pattern
`tex_checkerboard`	Two-colour checker
`tex_gradient`	Linear two-colour gradient
`tex_noise`	Fractal noise through a palette
`tex_spiral`	Animated spiral bands
`tex_ripple`	Concentric ripples
`tex_mandelbrot`	Mandelbrot set
`tex_julia`	Julia set
`tex_voronoi`	Voronoi cells
`tex_halftone`	Halftone dots
`tex_freq_map`	Audio-reactive spectrum

CHAPTER 16

Vector Fonts, the UI Toolkit and Dialogs

This chapter covers everything that puts text and interface on the screen: real TrueType and OpenType fonts rendered as resolution-independent vectors, the holographic stroke font, the `ui_*` toolkit of meters, gauges and controls, and the cinematic dialog box.

All of these draw into the same framebuffer as the 2D calls of chapter 10, so they belong between `fill()` and `present()`. To draw them over a 3D scene, call `flush_3d()` first.

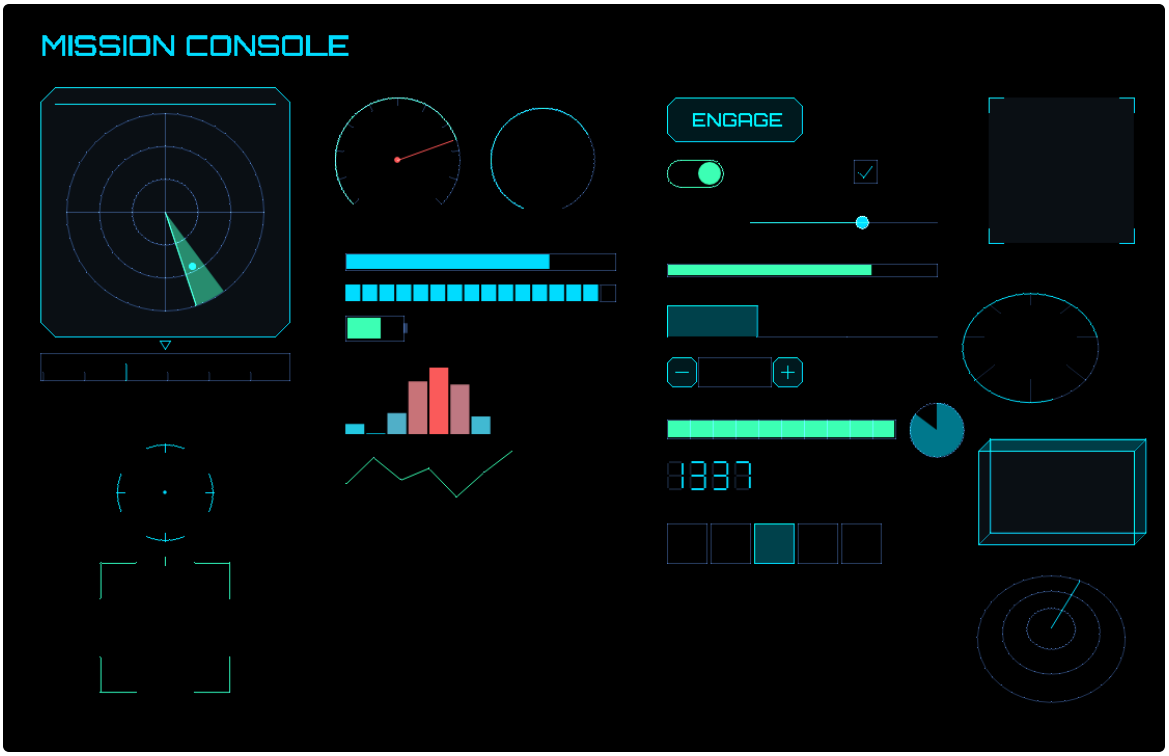


Figure 12.1. Every widget on this screen is a `ui_*` call, redrawn each frame in a few lines of Ling. Source: `examples/ui/hud_showcase.ling` in the Ling repository.

Part 1. Fonts and text

Ling draws two kinds of text.

- **Vector fonts.** `font_load` reads a TTF or OTF file; `font_text_fill`, `font_text` and `font_text_3d` draw its glyph outlines. The outlines keep their curves, so text is sharp at any size, in 2D or on a 3D plane. Latin, Chinese, Japanese, Korean and Thai all work if the font has the glyphs.
- **The built-in stroke font.** `ui_text` draws a monospaced single-stroke "holographic" face that needs no font file. It supports basic characters only.

The first time a glyph is used, its outline is extracted and cached on disk under `cache/fonts/<font>/<codepoint>.ling` (or `cache/fonts/<font>@<weight>/...` when a weight is given), relative to the directory the program runs in. Later runs read the cache.

Font loading is available in the native interpreter. In the web runtime `font_load` returns `-1`, so portable programs should check the handle and fall back to `ui_text`.

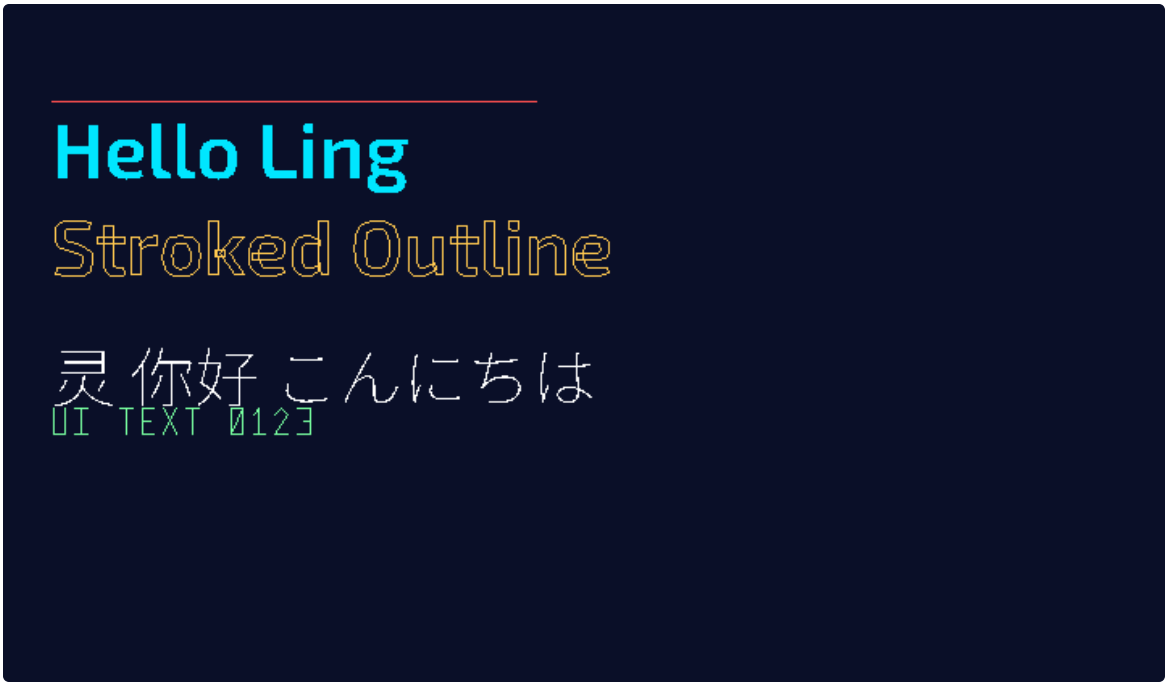


Figure 12.2. `font_text_fill` *(cyan),* `font_text` *(orange), a Chinese and Japanese line set in Noto Sans SC, and* `ui_text` *(green, bottom). The red line marks y = 60, the* `y` *given to the first line:* `y` *is the top of the text, so the glyphs hang below it.baseline.*

```
GUI
bind start = do {
    open_window(720, 420, "Fonts")
    bind f = font_load("assets/fonts/Exo2.ttf", 600)
    bind cjk = font_load("assets/fonts/NotoSansSC.ttf")
    while window_is_open() {
        fill(8, 10, 24)
        set_color(0, 230, 255)
        font_text_fill(f, 30, 60, 48, "Hello Ling")
        set_color(255, 200, 80)
        font_text(f, 30, 120, 48, "Stroked Outline")
        set_color(255, 255, 255)
        font_text_fill(cjk, 30, 200, 40, "灵 你好 こんにちは")
        set_color(120, 255, 160)
        ui_text(30, 250, 16, "UI TEXT 0123")
        present()
    }
}
```

Summary: fonts

Builtin	Purpose
`font_load(path, weight)`	Load a font, get a handle
`font_text_fill(font, x, y, px, text)`	Filled glyphs
`font_text(font, x, y, px, text)`	Stroked outlines
`font_text_3d(font, ..., size, text, rows)`	Text on a 3D plane
`font_width(font, px, text)`	Measure a string
`font_advance(font, char)`	Advance of one glyph
`font_glyph_outline(font, char, tol)`	Raw outline as polylines
`font_fallback(primary, fallback, scale)`	Per-character fallback font
`set_font_antialias(on)`, `get_font_antialias()`	Glyph smoothing
`ui_text(x, y, scale, text)`	Built-in stroke font

`font_load(path, weight)`

Loads a TrueType or OpenType font and returns a handle for the other `font_*` calls.

- **Aliases:** zh `加载字体` ; ja `フォント読込` ; ko `글꼴로드` ; th `โหลดฟอนต์`
- **Parameters:** `path` a file path. It is tried as given and then relative to the directory of the running script. `weight` optional variable-font weight, a positive number such as 600 for a solid, bold look; 0 or omitted uses the font's default.
- **Returns:** a handle (0 for the first font loaded, 1 for the next, and so on), or `-1` if the font could not be read. On failure a message such as `font_load failed (path): ...` is printed to the error stream.
- **Notes:** Check for `-1` and try a second path if your program may be started from different folders. The repository ships Exo2, Orbitron, Rajdhani, Noto Sans (SC, JP, KR, Thai) and Galmuri fonts in `assets/fonts/` .

```
GUI
bind f = font_load("assets/fonts/Exo2.ttf", 600)
if f < 0 { bind f = font_load("../../assets/fonts/Exo2.ttf", 600) }
```

`font_text_fill(font, x, y, px, text)`

Draws text as solid filled glyphs in the pen colour.

- **Aliases:** zh `填充字体` ; ja `フォント塗り` ; ko `글꼴채움` ; th `เติมฟอนต์`
- **Parameters:** `font` a handle from `font_load` ; `x` , `y` the top-left of the text line in pixels. Measured on screen, capital letters start about 0.3 `px` below `y` and the baseline lies about 0.97 `px` below `y` ; `px` the font size in pixels (default 16); `text` a string.
- **Returns:** nothing.
- **Notes:** Each glyph is filled independently so counters (the hole in "o") stay open. Edges are crisp unless you call `set_font_antialias(1)` . An invalid handle draws nothing.

```
GUI
set_color(0, 230, 255)
font_text_fill(f, 30, 60, 48, "Hello Ling")
```

`font_text(font, x, y, px, text)`

Draws text as stroked outlines in the pen colour and blend mode.

- **Aliases:** zh `字体文本`; ja `フォント文字`; ko `글꼴텍스트`; th `ข้อความฟอนต์`
- **Parameters:** as for `font_text_fill`.
- **Returns:** nothing.
- **Notes:** Cheaper than filled text and good for a wireframe look. Honours `set_blend` and `set_antialias`.

```
GUI
set_color(255, 200, 80)
font_text(f, 30, 120, 48, "Stroked Outline")
```

`ui_text(x, y, scale, text)`

Draws text in the built-in holographic stroke font. No font file is needed.

- **Aliases:** zh `界面文字`; ja `UI文字`; ko `UI텍스트`; th `ข้อความหน้าจอ`
- **Parameters:** `x`, `y` the top-left of the text; `scale` the glyph height in pixels, measured: glyphs fill rows `y` to `y + scale` (default 16); `text` the string.
- **Returns:** nothing.
- **Notes:** Drawn in the pen colour. Each glyph is `scale * 0.62` wide with a gap of `scale * 0.24`, so a string is about `len * scale * 0.86` pixels wide. Letters are drawn as capitals, and characters the face does not have (including all non-ASCII text) appear as a small box. It is not a substitute for a real font.

```
GUI
set_color(120, 255, 160)
ui_text(30, 250, 16, "UI TEXT 0123")
```

Part 2. The UI toolkit

The `ui_*` builtins are an **immediate-mode** toolkit. Nothing is stored between frames: you call the widget every frame, passing in the current value, and it draws itself and, for interactive controls, returns the new value. The program owns all state in ordinary variables.

```
GUI
bind start = do {
    open_window(480, 200, "Controls")
    bind volume = 0.4
    bind enabled = 0
    while window_is_open() {
        fill(6, 10, 22)
        bind volume = ui_slider(40, 60, 280, volume, 0, 1)       # in: value, out: new
value
        bind enabled = ui_toggle(40, 100, 52, 24, enabled)
        if ui_button(40, 140, 120, 36) == 1 { println("clicked") }
        present()
    }
}
```



Figure 12.3. The widgets of this chapter on one screen: panel and frame, bar, segmented bar, progress, slider, toggle, checkbox, stepper, gauge, ring, radar, tabs, health bar, cooldown, counter, slot grid, d-pad and button.

Conventions shared by every widget:

- **Coordinates** are screen pixels, origin at the top-left, y down. Rectangular widgets take `x, y, w, h`; circular ones take a centre `cx, cy` and a radius `r`.
- **Colours** come from the current **theme** (below). Most widgets accept an optional trailing `r, g, b` triple (0 to 255) that overrides their main colour for one call; the position of that triple is given in each entry. Pass all three or none.
- **Widgets draw no text.** A button has no label, a tab strip has no captions and a stepper shows no number. Draw labels yourself with `ui_text` or `font_text_fill`. This keeps the toolkit independent of fonts.
- **Interaction** reads the left mouse button and the cursor. A control returns its new value; a button returns 1 on the single frame the mouse goes down over it. Because that edge is tracked across frames, `present()` must be called once per frame.
- **Blend.** Widgets honour the current blend mode, so `set_blend(1)` makes them glow.

Summary: UI toolkit

Group	Builtins
Theme and layout helpers	`ui_theme`, `ui_theme_colors`, `ui_hot`, `ui_frame`, `ui_bevel`, `ui_panel`, `ui_scanlines`
HUD	`ui_radar`, `ui_compass`, `ui_reticle`, `ui_target`
Meters	`ui_bar`, `ui_segbar`, `ui_gauge`, `ui_ring`, `ui_vu`, `ui_spark`, `ui_battery`, `ui_progress`
Controls	`ui_button`, `ui_toggle`, `ui_slider`, `ui_checkbox`, `ui_tabs`, `ui_stepper`, `ui_tooltip`
Game UI	`ui_healthbar`, `ui_cooldown`, `ui_counter`, `ui_minimap`, `ui_dpad`, `ui_slotgrid`, `ui_vignette`
Pseudo-3D	`ui_gauge3d`, `ui_panel3d`, `ui_radar3d`
Sound	`ui_sound`

`ui_bar(x, y, w, h, value, max, r, g, b)`

A horizontal fill bar.

- **Aliases:** zh `进度条`; ja `バー`; ko `막대`; th `แถบ`
- **Parameters:** the rectangle (default 160 by 16); `value` the current value (default 0); `max` the value that fills the bar (default 1); optional fill colour at the seventh position.
- **Returns:** nothing.

```
GUI
ui_bar(40, 70, 280, 14, 70, 100)       # 70 percent
```

`ui_button(x, y, w, h, r, g, b)`

A bevelled button that highlights on hover and darkens when pressed. It draws no label.

- **Aliases:** zh `按钮` ; ja `ボタン` ; ko `버튼` ; th `ปุ่ม`
- **Parameters:** the rectangle (default 120 by 40); optional colour at the fifth position.
- **Returns:** `1` on the one frame the left mouse button goes down over it, otherwise `0` .
- **Notes:** Draw a caption with `font_text_fill` or `ui_text` . Compare against 1: `if ui_button(...) == 1 { ... }` .

```
GUI
if ui_button(20, 400, 120, 36) == 1 { println("engage") }
set_color(220, 240, 255)
bind lw = font_width(ui, 18, "ENGAGE")
font_text_fill(ui, 20 + (120 - lw) / 2, 400 + 18 - 0.64 * 18, 18, "ENGAGE")   # centred
in the 120 by 36 button
```

`ui_slider(x, y, w, value, min, max, r, g, b)`

A horizontal slider.

- **Aliases:** zh `滑块` ; ja `スライダー` ; ko `슬라이더` ; th `แถบเลื่อน`
- **Parameters:** `x, y` the left end of the track; `w` its width (default 160); `value` the current value; `min` , `max` the range (default 0 and 1); optional colour at the seventh position. There is no height parameter; the grab area is 20 pixels tall.
- **Returns:** the new value. While the button is held over the slider, the value follows the cursor.

```
GUI
bind volume = ui_slider(40, 156, 280, volume, 0, 1)
```

Summary: dialogs

Builtin	Purpose
`dialog_show(text, cps)`	Start a dialog
`dialog_step(dt)`	Advance the typewriter
`dialog_advance()`	Reveal the page, or go to the next page, or close
`dialog_active()`	Is a dialog open?
`dialog_typing()`	Is text still being revealed?
`dialog_close()`	Close it now
`dialog_color(role, r, g, b)`	Recolour a text role
`dialog_draw(x, y, w, h, font)`	Draw the box and text

CHAPTER 17

Animation: Anima

Anima is Ling's animation system. It rests on a simple idea: any moving thing, whether a 2D drawing, a 3D mesh, a swarm of particles or a handful of numbers, is driven by small **drivers** that you call every frame with the current time. A driver is a function from time (and a few settings) to a number. You feed the number into a size, a position, an angle or a pattern argument.

Anima divides its drivers along one axis that it calls **temperament**:

- **Organic** drivers (the Chinese character 灵, *ling*, "spirit") produce lifelike, slightly soft motion: breathing, swaying, walking, springy follow-through, reaching with two-bone inverse kinematics.
- **Mechanical** drivers (机, *ji*, "machine") produce exact, rigid motion: meshed gears, cams, slider-cranks, rack and pinion.

A character can mix both: a creature with a breathing chest and a mechanical jaw.

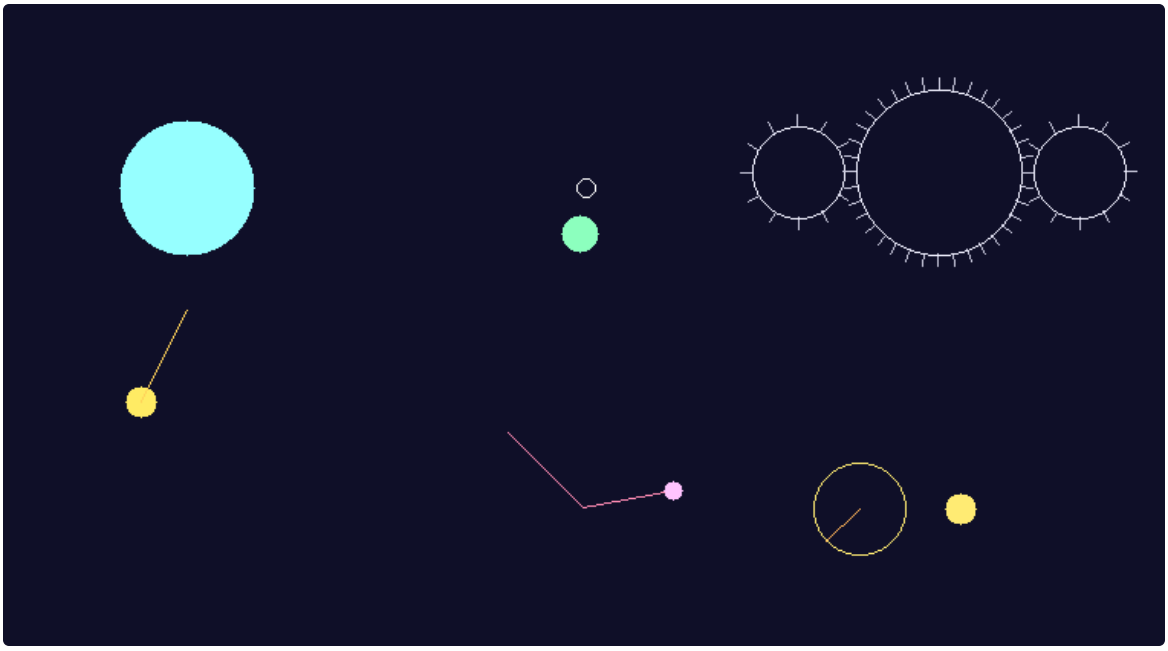


Figure 13.1. Every motion here is one Anima call per frame. Upper left: `breathe`. Below it: `gait_swing` and `gait_lift` moving a foot. Centre: `spring_to` chasing a moving target (the white ring). Lower centre: `ik2` bending a two-bone arm toward a target. Upper right: `gear_train` with 12, 36 and 12 teeth. Lower right: `piston` driven by a crank angle.

How to use a driver

Drivers are **pure functions**: they keep no hidden state, so calling one twice with the same arguments gives the same answer. Two consequences follow.

1. For time-based drivers (`breathe`, `wobble`, `gait_*`) you pass the current time yourself, usually the output of `time_now()` or a variable you add `1/60` to each frame. Nothing is "playing" behind your back.
2. For dynamic drivers (`spring_to`) the state is returned to you and you pass it back in next frame. The program owns the state.

Angles are in radians. Lengths and offsets are in whatever units your drawing uses.

```
fn r3(x: number) -> number {
    round(x * 1000) / 1000
}

bind start = do {
    println("tween(0, 10, 0.25)             = " + str(tween(0, 10, 0.25)))
    println("tween(0, 10, 2.0)              = " + str(tween(0, 10, 2.0)))
    println("tween_ease cubic_in at 0.5     = " + str(r3(tween_ease(0, 100, 0.5,
"cubic_in"))))
    println("tween_ease bounce_out at 0.5   = " + str(r3(tween_ease(0, 100, 0.5,
"bounce_out"))))
    println("breathe(0.25, 1, 0.2)          = " + str(r3(breathe(0.25, 1, 0.2))))
    println("wobble(0.25, 1, 2, 0)          = " + str(r3(wobble(0.25, 1, 2, 0))))
    println("gait_phase(0.3, 2)             = " + str(r3(gait_phase(0.3, 2))))
    println("gait_swing(0.125, 1, 30)       = " + str(r3(gait_swing(0.125, 1, 30))))
    println("gait_lift(0.125, 1, 10)        = " + str(r3(gait_lift(0.125, 1, 10))))
    bind sp = spring_to(0, 0, 1, 120, 14, 0.0166667)
    println("spring_to one step             = [" + str(r3(sp[0])) + ", " +
str(r3(sp[1])) + "]")
    bind a = ik2(1, 1, 1, 1)
    println("ik2(1, 1, 1, 1)                = [" + str(r3(a[0])) + ", " + str(r3(a[1]))
+ "]")
    println("gear_couple(3, 12, 36)         = " + str(r3(gear_couple(3, 12, 36))))
    println("gear_train(1, [12, 36, 12])    = " + str(gear_train(1, [12, 36, 12])))
    println("cam_lift(3.14159265, 2)         = " + str(r3(cam_lift(3.14159265, 2))))
    println("piston(0, 1, 3)                = " + str(r3(piston(0, 1, 3))))
    println("rack(2, 0.5)                   = " + str(rack(2, 0.5)))
    println("ease(quad_in, 0.5)             = " + str(ease("quad_in", 0.5)))
    println("ease(bounce, 0.5)              = " + str(r3(ease("bounce", 0.5))))
}
```

```
OUTPUT
tween(0, 10, 0.25)            = 2.5
tween(0, 10, 2.0)             = 10
tween_ease cubic_in at 0.5    = 12.5
tween_ease bounce_out at 0.5  = 76.563
breathe(0.25, 1, 0.2)         = 1.2
wobble(0.25, 1, 2, 0)         = 2
gait_phase(0.3, 2)            = 0.6
gait_swing(0.125, 1, 30)      = 21.213
gait_lift(0.125, 1, 10)       = 7.071
spring_to one step            = [0.033, 2]
ik2(1, 1, 1, 1)               = [1.571, -1.571]
gear_couple(3, 12, 36)        = -1
gear_train(1, [12, 36, 12])   = [1, -0.33333334326744 08, 1]
cam_lift(3.14159265, 2)        = 2
piston(0, 1, 3)               = 4
rack(2, 0.5)                  = 1
ease(quad_in, 0.5)            = 0.25
ease(bounce, 0.5)             = 0.766
```

The program above defines a small helper `r3` to round results to three decimals, because the drivers work in single precision and print long fractions. The output was produced by running it.

Summary: Anima builtins

Builtin	Temperament	Purpose
`tween(a, b, t)`	neutral	Linear interpolation
`tween_ease(a, b, t, curve)`	neutral	Eased interpolation
`ease(curve, t)`	neutral	Evaluate an easing curve
`breathe(t, rate, depth)`	organic	Breathing scale
`wobble(t, freq, amp, phase)`	organic	Sway or jiggle
`gait_phase(t, speed)`	organic	Walk-cycle phase
`gait_swing(t, speed, stride)`	organic	Leg swing
`gait_lift(t, speed, height)`	organic	Foot lift

Builtin	Temperament	Purpose
`spring_to(pos, vel, target, k, d, dt)`	organic	Spring step
`ik2(l1, l2, tx, ty)`	organic	Two-bone inverse kinematics
`gear_couple(angle, teeth_in, teeth_out)`	mechanical	Two meshed gears
`gear_train(angle, teeth)`	mechanical	A chain of meshed gears
`cam_lift(angle, lift)`	mechanical	Cam follower
`piston(angle, crank, rod)`	mechanical	Slider-crank
`rack(angle, radius)`	mechanical	Rack and pinion

Easing curves

Two builtins evaluate easing curves, and they use **different sets of names**. Both ignore case; unknown names fall back as noted.

`tween_ease` (Anima curves). Names may be written with or without underscores. Unknown names give `linear`.

Family	Names
Basic	`linear`, `step`
Quadratic	`quad_in`, `quad_out`, `quad_in_out`
Cubic	`cubic_in`, `cubic_out`, `cubic_in_out` (also `smooth`)
Sine	`sine_in`, `sine_out`, `sine_in_out`
Exponential	`expo_in`, `expo_out`, `expo_in_out`
Elastic	`elastic_in`, `elastic_out` (also `elastic`)
Back	`back_in`, `back_out`, `back_in_out` (also `overshoot`)
Bounce	`bounce_in`, `bounce_out` (also `bounce`)

`ease` (the UI toolkit's curves). Dashes and underscores are interchangeable. Unknown names give `quad_inout`.

Name	Meaning
`linear`	Straight line
`quad_in`, `ease_in`	Slow start
`quad_out`, `ease_out`	Slow finish
`quad_inout`, `ease`, `smooth`	Slow start and finish (the default)

Name	Meaning
`cubic_in`, `cubic_out`, `cubic_inout`	Stronger versions
`quart_out`	Very fast start, long slow finish
`sine`, `sine_inout`	Gentle sine in-out
`expo`, `expo_out`	Exponential decay
`back`, `back_out`	Overshoots, then settles
`elastic`, `elastic_out`	Springy overshoot
`bounce`, `bounce_out`	Bounces at the end

Note that `tween_ease` spells the in-out forms `quad_in_out` while `ease` spells them `quad_inout`; since `tween_ease` removes underscores before matching, both spellings work there.

Interpolation entries

`tween(a, b, t)`

Linear interpolation from `a` to `b`.

- **Aliases:** zh `补间`; ja `補間`; ko `트윈`; th `แทรกค่า`
- **Parameters:** `a` start value; `b` end value; `t` progress, **clamped** to the range 0 to 1.
- **Returns:** `a + (b - a) * t`.
- **Notes:** Because `t` is clamped, you can keep increasing it past 1 and the value stays at `b`.

```
GUI
bind x = tween(100, 700, (time_now() - t0) / 2.0)     # slides over two seconds
draw_disc(x, 300, 20)
```

`tween_ease(a, b, t, curve)`

Interpolation with an easing curve.

- **Aliases:** zh `缓动补间`; ja `緩和補間`; ko `이징트윈`; th `แทรกนุ่ม`
- **Parameters:** `a`, `b` start and end; `t` progress 0 to 1; `curve` an Anima curve name (see above), default `"linear"`.
- **Returns:** number. Elastic and back curves can overshoot `b` or undershoot `a`.

```
GUI
bind y = tween_ease(500, 200, p, "bounce_out")      # drops and bounces
draw_disc(400, y, 24)
```

`breathe(t, rate, depth)`

A breathing multiplier centred on 1.0: `1 + depth * sin(2 pi * rate * t)`.

- **Aliases:** ko `호흡`; th `หายใจ`; zh/ja `呼吸`
- **Parameters:** `t` time in seconds; `rate` breaths per second (default 1); `depth` the amplitude (default 0.1).
- **Returns:** a number near 1. Multiply a size by it.
- **Notes:** It also drives pattern arguments: breathe the radius of a `vtex_flower`.

```
GUI
set_color(120, 220, 255)
draw_disc(120, 120, 40 * breathe(t, 0.4, 0.12))
```

CHAPTER 18

Audio, Music and the DAW

This chapter documents the sound builtins of the Ling runtime. They fall into three layers that are independent of each other:

1. **Audio engine** (`audio_*`, `ui_sound`, `morph_note`): a real-time positional synthesiser and sample player with a master effect chain. Backed by the `ling-audio` crate.
2. **Music toolkit** (`music_*`): decoding, analysis, playback of whole tracks, a General-MIDI-style polyphonic synthesiser driven by `.ling` patch files, MIDI files, rhythm-game judging and karaoke helpers. Backed by `ling-music`.
3. **DAW** (`daw_*`, `fs_*`, `dlg_*`, `st_*`, `px_*`): the scripting surface of the LingStationX digital audio workstation. Backed by `ling-daw`.

Conventions used in every entry:

- Arguments shown with a default are optional. A missing argument takes the default listed; a wrong type silently becomes the default (numbers) or an empty string.
- All times are in **seconds** unless an entry says *beats*.
- The runtime opens the audio device when the interpreter starts. If no output device exists, the `audio_*` builtins silently do nothing and a line beginning `audio init failed (no sound)` is printed to standard error. Analysis builtins (`music_load`, `music_bpm`, …) and all `daw_*` editing and rendering builtins work without a device.
- Aliases: most `audio_*` and `music_*` builtins can also be called by a translated name. The entries list the Chinese (zh), Japanese (ja), Korean (ko) and Thai (th) names. Most of them additionally have Persian, Arabic, Hebrew, Urdu, French, German and Russian names, which are not listed here; see `src/runtime/mod.rs`. The `daw_*`, `st_*` and `px_*` families have no aliases.
- Entries are tagged B (beginner essential), I (intermediate) or A (advanced or rarely needed).

Audio engine

The audio engine owns one output stream. Four kinds of sound share it:

- **Tones**: long-lived oscillators in numbered slots, positioned in 4D space (`audio_tone`).

- **One-shots**: UI blips (`audio_blip`, `ui_sound`), positional sound effects (`audio_sfx`) and morph notes (`morph_note`).
- **Samples**: decoded files played back at a position or flat (`audio_sample_*`).
- **Background music**: one looping WAV (`audio_bgm`).

The listener is the camera. `audio_listener` takes the cosine and sine of the camera yaw and pitch, the same four numbers you pass to `set_camera`, and `set_camera_pos` moves the listener position too. Pan is equal-power left/right from the sound's camera-space X; level falls off exponentially with distance. The final mix is soft-clipped with `tanh`.

Limits: tone slots grow on demand, with 32 simultaneous blips, 64 SFX voices, 32 morph voices and 64 sample voices. When a pool is full the oldest voice is dropped.

Master level and background music

Builtin	Level	Summary
`audio_volume(level)`	B	Set the master volume.
`audio_bgm(path, volume)`	B	Load a WAV file and loop it as background music.
`audio_bgm_volume(level)`	B	Change the background-music volume without reloading it.

`audio_volume(level)`

Set the master volume.

- **Aliases:** zh `音量` · ko `음량` · th `ระดับเสียง`
- **Parameters:**
- `level` (number, default 0.7) - linear gain applied to everything the audio engine mixes. 1.0 is unity.
- **Returns:** nothing.
- **Notes:** Does not affect the `music_*` engine, which has its own `music_volume`.

```
GUI
bind start = do {
  audio_volume(0.5)
}
```

`audio_bgm(path, volume)`

Load a WAV file and loop it as background music.

- **Aliases:** zh `背景乐` · ja `BGM` · ko `배경음악` · th `เพลงพื้นหลัง`
- **Parameters:**
- `path` (string) - path to a `.wav` file, used exactly as given (it is not resolved against the script folder).
- `volume` (number, default 0.5) - playback level.
- **Returns:** nothing. A missing or unreadable file prints `audio: bgm load failed` to standard error and the scene continues in silence.
- **Notes:**
- Only WAV is accepted here (read through the `hound` library and resampled to the device rate). Use `audio_sample_load` or `music_load` for MP3, OGG, FLAC and AAC.
- A second call replaces the current background track.

```
GUI
bind start = do {
  audio_bgm("theme.wav", 0.4)
}
```

`audio_bgm_volume(level)`

Change the background-music volume without reloading it.

- **Aliases:** zh `背景乐音量` · ja `BGM音量` · ko `배경음악음량` · th `ระดับเสียงพื้นหลัง`
- **Parameters:**
- `level` (number, default 0.5)
- **Returns:** nothing. Does nothing if no background track was loaded.

```
GUI
bind start = do {
  audio_bgm("theme.wav", 0.4)
  audio_bgm_volume(0.15)
}
```

Positional tones and the listener

Builtin	Level	Summary
`audio_tone(slot, x, y, z, w, freq, amp, lfo_rate, lfo_depth)`	I	Create or retune a continuous positional oscillator.
`audio_listener(cos_yaw, sin_yaw, cos_pitch, sin_pitch)`	I	Set the listener orientation used for panning and attenuation.

`audio_blip(freq, duration, wave, amp)`

Play a short interface blip.

- **Aliases:** zh `提示音` · ja `ビープ音` · ko `효과음` · th `เสียงบี๊บ`
- **Parameters:**
- `freq` (number, default 660) - frequency in Hz.
- `duration` (number, default 0.08) - length in seconds; minimum 0.01.
- `wave` (string, default "sine") - `"sine"`, `"square"` (or `"sq"`), `"saw"` / `"sawtooth"`, `"tri"` / `"triangle"`, `"noise"` (or `"wn"`, `"ns"`). Matching is case-insensitive and an unknown name falls back to sine.
- `amp` (number, default 0.25)
- **Returns:** nothing.
- **Notes:** Fast attack, exponential decay. Not positional.

```
GUI
bind start = do {
  audio_blip(880.0, 0.06, "square", 0.2)
}
```

`ui_sound(name)`

Play one of the built-in interface sounds.

- **Aliases:** zh `界面音` · ja `UI音` · ko `인터페이스음` · th `เสียงปุ่ม`
- **Parameters:**
- `name` (string, default "click") - `"click"`, `"hover"`, `"confirm"` (two rising square blips), `"error"` (two low saw blips), `"toggle"`, `"tick"`. Any other name plays `"click"`.
- **Returns:** nothing.

```
GUI
bind start = do {
  ui_sound("confirm")
}
```

`music_load(path)`

Decode an audio file into memory and return a track handle.

- **Aliases:** zh `载入音乐` · ja `音楽読込` · ko `음악로드` · th `โหลดเพลง`
- **Parameters:**
- `path` (string) - WAV, FLAC, OGG (Vorbis), MP3 or AAC/M4A.
- **Returns:** track handle (integer from 0), or `-1` on failure (the reason is printed to standard error).
- **Notes:** The whole file is decoded at once, producing an interleaved stereo copy for playback and a mono mix-down for analysis. Long files cost memory proportional to their length.

```
bind start = do {
  bind missing = music_load("no_such_file.wav")
  print(missing)
}
```

```
OUTPUT
-1
```

`music_duration(track)`

Length of a decoded track in seconds.

- **Aliases:** zh `音乐时长` · ja `音楽長さ` · ko `음악길이` · th `ความยาวเพลง`
- **Parameters:**
- `track` (integer) - handle from `music_load`.
- **Returns:** seconds as a number; `0` for an unknown handle.

`music_play(track)`

Start playing a track.

- **Aliases:** zh `播放音乐` · ja `音楽再生` · ko `음악재생` · th `เล่นเพลง`
- **Parameters:**
- `track` (integer) - handle from `music_load`.
- **Returns:** nothing.
- **Notes:**
- Always makes `track` the engine's current track and starts it from the beginning, so calling it again restarts the song; there is no separate resume builtin. To resume after `music_pause`, call `music_play` with an id that is not a loaded track (for example `-1`), which leaves the current track alone and just continues it; or note `music_pos()` first and call `music_seek` after `music_play`.
- Playback stops by itself at the end of the track (it does not loop). The music output stream is opened on first use; if that fails a message is printed and the call does nothing.
- On the web target `music_seek` is a no-op and `music_volume` always addresses slot 0.

```
GUI
bind start = do {
  bind song = music_load("song.ogg")
  music_volume(0.8)
  music_play(song)
}
```

`music_pause()`

Pause playback, keeping the position.

- **Aliases:** zh `暂停音乐` · ja `音楽一時停止` · ko `음악일시정지` · th `หยุดเพลงชั่วคราว`
- **Returns:** nothing.
- **Notes:** See `music_play` for how to resume.

```
GUI
bind start = do {
  music_pause()
}
```

`music_stop()`

Stop playback and rewind to the start.

- **Aliases:** zh `停止音乐` · ja `音楽停止` · ko `음악정지` · th `หยุดเพลง`
- **Returns:** nothing.

```
GUI
bind start = do {
  music_stop()
}
```

`music_volume(level)`

Master volume of the music engine (track and synthesiser together).

- **Aliases:** zh `音乐音量` · ja `音楽音量` · ko `음악음량` · th `ระดับเพลง`
- **Parameters:**
- `level` (number, default 0.8) - clamped to 0..2.
- **Returns:** nothing.
- **Notes:** The output is soft-clipped with `tanh`, so levels above 1 distort gently rather than clip hard.

```
GUI
bind start = do {
  music_volume(0.6)
}
```

Synthesis

The music engine contains a polyphonic synthesiser. An *instrument* is a patch file in the `.ling` patch format: plain `key=value` tokens separated by spaces or newlines, with `#` starting a comment. Operators are numbered from 1 and an operator may frequency-modulate a lower-numbered one.

```
OUTPUT
name = "Rhodes EP"
gm = 5
poly = 16
op1.wave=sine  op1.ratio=1.0  op1.level=1.0  op1.adsr=0.002,1.2,0.0,0.4
op2.wave=sine  op2.ratio=14.0 op2.level=0.5  op2.fm=op1  op2.adsr=0.001,0.2,0.0,0.2
filter.cutoff=0.8 filter.reso=0.2 filter.env=0.3 filter.adsr=0,0.5,0.2,0.3
lfo.rate=5.0 lfo.pitch=0.05
amp = 0.7
```

Recognised keys: `name`, `gm` (General MIDI program number), `poly` (voices, default 16), `amp`, `lfo.rate`, `lfo.pitch`, `lfo.cutoff`, `lfo.amp`, `filter.cutoff`, `filter.reso`, `filter.env`, `filter.adsr`, `filter.on` (any `filter.*` key switches the filter on), and per operator `opN.wave` (`sine`, `saw`, `square` / `pulse`, `tri`, `noise`), `opN.ratio`, `opN.fixed` (fixed frequency in Hz), `opN.level`, `opN.detune`, `opN.adsr` (attack, decay, sustain, release, comma separated) and `opN.fm` (the operator it modulates). Unknown keys are ignored. The repository ships ready-made patches in `instruments/`.

Builtin	Level	Summary
`music_patch(path)`	B	Load an instrument patch and return its handle.
`music_note(instrument, pitch, duration, velocity)`	B	Play a note of fixed length.
`music_note_on(instrument, pitch, velocity)`	I	Press a key and hold it.
`music_note_off(instrument, pitch)`	I	Release a key pressed with `music_note_on`.

`music_patch(path)`

Load an instrument patch and return its handle.

- **Aliases:** zh `乐器音色` · ja `音色読込` · ko `악기패치` · th `แพตช์เครื่องดนตรี`
- **Parameters:**
- `path` (string) - a `.ling` patch file.
- **Returns:** instrument handle (integer from 0), or `-1` if the file cannot be read or the music engine could not start.
- **Notes:** Patch handles are passed as the first argument of `music_note`, `music_note_on` and `music_note_off`. They are unrelated to track handles.

```
GUI
bind start = do {
  bind piano = music_patch("instruments/grand_piano.ling")
  music_note(piano, "C4", 0.5, 0.9)
}
```

`music_note(instrument, pitch, duration, velocity)`

Play a note of fixed length.

- **Aliases:** zh `弹音符` · ja `音符演奏` · ko `음표연주` · th `เล่นโน้ต`
- **Parameters:**
- `instrument` (integer, default 0) - handle from `music_patch`.
- `pitch` (number or string, default 60) - MIDI number or note name.
- `duration` (number, default 0.5) - seconds the key is held; the patch's release tail follows.
- `velocity` (number, default 0.9) - 0 to 1.
- **Returns:** nothing.

```
GUI
bind start = do {
  bind lead = music_patch("instruments/square_lead.ling")
  music_note(lead, "E5", 0.25, 0.8)
}
```

`music_hz(pitch)`

Convert a MIDI number or note name to a frequency.

- **Aliases:** zh `音符频率` · ja `音符周波数` · ko `음표주파수` · th `ความถี่โน้ต`
- **Parameters:**
- `pitch` (number or string, default 69) - MIDI number or note name. An unparsable string is read as MIDI 69.
- **Returns:** frequency in Hz. A4 (69) is 440.

```
bind start = do {
  print(music_hz("A4"))
  print(round(music_hz(60) * 100) / 100)
  print(round(music_hz("C5") * 100) / 100)
}
```

```
OUTPUT
440
261.63
523.25
```

`music_note_name(hz)`

Name of the nearest note to a frequency.

- **Aliases:** zh `音名` · ja `音名称` · ko `음이름` · th `ชื่อโน้ต`
- **Parameters:**
- `hz` (number, default 0)
- **Returns:** a string such as `"A4"` or `"C#5"` (sharp spelling); an empty string if `hz` is not positive.

```
bind start = do {
  print(music_note_name(440.0))
  print(music_note_name(262.0))
  print(music_note_name(0.0))
}
```

```
OUTPUT
A4
C4
```

`daw_bpm()`

Project tempo.

- **Returns:** BPM as a number.

`daw_set_bpm(bpm)`

Set the project tempo.

- **Parameters:**
- `bpm` (number) - clamped to 20..400.
- **Returns:** nothing.

```
bind start = do {
  daw_new()
  daw_set_bpm(1000)
  print(daw_bpm())
  daw_set_bpm(96)
  print(daw_bpm())
}
```

```
OUTPUT
400
96
```

`daw_track_count()`

Number of tracks.

- **Returns:** integer.

`daw_track_add(kind, path)`

Add a track.

- **Parameters:**
- `kind` (string) - `"synth"`, `"drums"`, `"audio"`, `"sf2"`, `"growl"`, `"voice"`, `"prism"`, `"sampler"`, `"pluck"`, `"chip"`, `"cinema"` or `"script"`.
- `path` (string, optional) - for instrument kinds, a patch file to load; for `"sf2"`, the SoundFont file (required).
- **Returns:** index of the new track, or `-1` if the track limit is reached or a SoundFont cannot be loaded.
- **Notes:**

- New tracks start with a gentle compressor in their effect chain. Colours cycle through a seven-colour palette.
- An unknown `kind` is an error, not a `-1` return: under `ling run --interp` the script stops with `unknown instrument type`; under the default backend the call yields unit and no track is created.

```
bind start = do {
  daw_new()
  print(daw_track_add("synth"))
  print(daw_track_add("drums"))
  print(daw_track_add("audio"))
  print(daw_track_count())
}
```

```
OUTPUT
0
1
2
3
```

`daw_track_name(track)`

Name of a track.

- **Parameters:**
- `track` (integer)
- **Returns:** string; empty for an invalid index.

`daw_track_set_name(track, name)`

Rename a track.

- **Parameters:**
- `track` (integer)
- `name` (string)
- **Returns:** nothing.

`daw_track_set(track, field, value)`

Change a track property.

- **Parameters:**
- `track` (integer)
- `field` (string) - `"vol"` (clamped 0..2), `"pan"` (-1..1), `"mute"` (above 0.5 means on), `"solo"`, or `"color"` (palette index 0..6).
- `value` (number)
- **Returns:** nothing.

```
bind start = do {
  daw_new()
  bind t = daw_track_add("synth")
  daw_track_set(t, "vol", 0.5)
  daw_track_set(t, "pan", -0.25)
  daw_track_set(t, "mute", 1)
  print(daw_track_get(t, "vol"))
  print(daw_track_get(t, "pan"))
  print(daw_track_get(t, "mute"))
}
```

```
OUTPUT
0.5
-0.25
1
```

`daw_clip_add_midi(track, start, length)`

Add an empty MIDI clip.

- **Parameters:**
- `track` (integer)
- `start` (number) - beats, at least 0.
- `length` (number) - beats, at least 0.25.
- **Returns:** clip index, or `-1` for an invalid track.

`daw_note_add(track, clip, start, length, key, velocity)`

Add a note to a MIDI clip.

- **Parameters:**
- `track` (integer)
- `clip` (integer)
- `start` (number) - beats from the clip start.
- `length` (number) - beats, at least 0.03.
- `key` (number) - MIDI note number, clamped to 0..127.
- `velocity` (number) - clamped to 0.01..1.
- **Returns:** note index within the clip, or `-1` if the clip is not a MIDI clip.
- **Notes:** For a drum track, the key selects the drum voice (36 kick, 38 snare, 42 closed hat, ...).

`daw_bounce(path, start, end, tail, bits, format, quality)`

Render the project to an audio file, synchronously.

- **Parameters:**
- `path` (string) - output file; missing folders are created.
- `start` (number, beats)
- `end` (number, beats) - if not greater than `start`, the end of the project is used.
- `tail` (number, seconds, default 0) - extra time after `end` for reverb and release tails.
- `bits` (integer, default 16) - sample depth for WAV and FLAC.
- `format` (string) - `"wav"`, `"flac"` or `"ogg"`; empty infers it from the file extension and falls back to WAV.
- `quality` (number 0..1, default 0.5) - Ogg Vorbis quality.
- **Returns:** `1` on success, `0` on failure (see `daw_status`).
- **Notes:** Rendering is faster than real time and needs no audio device. For long projects use the asynchronous `daw_render_start` so a window can keep updating.

CHAPTER 19

Physics, Game Helpers and the Body Toolkit

This chapter covers the builtins that simulate things: soft and rigid bodies, a liquid grid, hyperbolic space, the 3D game helpers (textured meshes, collision, particles), the dialog box system and the acoustic body toolkit. They are backed by the crates `ling-physics`, `ling-game` and `ling-body`; the crates themselves are described in the crates chapter.

Conventions:

- **Handles** are integers counting from 0 in creation order. Calls with an unknown handle do nothing and return `0`, an empty list or unit; they never raise an error.
- **Defaults.** An omitted or non-numeric argument takes the default listed in the entry.
- **Headless use.** The simulation builtins (`soft_*`, `rb_*`, `liquid_step`, `body_*`, `coll_*`, `pfx_emit`, `pfx_step`, `dialog_*` state) run without a window. The drawing builtins (`liquid_draw`, `tm_draw`, `pfx_draw`, `dialog_draw`, `body_draw`, ...) draw into the open window and need one.
- **Names.** The `soft_*`, `rb_*` and `liquid_*` builtins also have translated names; each entry lists the Chinese (zh), Japanese (ja), Korean (ko) and Thai (th) ones. The `tm_*`, `coll_*`, `pfx_*` and `body_*` families have none.

Which way is up

Two coordinate conventions meet in this chapter, and mixing them up is the most common cause of "my body falls through the floor".

- The renderer is **y-down**: positive y is toward the bottom of the screen. The soft-body builtins follow it. A floor is at the *largest* y a body may reach, and gravity is a positive number.
- The rigid-body world (`rb_*`) is **y-up** inside the engine: its default gravity is `(0, -9.81, 0)` and `rb_floor` catches a body that is *above* the floor level and *moving toward negative y*.

If you flip `rb_gravity` to a positive y to match the renderer, `rb_floor` can no longer detect the floor, because it only reacts to a falling body (negative y velocity). Keep rigid bodies y-up and subtract y when you draw them, or leave `rb_floor` out and handle the floor yourself with

`rb_pos` and `rb_impulse`.

Soft bodies

A soft body is a spring-mass sphere (86 nodes) that squashes and stretches on impact. All soft bodies are independent of each other and of the rigid-body world, and they do not collide with each other.

Builtin	Level	Summary
`soft_ball(x, y, z, radius)`	B	Create a deformable ball.
`soft_step(id, dt, gravity)`	B	Advance a soft body by one time step.
`soft_bounce(id, floor_y, restitution)`	B	Collide a soft body with a horizontal floor.
`soft_contain(id, min_x, min_y, min_z, max_x, max_y, max_z, restitution)`	I	Keep a soft body inside a box.
`soft_kick(id, dx, dy, dz, strength)`	I	Give a soft body a push.
`soft_spin(id, ax, ay, az, rate)`	I	Add angular velocity about an axis through the centroid.
`soft_deform(id)`	I	How squashed the body is.
`soft_angular_speed(id)`	A	Tumbling speed.
`soft_centroid(id)`	B	Centre of mass.
`soft_nodes(id)`	B	Positions of every node.

`soft_ball(x, y, z, radius)`

Create a deformable ball.

- **Aliases:** zh `软球` · ja `ソフトボール` · ko `소프트볼` · th `ลูกบอลนุ่ม`
- **Parameters:**
- `x, y, z` (numbers, default 0) - centre.
- `radius` (number, default 1.0)
- **Returns:** soft-body id (integer from 0).

```
bind start = do {
  bind ball = soft_ball(0.0, 0.0 - 3.0, 0.0, 1.0)
  print(ball)
  print(len(soft_nodes(ball)) / 3)
}
```

```
OUTPUT
0
86
```

`soft_step(id, dt, gravity)`

Advance a soft body by one time step.

- **Aliases:** zh `软体步进` · ja `ソフト更新` · ko `소프트스텝` · th `ก้าวนุ่ม`
- **Parameters:**
- `id` (integer)
- `dt` (number, default 0.016) - seconds.
- `gravity` (number, default 15.0) - acceleration along +y. Positive pulls toward the floor of the screen.
- **Returns:** nothing.
- **Notes:** Four internal sub-steps run per call. Call it once per frame.

```
bind start = do {
  bind ball = soft_ball(0.0, 0.0 - 3.0, 0.0, 1.0)
  bind frame = 0
  while frame < 300 {
    soft_step(ball, 0.016, 9.8)
    soft_bounce(ball, 2.0, 0.5)
    bind frame = frame + 1
  }
  bind c = soft_centroid(ball)
  print("rests at y = " + str(round(c[1] * 100) / 100))
}
```

```
OUTPUT
rests at y = 1.09
```

`soft_bounce(id, floor_y, restitution)`

Collide a soft body with a horizontal floor.

- **Aliases:** zh `软体落地` · ja `ソフト着地` · ko `소프트바운스` · th `เด้งนุ่ม`
- **Parameters:**
- `id` (integer)
- `floor_y` (number, default 0) - the floor plane. With the y-down convention the body lives at `y <= floor_y`.
- `restitution` (number, default 0.5) - bounciness, limited to 0..0.99.
- **Returns:** nothing.
- **Notes:** A slow contact settles the node on the plane instead of reflecting it, so a resting ball does not creep upward.

`soft_centroid(id)`

Centre of mass.

- **Aliases:** zh `软体质心` · ja `ソフト重心` · ko `소프트중심` · th `จุดศูนย์กลางนุ่ม`
- **Parameters:**
- `id` (integer)
- **Returns:** list `[x, y, z]`; `[0, 0, 0]` for an unknown id.

`soft_nodes(id)`

Positions of every node.

- **Aliases:** zh `软体节点` · ja `ソフト節点` · ko `소프트노드` · th `จุดนุ่ม`
- **Parameters:**
- `id` (integer)
- **Returns:** flat list `[x, y, z, x, y, z, ...]` of 86 nodes. Draw it to show the deformed surface.

Rigid bodies

One shared world of axis-aligned boxes (half-size 0.5 on every axis) with linear and angular dynamics. Bodies collide with each other with a simple O(n squared) test. See the note on y-up earlier in this chapter.

Builtin	Level	Summary
`rb_add(x, y, z, mass)`	B	Add a rigid body to the world.
`rb_gravity(gx, gy, gz)`	I	Set the world gravity.
`rb_step(dt)`	B	Advance the whole world.
`rb_floor(index, floor_y, restitution, friction)`	B	Bounce a body off a horizontal floor.
`rb_impulse(index, ix, iy, iz)`	I	Apply a linear impulse.
`rb_torque(index, tx, ty, tz)`	I	Apply a torque for the next step.
`rb_spin(index, wx, wy, wz)`	I	Add to the angular velocity.
`rb_pos(index)`	B	Position of a body.
`rb_rot(index)`	I	Orientation of a body.

`rb_add(x, y, z, mass)`

Add a rigid body to the world.

- **Aliases:** zh `刚体添加` · ja `剛体追加` · ko `강체추가` · th `เพิ่มวัตถุแข็ง`
- **Parameters:**
- `x, y, z` (numbers, default 0) - start position.
- `mass` (number, default 1.0)
- **Returns:** body index.
- **Notes:** New bodies have a restitution of 0.6 and a unit-cube extent.

`rb_step(dt)`

Advance the whole world.

- **Aliases:** zh `刚体步进` · ja `剛体更新` · ko `강체스텝` · th `ก้าวแข็ง`
- **Parameters:**
- `dt` (number, default 0.016) - seconds.
- **Returns:** nothing. Integrates every body (including rotation) and resolves body-body collisions.

`rb_floor(index, floor_y, restitution, friction)`

Bounce a body off a horizontal floor.

- **Aliases:** zh `刚体落地` · ja `剛体着地` · ko `강체바닥` · th `พื้นแข็ง`
- **Parameters:**
- `index` (integer)
- `floor_y` (number, default 0)
- `restitution` (number, default 0.6)
- `friction` (number, default 0.6) - converts sideways motion into roll.
- **Returns:** nothing.
- **Notes:** Call it every frame for every body that should be able to hit the floor. It acts only when the body's lower face is below `floor_y` and it is moving toward negative y; see the y-up note.

```
bind start = do {
  bind box = rb_add(0.0, 5.0, 0.0, 1.0)
  rb_impulse(box, 2.0, 0.0, 0.0)
  bind frame = 0
  while frame < 240 {
    rb_step(0.016)
    rb_floor(box, 0.0, 0.5, 0.6)
    bind frame = frame + 1
  }
  bind p = rb_pos(box)
  print("x = " + str(round(p[0] * 100) / 100))
  print("y = " + str(round(p[1] * 100) / 100))
}
```

```
OUTPUT
x = 4.5
y = 0.5
```

`rb_pos(index)`

Position of a body.

- **Aliases:** zh `刚体位置` · ja `剛体位置` · ko `강체위치` · th `ตำแหน่งแข็ง`

- **Parameters:**
- `index` (integer)
- **Returns:** list `[x, y, z]`.

`liquid_new(width, height)`

Create a fluid grid.

- **Aliases:** zh `新建液体` · ja `液体新規` · ko `액체생성` · th `สร้างของเหลว`
- **Parameters:**
- `width` (integer, default 64) - cells; at least 2.
- `height` (integer, default 64) - cells; at least 2.
- **Returns:** grid id.

`liquid_splat(id, x, y, kind, amount, radius)`

Add fluid in a disc.

- **Aliases:** zh `液体注入` · ja `液体追加` · ko `액체분사` · th `หยดของเหลว`
- **Parameters:**
- `id` (integer)
- `x, y` (numbers, default 0) - centre in cells.
- `kind` (integer, default 0) - `0` water, `1` oil.
- `amount` (number, default 1.0)
- `radius` (number, default 4.0) - cells.
- **Returns:** nothing.

`liquid_step(id, dt)`

Advance one grid.

- **Aliases:** zh `液体步进` · ja `液体更新` · ko `액체스텝` · th `ก้าวของเหลว`
- **Parameters:**
- `id` (integer)
- `dt` (number, default 0.016) - seconds.
- **Returns:** nothing.

`dialog_show(text, chars_per_second)`

Start a dialog.

- **Aliases:** zh `对话显示` · ja `会話表示` · ko `대화표시` · th `แสดงบทสนทนา`
- **Parameters:**
- `text` (string) - text with the markup above.
- `chars_per_second` (number, default 32) - typing speed, at least 1.
- **Returns:** nothing.
- **Notes:** Replaces any dialog already open.

`dialog_step(dt)`

Advance the typewriter and the auto-advance timer.

- **Aliases:** zh `对话步进` · ja `会話更新` · ko `대화스텝` · th `ก้าวบทสนทนา`
- **Parameters:**
- `dt` (number, default 0.016) - seconds.
- **Returns:** nothing. Call it every frame.

`dialog_advance()`

Respond to the player pressing the confirm key.

- **Aliases:** zh `对话推进` · ja `会話送り` · ko `대화진행` · th `เลื่อนบทสนทนา`
- **Returns:** nothing. If the page is still typing it is revealed at once; otherwise the dialog moves to the next page, or closes after the last page.

`dialog_active()`

Whether a dialog is open.

- **Aliases:** zh `对话激活` · ja `会話中` · ko `대화중` · th `บทสนทนาทำงาน`
- **Returns:** `true` or `false` (a real boolean).

`dialog_typing()`

Whether the current page is still being typed.

- **Aliases:** zh `对话打字` · ja `会話タイプ中` · ko `대화타이핑` · th `กำลังพิมพ์บทสนทนา`

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

`dialog_draw(x, y, width, height, font)`

Draw the dialog box and its revealed text.

- **Aliases:** zh `对话绘制` · ja `会話描画` · ko `대화그리기` · th `วาดบทสนทนา`
- **Parameters:**
- `x, y` (numbers, default 40 and 0) - top-left of the box.
- `width, height` (numbers, default 720 and 150)
- `font` (integer, default -1) - a font handle from `font_load`; a negative value uses the runtime's fallback text drawing.
- **Returns:** nothing.

```
GUI
bind start = do {
  open_window(800, 300)
  dialog_show("{n}Ling{/} arrives at {p}the harbour{/}.||She finds a {i}silver
key{/}.", 30)
  dialog_step(0.5)
  dialog_draw(40, 120, 720, 150)
  present()
}
```

Stepping a dialog without a window

```
bind start = do {
  dialog_show("Hello {n}Ling{/}.||Second page.", 100)
  print("active: " + str(dialog_active()))
  print("typing: " + str(dialog_typing()))
  dialog_step(0.5)
  print("typing after 0.5 s: " + str(dialog_typing()))
  dialog_advance()
  print("page 2 typing: " + str(dialog_typing()))
  dialog_advance()
  dialog_advance()
  print("active at the end: " + str(dialog_active()))
}
```

```
OUTPUT
active: true
typing: true
typing after 0.5 s: false
page 2 typing: true
active at the end: false
```

Small utilities in the game helpers

These builtins live in the same module as the `tm_` family but have nothing to do with meshes. They are handy in games with save codes, networking or shared random worlds.

Builtin	Level	Summary
`str_to_hex(text)`	I	Encode text as lowercase hexadecimal.
`hex_to_str(hex)`	I	Decode hexadecimal to text.
`random_hex(bytes)`	I	Random bytes from the operating system, as hex.
`rand_seed(n)`	I	Reseed `rand()`.
`net_local_ip()`	A	This machine's LAN address.
`clipboard_set(text)`	A	Put text on the system clipboard.
`clipboard_get()`	A	Read the system clipboard.

CHAPTER 20

Input, Microphone and Speech-to-Text

This chapter covers three kinds of input beyond the keyboard and mouse: game controllers, the microphone, and recognising speech in recorded audio. The keyboard and mouse builtins (`key_down`, `key_pressed`, `mouse_x`, ...) belong to the window chapter.

- **Game controllers** come in two generations of builtin. The `pad_*` family is the modern one: up to four players, edge-detected buttons, analog triggers, and the same code path on desktop and in the browser. The older `gamepad_*` family reads the first active controller by name and is the one that can make a controller vibrate. Both are backed by the `ling-input` crate ("Sensorium") and the `gilrs` library; in a browser build the `pad_*` family uses the W3C Gamepad API.
- **The microphone** (`mic_*`, plus `music_mic_pitch` in the music chapter) is backed by `ling-mic`.
- **Speech-to-text** (`stt_*`) runs the Whisper model locally. Nothing is sent over the network.

Every `pad_*`, `gamepad_*` and `mic_*` builtin is safe to call when no device exists: buttons read as not pressed, axes as 0, and the microphone as silent. Treat the return value of `mic_open` and `pad_count` as the way to find out whether hardware is present.

Game controllers: the pad_ family

Call `pad_poll` once per frame, before reading anything. It drains the controller events, advances the edge detection and returns the number of connected pads. Player slots count from 0; there are four.

Button names accepted by `pad_button` and `pad_pressed` (case-insensitive; several names are aliases of the same button):

Button	Names
bottom face	`a`, `south`, `cross`
right face	`b`, `east`, `circle`
left face	`x`, `west`, `square`

Button	Names
top face	`y`, `north`, `triangle`
left shoulder	`lb`, `l1`, `left_shoulder`
right shoulder	`rb`, `r1`, `right_shoulder`
left trigger (digital)	`lt`, `l2`, `left_trigger`
right trigger (digital)	`rt`, `r2`, `right_trigger`
start	`start`, `menu`, `options`
select	`select`, `back`, `share`, `view`
guide	`guide`, `home`
stick clicks	`l3`, `left_stick`; `r3`, `right_stick`
d-pad	`up` / `dpad_up`, `down` / `dpad_down`, `left` / `dpad_left`, `right` / `dpad_right`

An unknown name reads as not pressed. On a desktop build the face buttons are positional (the bottom button is `a` on every vendor's pad), so you never need to branch on the controller brand.

Stick direction. On desktop the vertical axes follow the screen convention: pushing a stick *up* gives a *negative* `pad_ly` or `pad_ry`. In a browser build the values are those of the Gamepad API, which use the same screen-down convention.

Builtin	Level	Summary
`pad_poll()`	B	Advance controller input by one frame.
`pad_count()`	B	Number of connected pads.
`pad_connected(player)`	B	Whether a pad is assigned to a player slot.
`pad_button(player, name)`	B	Whether a button is held down.
`pad_pressed(player, name)`	B	Whether a button went down this frame.
`pad_lx(player)`	B	Left stick, horizontal.
`pad_ly(player)`	B	Left stick, vertical.
`pad_rx(player)`	B	Right stick, horizontal.
`pad_ry(player)`	B	Right stick, vertical.
`pad_lt(player)`	I	Left analog trigger.
`pad_rt(player)`	I	Right analog trigger.
`pad_rumble(player, low, high)`	I	Request controller vibration through the Sensorium backend.

`pad_poll()`

Advance controller input by one frame.

- **Aliases:** zh `手柄轮询` · ja `パッド更新` · ko `패드폴링` · th `อัปเดตแพด`
- **Returns:** the number of connected pads (integer).
- **Notes:**
- Call it once per frame before any `pad_*` read. The first call opens the controller backend; if that fails it returns 0 and later reads give neutral values.
- Edge detection uses a fixed 60 Hz step, independent of your real frame rate.

```
GUI
bind start = do {
  open_window(640, 360)
  while window_is_open() {
    bind pads = pad_poll()
    if pad_pressed(0, "a") {
      print("A pressed on a pad; pads connected: " + str(pads))
    }
    present()
  }
}
```

`pad_count()`

Number of connected pads.

- **Aliases:** zh `手柄数` · ja `パッド数` · ko `패드수` · th `จำนวนแพด`
- **Returns:** integer, as of the last `pad_poll`.

`pad_connected(player)`

Whether a pad is assigned to a player slot.

- **Aliases:** zh `手柄连接` · ja `パッド接続` · ko `패드연결` · th `แพดเชื่อม`
- **Parameters:**
- `player` (integer, default 0) - slot 0 to 3.
- **Returns:** `true` or `false`.

`pad_button(player, name)`

Whether a button is held down.

- **Aliases:** zh `手柄按键` · ja `パッドボタン` · ko `패드버튼` · th `ปุ่มแพด`
- **Parameters:**
- `player` (integer, default 0)
- `name` (string) - see the table of names above.
- **Returns:** `true` while the button is down.

`pad_pressed(player, name)`

Whether a button went down this frame.

- **Aliases:** zh `手柄按下` · ja `パッド押下` · ko `패드눌림` · th `แพดกด`
- **Parameters:**
- `player` (integer, default 0)
- `name` (string)
- **Returns:** `true` only on the frame of the press.
- **Notes:** In a browser build only the current snapshot is available, so `pad_pressed` behaves like `pad_button` there.

`pad_lx(player)`

Left stick, horizontal.

- **Aliases:** zh `手柄左X` · ja `パッド左X` · ko `패드왼X` · th `แพดซ้ายX`
- **Parameters:**
- `player` (integer, default 0)
- **Returns:** -1 (left) to 1 (right); `0` without a pad.
- **Notes:** The value is the stick's raw position. No extra deadzone is applied by the builtin, so a worn stick can read slightly off zero at rest; ignore values with a small absolute value in your game.

`pad_ly(player)`

Left stick, vertical.

- **Aliases:** zh `手柄左Y` · ja `パッド左Y` · ko `패드왼Y` · th `แพดซ้ายY`

- **Parameters:**
- `player` (integer, default 0)
- **Returns:** -1 (up) to 1 (down); `0` without a pad.

`pad_rx(player)`

Right stick, horizontal.

- **Aliases:** zh `手柄右X` · ja `パッド右X` · ko `패드오X` · th `แพดขวาX`
- **Parameters:**
- `player` (integer, default 0)
- **Returns:** -1 to 1.

`pad_ry(player)`

Right stick, vertical.

- **Aliases:** zh `手柄右Y` · ja `パッド右Y` · ko `패드오Y` · th `แพดขวาY`
- **Parameters:**
- `player` (integer, default 0)
- **Returns:** -1 (up) to 1 (down).

`mic_open()`

Open the default microphone and start capturing.

- **Aliases:** zh `开麦克风` · ja `マイク開く` · ko `마이크열기` · th `เปิดไมค์`
- **Returns:** `1` on success, `0` if there is no device or access was refused. On the web target it returns unit.
- **Notes:** Call it once. Reading levels afterwards costs almost nothing.

```
GUI
bind start = do {
  if mic_open() == 1 {
    print("listening")
  } else {
    print("no microphone")
  }
}
```

`mic_rms()`

Loudness of the latest buffer.

- **Aliases:** zh `麦克风音量` · ja `マイクRMS` · ko `마이크RMS` · th `เสียงRMS`
- **Returns:** root-mean-square level from 0 to 1; `0` if the microphone is closed.

```
GUI
bind start = do {
  mic_open()
  bind level = mic_rms()
  print(level)
}
```

PART

Showcase

Real applications written entirely in Ling.

CHAPTER 21

Case Studies: Applications Built in Ling

These are real programs, shipped by the author of the language, written in Ling from the first line to the last. They are included to show what the language does at scale and what its authors learned doing it. Line counts are lines of Ling in the application's own `ling/` folder, measured with `wc -l`, and exclude assets, generated data and the Rust crates the programs call. Screenshots were taken from the running programs; the wallet's were taken with its simulated (`--mock`) backend, so every balance, address and peer on them is made up.

Application	Kind	Ling code	Native help
LingStationX	Digital audio workstation	25,900 lines, 60 modules	`ling-daw` audio engine (Rust)
Ling Wallet	Post-quantum wallet, node and miner	8,800 lines, 33 modules	`ling-crypto`, `ling-net` (Rust)
LingDev	Game scene, timeline and cutscene editor	28,400 lines, 52 modules	Software 3D renderer is Ling
LingModeler	3D modeler, vector studio, texture lab	16,000 lines, 32 modules	None beyond the runtime
Horizon	Web browser with its own HTML, CSS and script engines	16,300 lines, 12 modules	Image and video decoders
Soul Symphony, Ling Edition	3D game in six languages	26,000 lines of game code	Runtime graphics and audio

LingStationX

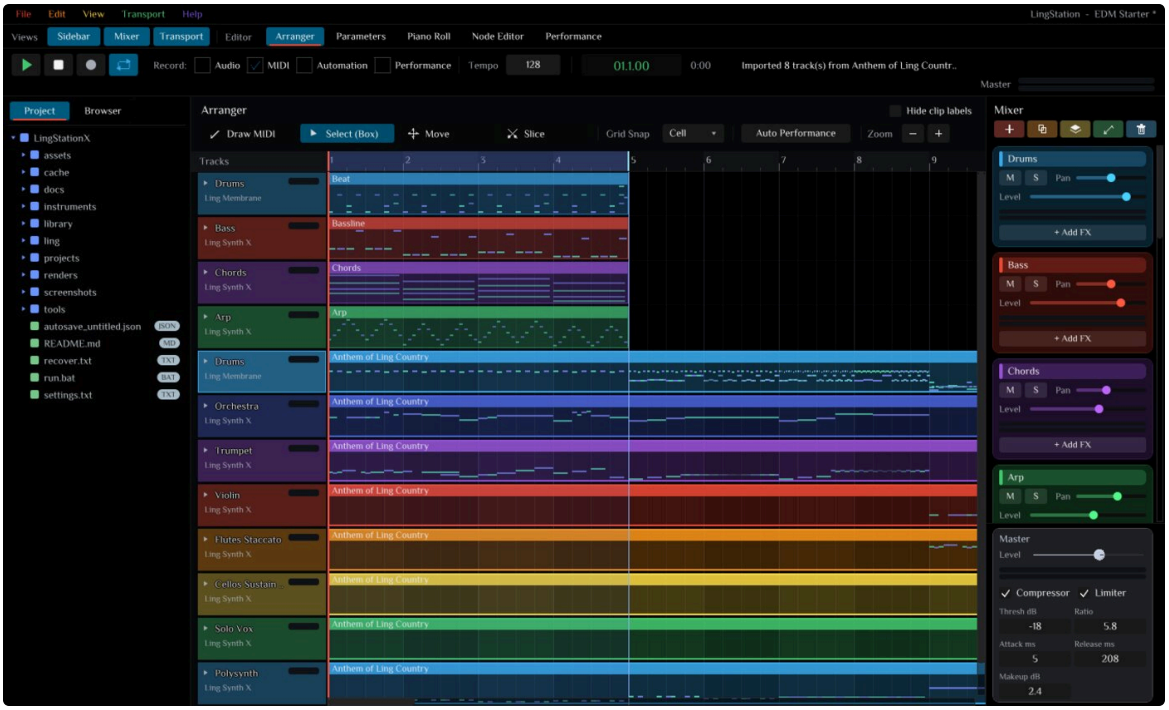


Figure 90.1. The arranger. Each track tile, clip, ruler tick and mixer strip is drawn by Ling code every frame; the clips of an imported eight-track song sit below four template tracks.

What it is. A complete digital audio workstation: arranger with automation lanes, piano roll with per-note expression, mixer with effect racks, a node editor for signal routing, a clip launcher for live performance, a file browser, a theory assistant, MIDI and audio import and export, and several instruments (a subtractive, additive and FM synth, a physical-modelling drum machine, a bass synth, a 16-voice synth called Prism, and a formant singer). It ships for Windows, Linux and macOS, and as an app inside the LingOS operating system on x86-64 and Raspberry Pi 5.

Why Ling fits. A DAW has two halves with opposite needs: a hard-real-time audio engine and a huge, fast-changing user interface. Ling splits them cleanly. The engine is a Rust crate that Ling controls through `daw_*` builtins, so the audio thread never waits on a script. Everything the user touches is Ling: a screen is a function, a widget is a function, and a new panel is added in an afternoon without touching the engine. Instruments are text patches, and users can write their own effects and instruments in a few lines of the LingDSP expression language in `scripts/`, reloaded while the program runs.

Size and structure. 60 modules in `ling/` (25,894 lines): `core` for input and widgets, `ds` for the design system, `actions` for the `do_action(id)` command dispatcher, then one module per screen (`arranger`, `roll`, `mixer`, `nodes`, `perf`). More than 900 patch files and several project templates live in `instruments/` and `assets/`. Chapter 10 walks through the architecture.



Figure 90.2. The Parameters tab showing Ling Prism, a 16-voice synthesiser, with its oscillators, filter, envelopes and modulation matrix. The panel is a Ling function reading the instrument's parameter list from the engine.

Lessons learned.

1. **Put the engine behind a narrow builtin API and keep policy in the script.** The Rust side offers verbs (`daw_note_add`, `daw_undo_push`); the Ling side decides when to call them. It made the engine testable headless, and the same `daw_*` calls render previews on a web server.
2. **Immediate-mode UI works if you cache what is expensive.** Redrawing everything every frame is simple and correct, but a per-note drawing loop in script cost about 2 ms per clip per frame. The fix was one native call (`px_notes`) and a cache keyed on the project revision.
3. **Shared state belongs in one store.** A function cannot see its caller's variables, so the program keeps all state in `st_get` / `st_set`, prefixed by module. It reads like global variables but is searchable, serialisable, and drivable from tests.

4. **Make the program testable from the command line early.** `--shot`, `--do` and `--drive` replay scripted input and write a screenshot. `tools/smoke.py` opens every dialog, tab and menu this way, and the project's `screenshots/` folder holds more than a thousand captures.
5. **Know the language's sharp edges and write them down.** The team's `DEV_NOTES.md` lists them: every keyword of every supported human language is reserved (so `si`, `ok` and `fit` cannot be variable names), `&&` does not short-circuit so bounds must be checked first, and boolean-returning builtins differ from number-returning ones.

Ling Wallet

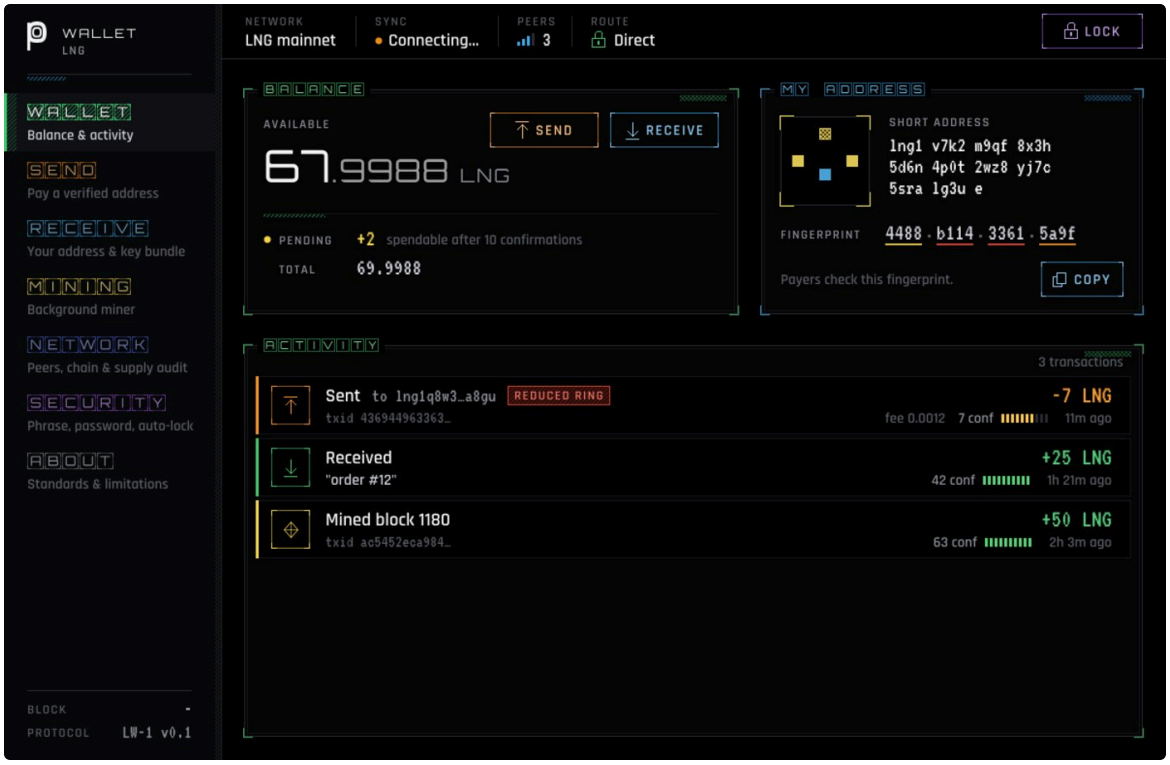


Figure 90.3. The wallet screen on the simulated backend: balance, the fingerprint of the user's address and three activity rows, including a send flagged as using a reduced ring.

What it is. A desktop wallet, full node and background miner for LNG, an experimental privacy coin, in one program. Payments use fresh one-time addresses, ring signatures over same-value decoys, a hybrid post-quantum key exchange, and Dandelion++ propagation. The

wallet file is encrypted with Argon2id and XChaCha20-Poly1305. A 4-D "HyperTorus" visualisation animates the miner. It is explicitly labelled unaudited and experimental, in its README, its About screen and a written threat model.

Why Ling fits. The project's thesis is that a security-critical program is easier to trust when it is all in one readable language. Ling contributes three things that matter here: **isolates with capabilities** (the miner can only hash; only the wallet thread can create a key), **opaque key handles** (secrets never become Ling values), and a **strict byte layer** (hex strings plus bounds-checked readers). The UI and the engine meet at a small contract of non-blocking `be_*` calls, implemented once for the real node and once as a mock.

Size and structure. 33 modules (8,824 lines): `core/` for consensus and wallet logic without I/O, `isolates/` for the node, wallet and miner threads, `ui/` for sixteen screen and widget modules, plus `tests/` for byte-compatibility against a reference implementation. Chapter 11 walks through the cryptographic recipes.

Lessons learned.

1. **Make the architecture enforce the security policy.** Capabilities are checked by the runtime, not by convention. A reviewer can answer "can the miner read the wallet file?" from one `thread_spawn` line.
2. **Ship a mock backend.** `--mock` made every screen designable and screenshot-testable without a chain, a network or a real key. It also stopped screenshots from ever containing real data.
3. **Test against a reference.** Expected values in `t_crypto.ling` came from the original Rust reference implementation, so a passing test means byte compatibility, not merely self-consistency.
4. **Put the checks where a human can fail.** The send flow shows the recipient's fingerprint and refuses to continue until the user confirms it matches, because a checksum protects against typos but not against a swapped address.
5. **Be honest in the product.** Standard, novel and experimental primitives are labelled as such on the About screen, and `wipe` is documented as best effort because it is. Trust comes from stating the limits.

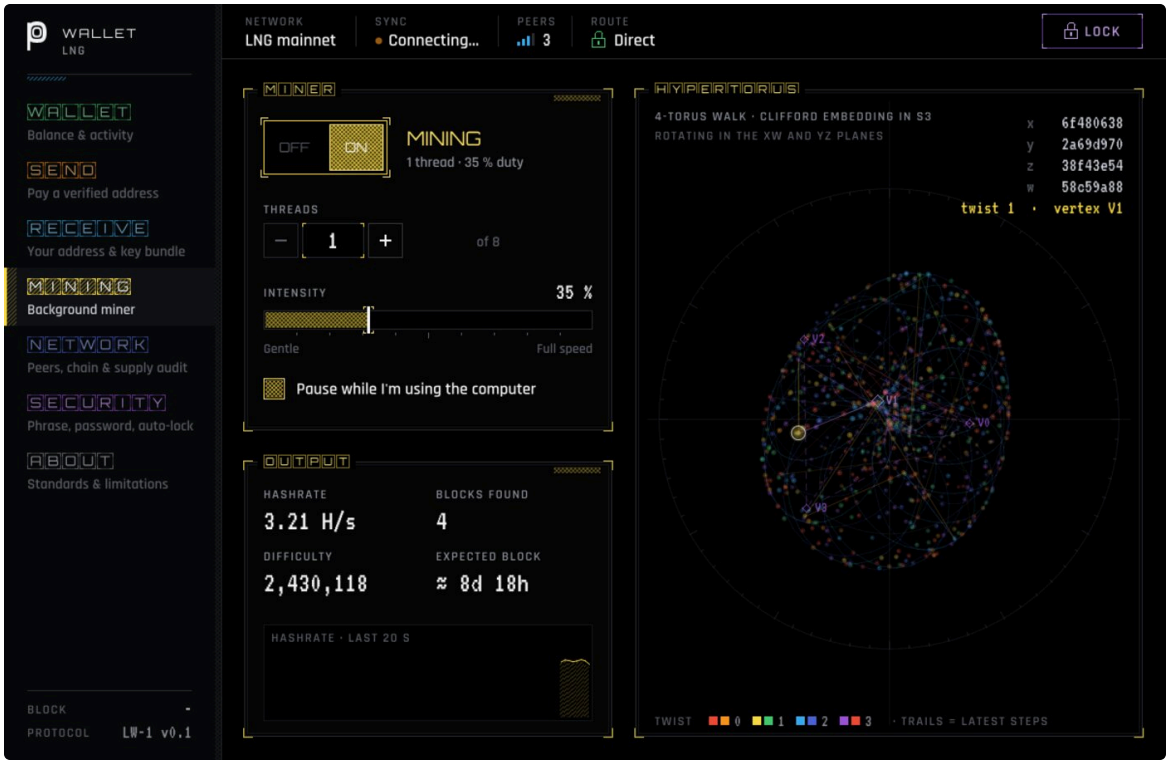


Figure 90.4. The Mining screen on simulated data: thread count, duty-cycle intensity, a pause-while-I-am-using-the-computer option and the 4-D torus walk drawn from `lw_pow_trace` samples.

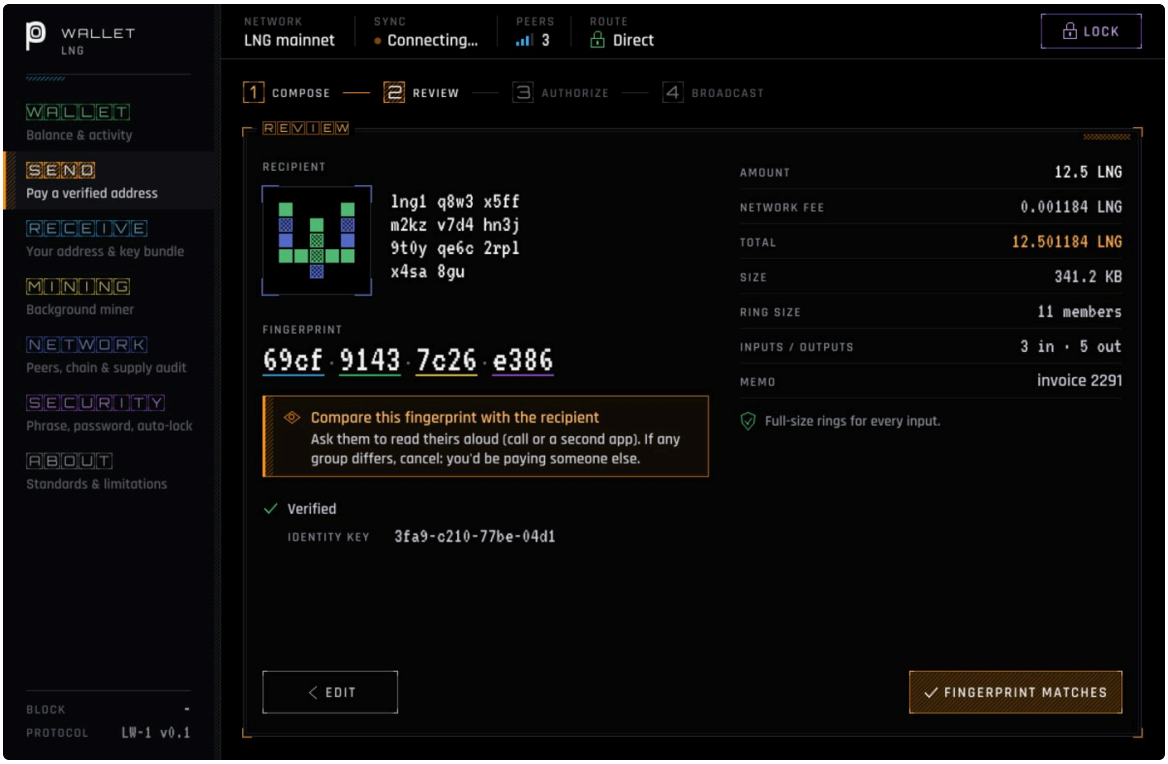


Figure 90.5. The review step of a payment (simulated data): the recipient's identicon and fingerprint, fee, ring size and a prompt to compare the fingerprint out of band.

LingDev

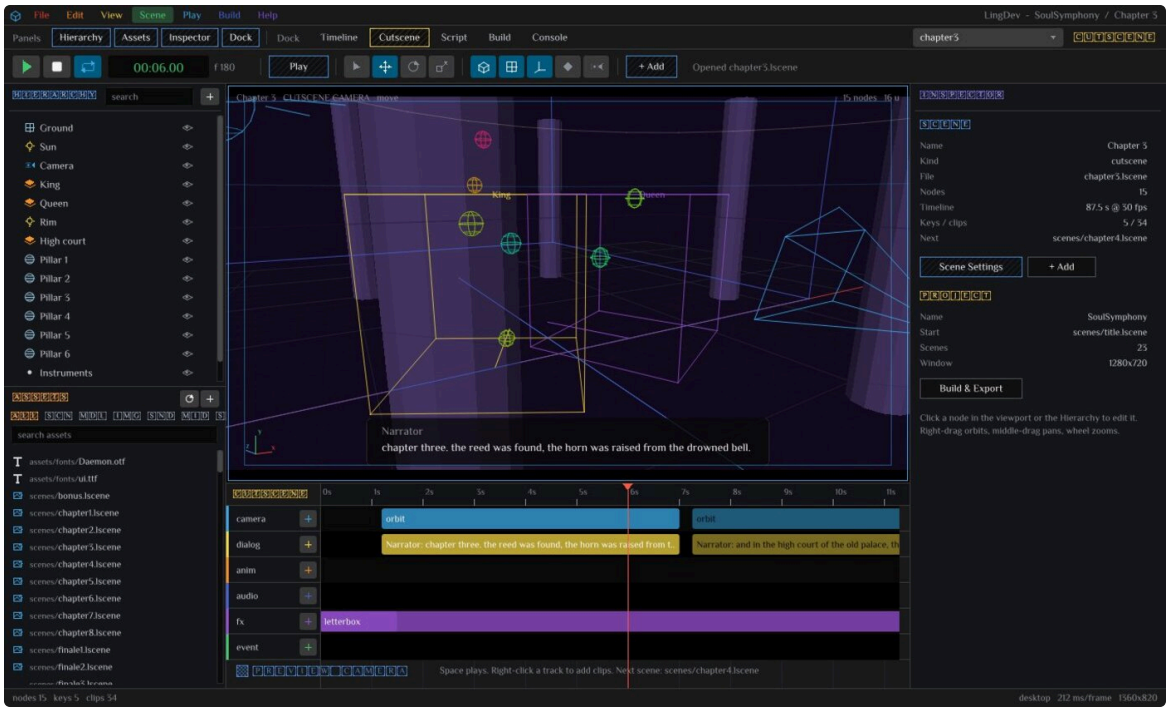


Figure 90.6. LingDev editing a cutscene of the Soul Symphony Ling edition: hierarchy and assets on the left, the 3D viewport with the camera path, the inspector on the right and the cutscene tracks in the dock.

What it is. A scene, timeline and cutscene editor for games, itself written in Ling. It edits 3D levels, 2D menus and HUDs, and cinematic cutscenes with six track types (camera, dialog, animation, audio, effects, events). It has a viewport with gizmos, a Flash-style keyframe timeline with 21 easing curves, a script editor, a one-key build panel for Windows, Linux, macOS, the web, a Raspberry Pi kernel image and LingOS, and importers that turned existing games into projects. Six complete games are built with it: a first-person shooter, a breakout game, a rhythm game, an arena shooter, a storybook platformer and the 2019 Soul Symphony.

Why Ling fits. An editor and the thing it edits should speak the same language. LingDev's runtime (19 modules, 18,500 lines) is Ling, copied into every project, and the editor `use`s the same modules to play a game inside its own window. There is no export step that can disagree with the preview.

Size and structure. 28,355 lines: 9,856 in the editor (`app`, `project`, `timeline`, `cutscene`, `inspector`, `hier`, `view3d`, `view2d`, `buildpanel`, `importers`) and 18,499 in the shared runtime `rt/`. Projects are a `lingdev.toml`, line-based `.lscene` files and plain `.ling` scripts. Chapter

12 has recipes from it.

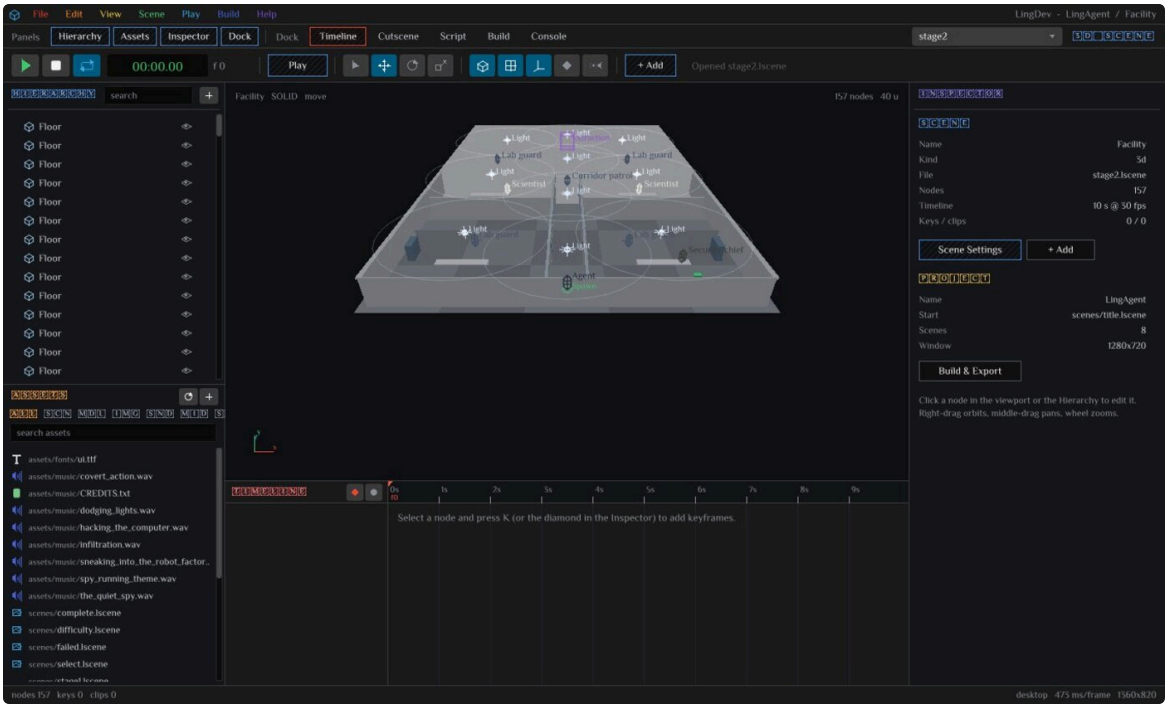


Figure 90.7. The same editor editing a first-person stage: guards, lights and the extraction point in the 3D viewport, drawn by the software renderer in the shared runtime.

Lessons learned.

1. **One runtime for the editor and the game.** Parsing, serialising and rendering a scene happen in shared code, so the preview cannot drift from the product.
2. **Design files for diffs.** Line-per-record text with ignorable unknown fields lets artists use version control and lets old projects open in new versions.
3. **Snapshots beat inverse operations for undo.** Serialising the scene to a string takes a few lines and makes every edit undoable, including edits that did not exist yet.
4. **A software renderer is a feature.** Drawing 3D with Ling code (`rt_sw3d`) means the same viewport code runs on the desktop and in the LingOS builds, without depending on a GPU API.
5. **Easing and interpolation are worth writing in the language.** `rt_ease.ling` runs identically on every platform, and a designer can read it.

LingModeler

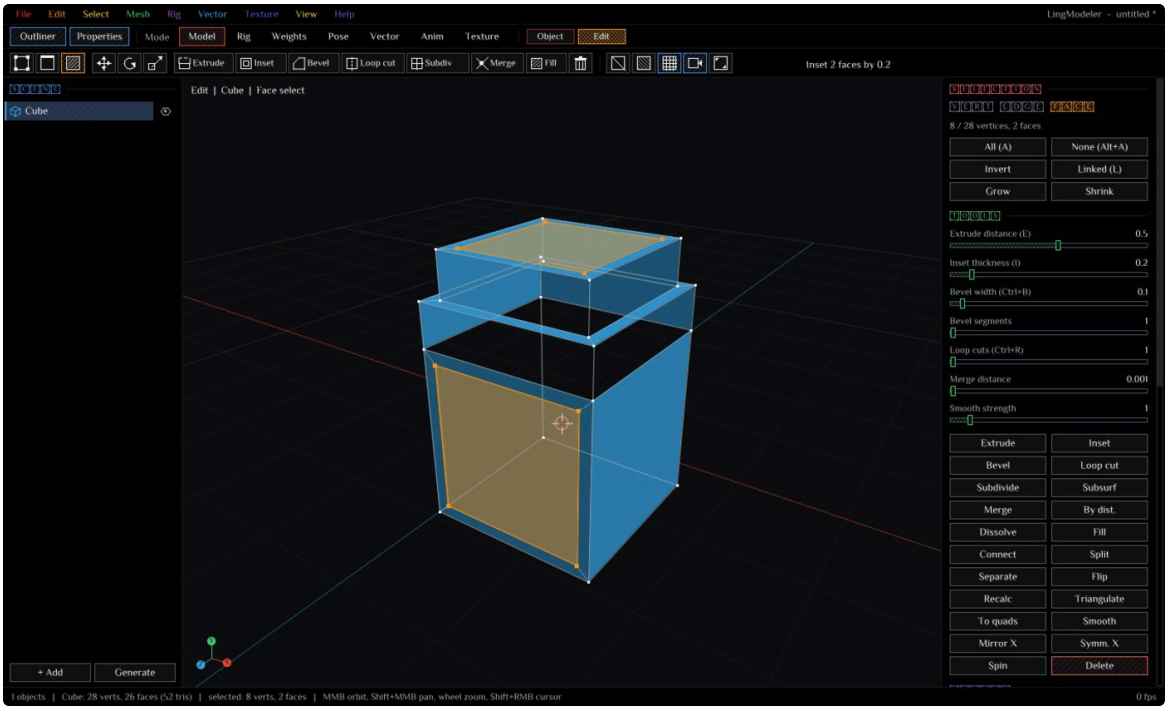


Figure 90.8. Edit mode on a box mesh, with the extrude and inset panels on the right.

What it is. A lightweight 3D modeler, 2D vector illustrator, animation studio, procedural texture lab and rigging tool in one window. Modes cover object and edit (box modeling with extrude, bevel, loop cut, subdivision, UV projections), rig (a 19-bone character armature), weight painting with auto weights, pose and keyframed actions, vector paths and shapes, 2D animation with onion skin, and 16 procedural texture generators. It reads and writes OBJ, glTF, SVG and PNG, and its native `.lmesh` format is valid Ling source.

Why Ling fits. Ling's lists and string-keyed store make a mesh a handful of flat arrays; copying them for undo is cheap because values share storage until edited. Because the project file is itself Ling source, a game can `use "hero.lmesh"` and call `draw_Hero(...)` without writing a loader. The same program runs on Windows, macOS, Linux, and inside LingOS on x86-64 and Raspberry Pi 5.

Size and structure. 32 modules (16,015 lines). The largest are `meshops` (1,873, the modeling operators), `vec2d` (1,537), `view3d` (1,534) and `rig` (1,205).

Lessons learned.

1. **Choose data layouts that fit the language.** Flat parallel lists (`vx`, `vy`, `vz`, `fi`, `fs`, `fc`) are simple, copy cheaply and handle tris, quads and n-gons together.
2. **A file format that is also source code is free documentation.** Anyone can open a `.lmesh` and read it.
3. **Keep a procedural texture engine tiny.** Sixteen generators are a hash, a noise, an fBm and some arithmetic. Pass the tile period into the noise and every texture can be seamless.
4. **State limits plainly.** The README lists what the importers skip (SVG text, gradients, clip paths; glTF animations). Users forgive limits they were told about.

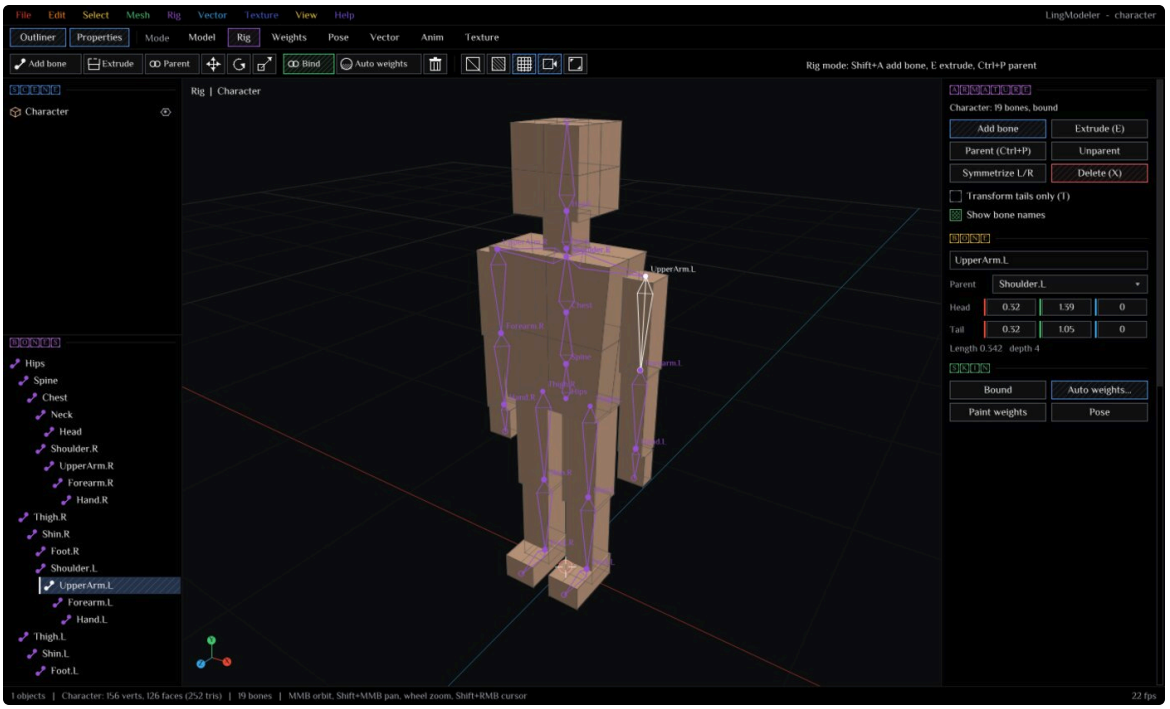


Figure 90.9. Rig mode: a generated character with its 19-bone armature, bone hierarchy on the left and bone properties on the right.

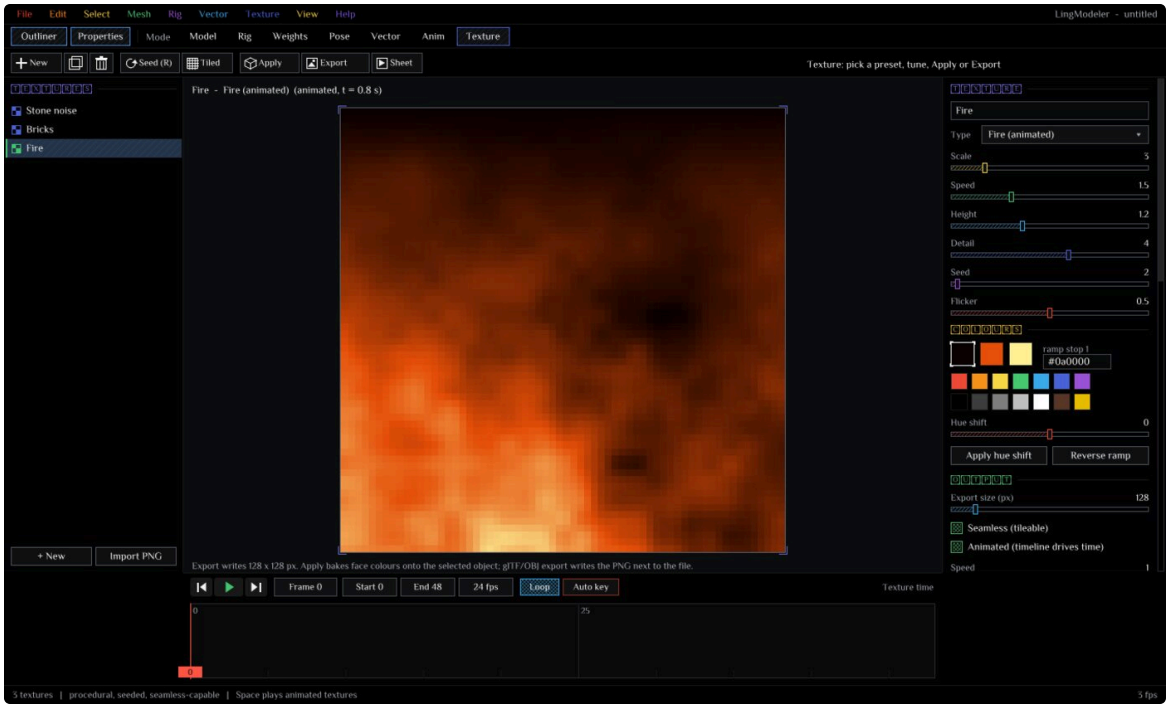


Figure 90.10. The texture lab showing the animated fire generator with its colour ramp and seamless-tiling options.

Horizon

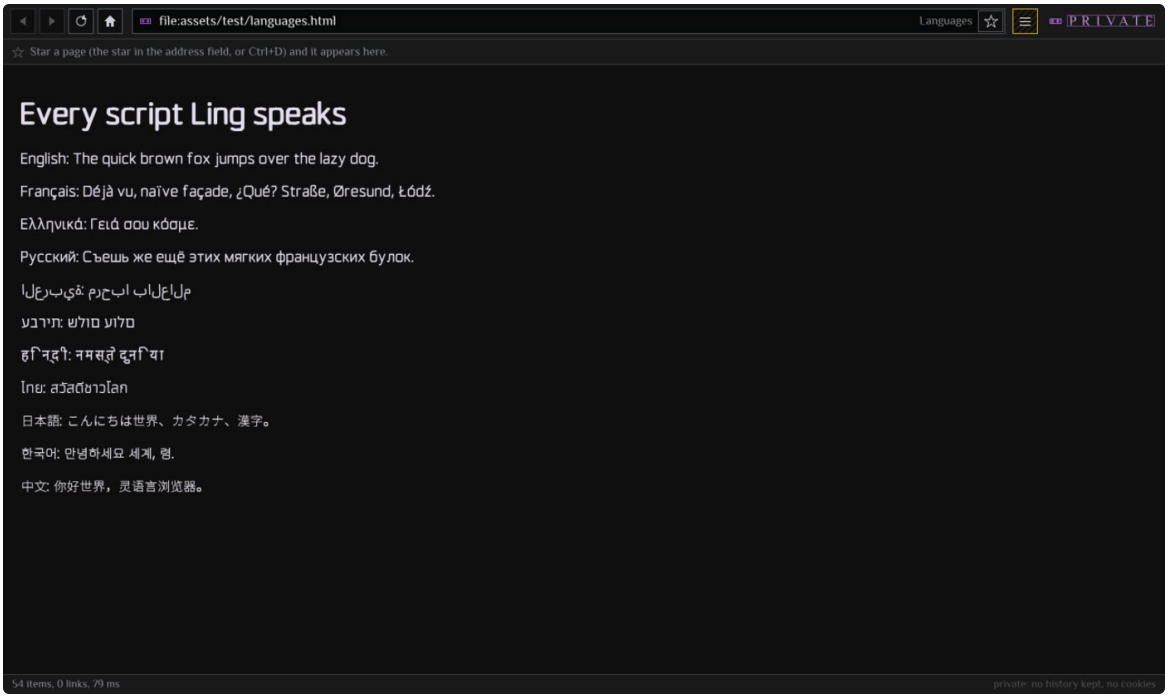


Figure 90.11. Horizon rendering its test page `languages.html`: Latin, Greek, Cyrillic, Arabic, Hebrew, Devanagari, Thai, Japanese, Korean and Chinese, set in its own stroke font.

What it is. A private web browser, a Ling program in the look of the other apps: toolbar, address field, bookmarks bar, status line. It parses HTML and CSS (selectors, cascade, flexbox, grid, tables, floats, media queries), lays pages out, interprets page scripts in a sandbox it implements itself (no JavaScript engine is involved), draws images and plays H.264 video through the runtime's player, and removes tracking parameters. History is kept only in memory, with no cache and no cookies. It also exists as a `no_std` Rust layout library for LingOS.

Why Ling fits. A browser is mostly text processing and tree walking, which Ling does well, and the headline lesson is that it is fast enough. Rules are bucketed by their rightmost class, id or tag, with ancestor filters, which makes a sheet of 20,000 rules workable. A headless mode (`--dump file:page.html`) prints the laid-out page, so the engine is tested without a window; the language test page runs 83 checks of the script interpreter and reports "all 83 checks passed" in about 280 ms.

Size and structure. 16,264 lines in `ling/`: `html`, `css` (3,250), `layout` (3,295), `jsdom` (3,964), `js` and `jsrun`, `ui` and `app`. Image and video decoding, networking and fonts are runtime builtins.

Lessons learned.

1. **Write a headless mode first.** `--dump` and `--click` made the engine testable and let a document check itself.
2. **Budget everything a page can do.** Scripts get a time allowance of a few seconds at load, 0.4 seconds per event and 60 ms per animation frame; over that, a script is stopped and the page carries on.
3. **Block by default.** `fetch`, cookies, `eval` and workers are off on purpose. The privacy features are the absence of features.
4. **Know your backend.** The browser's notes say to run it on the default JIT, because the tree-walking interpreter runs out of stack at about 40 nested calls on Windows, too few for the script sandbox. Other apps in this chapter are developed on the interpreter. Test your program on every backend you ship.

Soul Symphony, Ling Edition

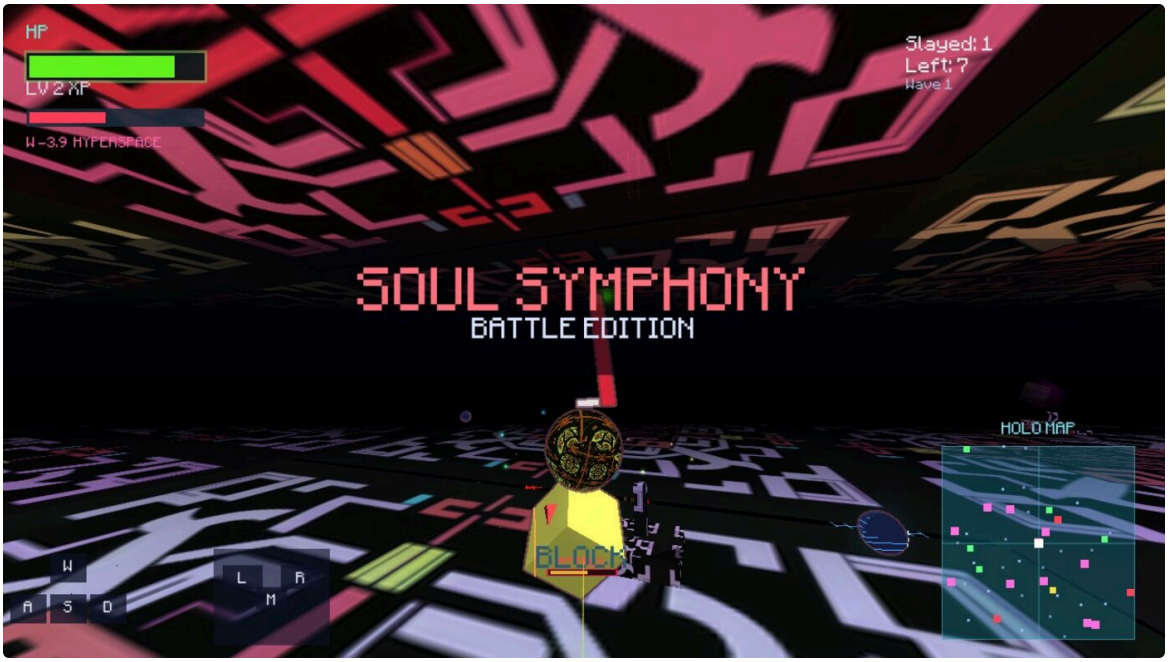


Figure 90.12. Soul Symphony Battle Edition running from a LingDev project: vector lattices, a heads-up display and a holographic map.

What it is. A spiritual 3D adventure in a soul realm: collect the letters of the world-song, clear a monster lattice and duel in an arcade arena. It is a port of the author's 2019 game with full voice-over and subtitles in six languages (1,578 lines of voice-over), and it is released for Windows, Linux and macOS.

Why Ling fits. The project describes itself as written end to end in Ling. Geometry is vector code, levels and bosses are Ling modules, and dialogue is stored per language as Ling data modules. LingDev can import the game's cutscenes, voice-over, models, music and hub level, which turns a finished game back into an editable project.

Size and structure. 47 files and about 26,000 lines in `game/` (arcade, arena, boss, companions, level, lattice, audio). The repository has more than 650 `.ling` files in all; most of the additional lines are dialogue font data for Chinese, Japanese, Korean and Thai, produced by tooling, not typed.

Lessons learned.

1. **Separate content from code.** Chapters are previews (`chapter1_preview.ling` through `chapter8_preview.ling`), so each can be tested alone.
2. **Treat localisation as data.** Six languages of text, voice and fonts are data modules selected at run time.
3. **Make a finished game importable.** Because the game is data and Ling modules, an editor can open it, which is the strongest test of an open format.

Also shipped

Smaller programs from the same workshop show Ling on the server side. They are described in their READMEs; the line counts are lines of Ling in each project.

Project	What it is	Lines of Ling
`studio.linglin.art`	A web site that sells LingStationX, hosts subscribers' cloud projects, and renders previews offline with the DAW engine (`daw_bounce`) in detached jobs; SQLite storage	about 9,100 in `app`, `controllers` and `models`
`jadepos.store`	A point-of-sale for merchants on `ling-http` and `ling-crypto`: Argon2id, BLAKE3, TOTP two-factor accounts, card validation, and only brand and last four digits stored	9,310
`fu.ling-lang.org`	The package registry for Ling, like crates.io, with two-factor accounts, written in Ling (Thai keywords)	5,620

Project	What it is	Lines of Ling
`countries.ling-gov.org`, `music.ling-lin.me`	Small site back ends in the same house style	3,594 and 1,982

What the larger programs have in common is worth stating once more. They keep a thin native layer (an audio engine, a crypto library, decoders) and write everything else, including user interfaces, file formats, parsers, renderers and test harnesses, in Ling. They share one visual design system and one set of conventions, so a skill learned in one transfers to the others. And a skill learned in one of them, such as reading a `.lmesh` or a `.lscene`, is plain Ling, so it transfers without a toolchain.

APPENDIX

Known Limitations and Gotchas

Ling is a young language that moves quickly. This appendix collects the behaviours that surprise newcomers most often, so that you can recognise them in minutes instead of hours. Everything here was verified against Ling 2030.1.53 using both backends: the default Cranelift JIT (`ling run`) and the tree-walking interpreter (`ling run --interp`).

> TIP
>
> When a program misbehaves, run it once with `ling run --interp`. The interpreter reports unknown functions and runtime errors that the JIT silently turns into `()`.

Unknown names fail silently on the JIT

On the default backend, calling a function that does not exist returns the unit value `()` and the program carries on. A typo such as `draw_rect` (which does not exist) produces no error and no drawing. The interpreter reports `unknown function`. Make `--interp` part of your edit-run loop whenever something "does nothing".

There is no assignment statement

`x = x + 1` is a parse error. Ling rebinds instead: `bind x = x + 1`. Rebinding shadows the earlier binding in the current scope.

Loops and scope

Construct	Does `bind` inside update an outer variable?
`while` body	Yes. The body shares the enclosing scope.
`if` body	Yes.
`for` body	No. Each iteration gets a fresh copy of the scope.

To accumulate a total, use a `while` loop with a counter. A `bind` inside a `do { }` block is discarded on the interpreter but leaks into the outer scope on the JIT, so do not depend on either behaviour.

Range bounds must be simple

`0..n` and `0..(n + 1)` work. `0..n + 1` is a parse error, and `0..len(xs)` silently misbehaves. Wrap the bound in parentheses: `0..(len(xs))`.

Numbers and text

- Numeric literals are decimal digits with at most one fraction. There is no `1e3`, `0xff`, `1_000`, `.5` or `3.`.
- Strings use double quotes only and have no interpolation. Build text with `+` and `str()`.
- `print` and `println` both end the line, and both join their arguments with no separator: `println("sum", 1 + 2)` prints `sum3`.
- Single-precision results can print with noise (`0.800000011920929`). Use `round(x * 100) / 100` when you display them.

Comparing structured values

`==` on lists, records and variants is always false, even for identical contents. Compare fields or elements yourself.

What the type system does and does not do

Ling is dynamically typed. Type annotations such as `x: number` and `-> T` are parsed and ignored. There is no map or dictionary type, no set, no tuple and no JSON parser. Model key-value data with parallel lists or with records, or store it in SQLite through the `db_*` builtins.

Keywords that currently do nothing

These words are reserved by the lexer but have no effect at run time:

`stop`, `again`, `change`, `post`, `give`, `fit`, `can`, `sure`, `maybe`, `pure`, `spawn`, `none`

In particular `stop` and `again` do not break out of or continue a loop. Restructure the loop condition instead. The ownership words `own`, `lend`, `share`, `move` and `copy` are parsed and ignored, with no borrow or move checking. `async` runs synchronously.

Closures

A closure passed to a builtin (`try`, `list_map`, `list_filter` and similar) works only on the interpreter. Closures held in variables are unreliable on the JIT. A named top-level `fn` works everywhere, so prefer it. The one exception is `try()`: on the JIT it always returns an error (`try: first argument must be a function`), even with a named function, and it works only under `--interp`. Programs that call `http_serve` run on the interpreter automatically, so `try` works inside request handlers.

JIT and interpreter differences at a glance

Situation	JIT (default)	Interpreter
Unknown function	returns `()` silently	error
`1 / 0`	`()`	`inf`
`pi` as a bare name	`()` (use `pi()`)	value
`list_push`	may mutate the original list	returns a new list
Empty text or list in a condition	truthy	falsy
Out-of-range index	empty value	error

Always write `bind xs = list_push(xs, v)` so that your code is correct on both.

Tooling

- `lingfu new` creates `src/start.ling`, but `lingfu build` expects `main.ling` unless the manifest says otherwise. Add `entry = "src/start.ling"` to the `[spirit]` section of `Ling.toml`.
- `lingfu check`, `lingfu fmt` and `lingfu translate` currently call `ling` subcommands that do not exist, so they report success without doing anything.
- Localised `lingfu` command names (for example `灵符 运行`) work only when the executable itself is renamed.

- Native `ling build` compiles generated Rust through Cargo and needs a Rust toolchain on the build machine.

Modules and shared code

`use "file"` flattens the loaded file into one namespace. Functions and (in the tested version) top-level `bind` constants of the loaded file are visible, but if two files define the same function name, the last one loaded silently wins, even for callers in the earlier file. Calls between sibling functions inside a `mod` block are unreliable. Prefix your function names with the library name, for example `geom_area`.

Testing

`lingfu test` runs programs on the JIT, which never fails at run time, so it can print "All tests passed!" for a program whose assertions failed. Make your tests print a clear PASS or FAIL line, and check that line, or run them under `ling run --interp`, which stops with an error. Helper files placed inside `tests/` are treated as tests and fail with "no entry point".

Performance

The JIT is more than a hundred times faster than the interpreter for arithmetic such as recursive Fibonacci, but appending to a string in a loop is quadratic on the JIT: 40,000 appends took 1.1 s, against 39 ms on the interpreter. Collect pieces in a list and join them once. Isolates run on the interpreter, so they help with blocking work but not with raw arithmetic.

Parse traps

A line that begins with `[` or `-` continues the previous expression. `wait` is reserved, and `append_file`, `panic` and `exit` are not builtins.

Graphics and audio

- Window coordinates have +y pointing down. A cone's apex points down by default.

- In `font_text`, `font_text_fill` and `ui_text`, `y` is the top of the text line, not its baseline.
- 3D calls are queued and always draw on top of 2D calls unless you call `flush_3d()`.
- Colour arguments are not clamped; 300 spills into the next channel.
- `rb_*` physics is y-up with gravity of -9.81, while `soft_*` is y-down.
- `delta_time()` is not implemented. Subtract successive `time_now()` values.

Versions

Check which `ling` you are running with the install location and the build date of the executable, and update with `cargo install ling-lang lingfu --force`. A few builtins added recently (`str_to_hex`, `hex_to_str`, `net_local_ip`, `rand_seed`, `clipboard_set`, `clipboard_get`) are missing from older installs: they return `()` on the JIT and report `unknown function` on the interpreter.