

The Lune Programming Language

A lazy functional language whose compiler teaches you

Hiroshi Koide

Lune v0.1

Contents

	Preface: Welcome to Lune	4
Part I: One Lap Around		
Chapter 1	A Gentle Tour	7
Part II: Learning the Language		
Chapter 2	Values, Types and Expressions	19
Chapter 3	Functions	30
Chapter 4	Lazy Evaluation — the Heart of Lune	40
Chapter 5	Algebraic Data Types and Pattern Matching	55
Chapter 6	Records	68
Chapter 7	Null Safety	75
Chapter 8	Lists and Streams	86
Chapter 9	Writing Imperatively — var, while, for and IO	95
Chapter 10	Modules	104
Part III: The Compiler and the Tools		
Chapter 11	Talking to the Compiler	112
Chapter 12	The REPL, Formatting and Checking	121
Chapter 13	Building a Program — Case Studies	130
	Appendix A: Language Reference Manual	140
	Appendix B: Standard Library Reference	151
	Appendix C: The Diagnostic Codes	158
	Appendix D: CLI and REPL Command Reference	163
	Appendix E: The Design of Lune, and What Comes Next	168
	Index	175

Sections within a chapter are not listed. Follow them from the sidebar in the HTML edition, or from the bookmarks in the PDF.

Preface: Welcome to Lune

A sheet of paper is pinned to the wall of the after-school clubroom:

Club motto: don't do it until you need to. Once done, never forget it.

You join expecting an easy club, and then you are told that the motto is about evaluation strategy — that is the opening scene in this language's repository. This book is what comes next.

Everything the club teaches is collected here, so that you can learn it without coming to the clubroom. The senior members do not appear in these pages, but **the compiler plays every one of their parts.**

What Lune is — Lazy and Native

Start with the name.

Lazy is the evaluation strategy. A value in Lune is not computed until it is needed. Writing `let` makes nothing happen; the language waits until the moment the value is actually used. Once computed, it remembers, and never computes it again. Exactly the motto.

Native is not native code. It is **native language**. Lune's compiler explains itself in the language you think in.

```
$ lune --check guide.lune
error[TYP0001]: undefined name: cont
  --> guide.lune:2:13
    |
2 | let total = cont + 5
    |             ^^^^ name is not defined
    = hint: did you mean `count`?
    = help: run `lune explain TYP0001` for a detailed explanation
```

This edition shows English, which is what Lune prints by default. Run `lune --check -lang ja` and the same compiler says the same thing in Japanese, down to the hint. That switch is the point of the design: the effort of decoding an error message in a

foreign language should go into learning to program instead. It is easy to forget that this is a cost at all when the messages already happen to be in your language.

The attitude of this book — errors are teaching material

One line from the club president, borrowed just once:

In this club, an error is not a failing grade. It is teaching material.

That is a design decision rather than a nice turn of phrase. Every Lune diagnostic carries three things: **what happened**, **where it happened**, and **what to do next**. In the example above, `^^^^` gives the place, `= hint:` gives the next move, and `= help:` points at a longer explanation.

All 30 diagnostic codes come with one — what it means, the smallest example that reproduces it, and how to fix it. Type `lune explain TYP0001` to read it. **An error number is not something to memorise; it is a key to look things up with.**

So this book produces errors on purpose. Every chapter has a **Break it** box.

Break it — type `1 + true` into the REPL. What comes back?

```
lune> 1 + true
error[TYP0003]: +: expected Int, got Bool
  = help: run `lune explain TYP0003` for a detailed explanation
```

If you got an error, it worked. **An error means you got it right.**

How to read this book

Read it with a REPL open. This is the one request that matters most.

```
$ lune
Lune v0.1.0 REPL. Type :help or :quit.
lune>
```

If installing is a nuisance, there is a [Playground](#) that runs in the browser — the actual implementation, running inside the page, with nothing to install. It has a REPL too. Every complete example in this book has an **Open in the Playground** link under it.

Every output printed in this book was **taken from a real run of the implementation**. You will get the same thing. If you do not, either this book is wrong or you have found something interesting.

- **Part I (chapter 1)** — one lap around the language. Rules come later; here you write, run, and break things.
- **Part II (chapters 2–10)** — the language itself. Values and types, functions, **lazy evaluation (chapter 4)**, algebraic data types and `match`, records, null safety, lists and streams, imperative code, modules.
- **Part III (chapters 11–13)** — the compiler and the tools. Reading diagnostics, the REPL and the formatter, and case studies that put it together.
- **Appendices A–E** — the reference half. Grammar, standard library, the diagnostic catalogue, the CLI and REPL, and the design rationale.

In a hurry? Chapter 1 → chapter 4 → chapters 2 and 3 → chapter 5 → chapter 8 still reaches the core of the language. **Chapter 4 (lazy evaluation) is what this language is for**, so do not skip it: with `:thunks` and `:trace` you get to watch evaluation happen.

The exercises come with answers, folded up directly beneath each question. Think first, then open.

What you need to know already

If you have written variables, functions and conditionals in some programming language, that is enough. No functional programming experience is required. You do not need to know what lazy evaluation is — chapter 4 is more fun if you don't.

Have a good club session.

Open the REPL and type `1 + 2`. Chapter 1 starts there.

Chapter 1: A Gentle Tour

There has only ever been one way to pick up a new programming language: write programs in it. In this chapter you will write Lune programs, run them, and break them. The precise rules can wait for later chapters; first, one lap around the whole language.

One thing about the manner of this book, before we start. **Do not be afraid of errors.** Lune’s compiler treats an error not as a notice of failure but as teaching material. In the second half of this chapter we will cause errors on purpose and let the compiler teach us.

1.1 Calculating in the REPL

Lune has a REPL — an interactive session. Type an expression and it evaluates it on the spot and hands back the result. That is where we begin.

```
$ lune
Lune v0.1.0 REPL. Type :help or :quit.
lune>
```

On notation — shell commands are shown after `$`, and REPL input after `lune>`. The `lune` command means `./bin/lune` in the repository (put it on your `PATH`, or `alias lune=./bin/lune`, and you can type exactly what the book types). If you installed with `pip install lune-lang`, `lune` is already on your `PATH`.

One more. This book prints the compiler’s diagnostics in **English**, which is what Lune prints by default. To see them in Japanese instead, set `LUNE_LANG=ja` or pass `--lang ja` (inside the REPL, `:lang ja`).

Type some numbers.

```
lune> 1 + 2
3 : Int
lune> 40 + 2
42 : Int
```

Results are shown as `value : type`. `3` is the value, `Int` is the type. Every expression in Lune has a type, and the REPL always tells you what it is.

Strings are written with `"` and joined with `+`.

```
lune> "hello, " + "world"
"hello, world" : String
```

To leave the REPL, `:quit` (or `:q`).

```
lune> :quit
bye
```

`:help` lists the commands. The REPL is the best laboratory you have for the whole of this book. Whenever a question comes up, type it in and find out.

1.2 Naming values with `let`

`let` gives a value a name.

```
lune> let x = 41
ok
lune> x + 1
42 : Int
```

To a declaration the REPL answers only `ok`. Notice that no value is printed. That is not laziness on the REPL's part — well, it is, but deliberately: **a Lune `let` has not computed anything at the moment it gives the name.**

Coming attraction: the computation has not happened yet — the REPL's `:thunks` command lets you look at whether a binding has been evaluated, without evaluating it.

```
lune> let y = 1 + 1
ok
lune> :thunks y
y : unevaluated
lune> y
2 : Int
lune> :thunks y
y : evaluated = 2
```

`1 + 1` was computed at the moment `y` was used, and not before. This is **lazy evaluation** — the heart of Lune's design, and the subject of chapter 4. For now, remember only that `let` writes down a promise to compute, and read on.

1.3 Writing a program to a file

The REPL is a laboratory, but programs are written to files. Create `hello.lune` in your editor.

```
module hello

def greet(name: String): String =
  "hello, " + name

let main = println(greet("world"))
```

Reading from the top:

- `module hello` — declares that this file is the module `hello`. A Lune source file always starts with this declaration.
- `def greet(name: String): String = ...` — a function definition. The parameter `name` has type `String` and so does the result. The body is indented.
- `let main = println(greet("world"))` — a top-level binding. `println` is a built-in that prints a value.

First, check the types without running anything.

```
$ lune --check hello.lune
type check OK
```

Now run it. The execution model of Lune v0.1 is “**pick one top-level binding and evaluate it**”. Pass the name of the binding to `--eval`.

```
$ lune --eval main hello.lune
hello, world
()
```

Two lines. The first is what `println` printed. The second, `()`, is the value of `main` itself — the `Unit` value that `println` returns (`--eval` prints the value of the binding it evaluated, last).

`println` prints a `String` **exactly as it is**: no quotes, and escapes such as `"a\nb"` come out as real newlines. Values that are not strings are printed in Lune’s standard display form (`show` form). To print a string in quoted `show` form, write `println(show(value))`. Chapter 2 sorts out the display rules.

1.4 if is an expression

Lune’s `if` is an **expression**: it produces a value. This is a little different from the statement it is in C or Python.

```
lune> if 5 > 3 then "big" else "small"
"big" : String
```

Being an expression, its result can be bound with `let` or used as the body of a function. Wherever a value is required, `else` cannot be omitted — without it there would be no value when the condition is false.

For longer bodies there is a block form. The REPL continues the input when a line ends in `=` or `:` (`...` is the continuation prompt; a blank line finishes).

```
lune> def abs(x: Int): Int =  
...     if x < 0:  
...         -x  
...     else:  
...         x  
...  
ok  
lune> abs(-5)  
5 : Int
```

1.5 Playing with lists — a temperature table

Introductory books on C traditionally start by building a Fahrenheit-to-Celsius table. In tribute, so will this one — but the Lune way, with a **list** instead of a loop.

Lists are written with `[ ]`. They display in Lisp-like `( )` form.

```
lune> [1, 2, 3]  
(1 2 3) : List[Int]  
lune> range(0, 5)  
(0 1 2 3 4) : List[Int]
```

`range(start, end)` builds the list of integers from `start` up to but not including `end`. The three basic tools for working on lists are `map` (apply a function to every element), `filter` (keep the elements that satisfy a condition) and `fold` (collapse the list into a single value).

```
lune> map(range(1, 6), fn x -> x * 2)  
(2 4 6 8 10) : List[Int]  
lune> filter(range(1, 10), fn x -> x % 2 == 0)  
(2 4 6 8) : List[Int]  
lune> fold([1, 2, 3, 4], 0, fn acc x -> acc + x)  
10 : Int
```

`fn x -> x * 2` is a small unnamed function — a lambda. It works without a type on the parameter because the compiler infers it from the type `map` expects (chapter 3).

Now widen that `range` a great deal.

```
lune> take(filter(range(1, 100000000), fn x -> x % 7 == 0), 5)  
(7 14 21 28 35) : List[Int]
```

Five multiples of seven out of a hundred-million-element list, and **no waiting**. Lune did not build a list of a hundred million and then take five from it; it grew the list exactly as far as taking five required. “Compute nothing until it is needed” is what Lune calls **lazy evaluation**, and it sits at the centre of the language. Chapter 4 is about nothing else.

The tools are in place. Here is the temperature table, in a file `temperature.lune`:

```
module temperature

# A table converting 0-300 degrees Fahrenheit to Celsius, in steps of 20.
def toCelsius(f: Int): Double =
  (f - 32) * 5 / 9

let fahrenheit = map(range(0, 16), fn i -> i * 20)

let table = map(fahrenheit, fn f -> (f, toCelsius(f)))
```

`#` starts a comment that runs to the end of the line. `toCelsius` returns `Double` because `/` in Lune **always returns Double**. There is no truncating-integer-division trap as in C — in exchange, `Int` and `Double` cannot be added to each other, which chapter 2 covers in detail.

In `fn f -> (f, toCelsius(f))`, the `(f, toCelsius(f))` is a **tuple**: a pair of two values. Run it:

```
$ lune --eval table temperature.lune
((0, -17.77777777777778) (20, -6.666666666666667) (40, 4.444444444444445)
(60, 15.555555555555555) (80, 26.666666666666668) (100, 37.77777777777778)
(120, 48.888888888888886) (140, 60.0) (160, 71.11111111111111) (180,
82.22222222222223) (200, 93.33333333333333) (220, 104.44444444444444)
(240, 115.55555555555556) (260, 126.66666666666667) (280,
137.77777777777777) (300, 148.88888888888889))
```

The correspondence between Fahrenheit and Celsius, as a list of `(fahrenheit, celsius)` tuples. No loop, no counter variable, no variable holding a partial result. “Take the numbers 0 to 15, multiply by 20 to get the Fahrenheit list, and map each one to a pair with its Celsius value” — the program reads like the specification. That is what writing Lune is normally like.

1.6 Your first error — a conversation with the compiler

Now for the part you have been waiting for: **let's get it wrong on purpose**. Change `hello.lune` slightly into `typo.lune`, mistyping `greeting` as `greting`.

```
module hello

let greeting = "hello, world"

let main = println(greting)
```

```
$ lune --check typo.lune
```

```
error[TYP0001]: undefined name: greting
--> typo.lune:5:20
|
5 | let main = println(greting)
|                      ^^^^^^^^ name is not defined
  = hint: did you mean `greeting`?
  = help: run `lune explain TYP0001` for a detailed explanation
```

Dissect it from the top. Every Lune diagnostic has this shape.

Line	Meaning
<code>error[TYP0001]: ...</code>	severity (error), the diagnostic code (TYP0001), and a summary
<code>--> typo.lune:5:20</code>	the place — file:line:column
<code>5 let main = ... and</code> <code>^^^^^^</code>	the offending source, and a marker under the offending part
<code>= hint: ...</code>	a suggested fix — here, “did you mean <code>greeting</code> ?”
<code>= help: run \ lune explain</code> <code>TYP0001 ``</code>	how to find out more

On notation — the path on the `-->` line is really an absolute path on your machine. The book drops the working directory.

Do what the `help` line says. **Every diagnostic code in Lune comes with an explanation written to teach.**

```
$ lune explain TYP0001
```

```
error[TYP0001]: undefined name
```

A name was used that is not bound in the current scope and is not provided by the prelude or an import.

Example that triggers it:

```
let y = x + 1      # x was never defined
```

How to fix:

Define or import the name before using it, and check the spelling.

What happened, the smallest example that reproduces it, and how to fix it. Get into the habit of reading this every time an error appears and the compiler becomes a private tutor.

A typo like this one is a mechanically applicable fix, so `lune fix` will do it for you.

```
$ lune fix typo.lune
module hello

let greeting = "hello, world"

let main = println(greeting)
```

It printed the corrected source (`--write` edits the file in place).

This loop — **break it** → **read the diagnostic** → **understand it with** `explain` → **fix it** — is how this book teaches. Every chapter has a **Break it** box for the purpose. Here is the first.

Break it — add an `Int` and a `Bool` in the REPL.

```
lune> 1 + true
error[TYP0003]: +: expected Int, got Bool
  = help: run `lune explain TYP0003` for a detailed explanation
```

TYP0003 (type mismatch) is the diagnostic you will meet most often from here on. Read its explanation now with `:explain TYP0003` (inside the REPL, `:explain` stands in for `lune explain`).

1.7 A map of what follows

That is one lap. You can already write a Lune program, type-check it, run it, and read and fix an error. The rest of the book walks slowly back over the ground this chapter sprinted across.

- **Chapters 2–3** — values, types and expressions, then functions. Partial application and the pipeline operator `|>` join the toolkit.
- **Chapter 4** — lazy evaluation. The whole of what §1.2 hinted at. This is the main reason to learn Lune.
- **Chapters 5–8** — the tools for giving data a shape: algebraic data types, pattern matching, records, null safety, and infinite lists.
- **Chapters 9–10** — writing imperatively, and splitting a program up.
- **Chapters 11–13** — putting the compiler and the tools to work, and building something larger.

Reading in order is recommended, but once you have read through chapter 4 you can pick and choose.

Summary

What you wrote	What it means
<code>let name = expr</code>	a binding (a promise to compute); evaluated when used
<code>def f(x: T): U = body</code>	a function definition
<code>fn x -> expr</code>	a lambda (an unnamed function)
<code>if c then a else b / if c: block</code>	a conditional; an expression that produces a value
<code>[1, 2, 3]</code>	a list; displayed as <code>(1 2 3)</code>
<code>(a, b)</code>	a tuple
<code>module m</code>	the module declaration; goes at the top of the file

Command	What it does
<code>lune</code>	start the REPL
<code>lune --check FILE</code>	type-check only
<code>lune --eval NAME FILE</code>	evaluate the binding NAME and print it
<code>lune explain CODE</code>	explain a diagnostic code
<code>lune fix FILE</code>	apply the mechanically applicable fixes
<code>export LUNE_LANG=ja</code>	make Japanese the default diagnostic language (<code>--lang</code> wins over it)
<code>:quit</code> <code>:help</code> <code>:thunks</code> <code>:explain</code>	REPL commands

Exercises

Exercise 1-1 (★) Change the greeting in `hello.lune` to your own name and a greeting you like, and run it. Then change the `+` in the body of `greet` to `-` and see what diagnostic comes out.

Answer

"hello, " - name is a type error.

```
error[TYP0003]: -: expected numeric type, got String
  = help: run `lune explain TYP0003` for a detailed explanation
```

`+` works for joining strings, but `-` is for numbers only. Notice how the diagnostic explains it from the operator's point of view: `-` expected a numeric type and got a `String`.

Incidentally, deleting the argument to make it `greet()` is **not** a type error (`lune --check` accepts it). Lune's functions can be given their arguments a few at a time — that is partial application, one of the stars of chapter 3.

Exercise 1-2 (★) Build the reverse of `temperature.lune`: a Celsius-to-Fahrenheit table for 0-100 degrees in steps of 20. The formula is `F = C × 9/5 + 32`.

Answer

```
module answers

# Exercise 1-2: Celsius to Fahrenheit, 0-100 degrees in steps of 20.
def toFahrenheit(c: Int): Double =
  c * 9 / 5 + 32.0

let table = map(range(0, 6), fn i -> (i * 20, toFahrenheit(i * 20)))
```

```
$ lune --eval table ex1-2.lune
((0, 32.0) (20, 68.0) (40, 104.0) (60, 140.0) (80, 176.0) (100, 212.0))
```

There is one trap. Writing `+ 32` gives `TYP0003 (+: expected Double, got Int)`. `c * 9 / 5` is already a `Double` because of the `/`, so what you add has to be written `32.0`. Lune does not silently mix `Int` and `Double`.

Exercise 1-3 (★★) For the `fahrenheits` of `temperature.lune`, use `fold` to compute the total, and then combine it with `length` to get the average.

Answer

```
module answers

# Exercise 1-3: fold the Fahrenheit temperatures into a total, then an
average.
let fahrenheit = map(range(0, 16), fn i -> i * 20)

let total = fold(fahrenheit, 0, fn acc f -> acc + f)

let average = total / length(fahrenheit)
```

```
$ lune --eval total ex1-3.lune
2400
$ lune --eval average ex1-3.lune
150.0
```

Learn the shape `fold(list, initial, fn accumulated element -> new accumulated)`. `average` comes out as `150.0` (a `Double`) because `/` always returns one.

Exercise 1-4 (★★) From the table in `temperature.lune`, use `filter` to pick out only the Fahrenheit temperatures that are below freezing in Celsius.

Answer

```
module answers

# Exercise 1-4: keep only the Fahrenheit temperatures that are below
freezing.
def toCelsius(f: Int): Double =
  (f - 32) * 5 / 9

let fahrenheit = map(range(0, 16), fn i -> i * 20)

let freezing = filter(fahrenheit, fn f -> toCelsius(f) < 0.0)
```

```
$ lune --eval freezing ex1-4.lune
(0 20)
```

Only 0 and 20 degrees Fahrenheit are below freezing. The right-hand side of the comparison is `0.0` rather than `0` for the same reason as in exercise 1-2.

Exercise 1-5 (★) Practice at predicting a diagnostic *before* asking for it. What happens if you pass two arguments, as in `greet("world", "again")`? Predict, then run it, then read `lune explain TYP0005`.

Answer

```
error[TYP0005]: expected at most 1 arguments, got 2
--> arity.lune:6:20
|
6 | let main = println(greet("world", "again"))
|                      ^^^^^^ wrong number of arguments
= help: run `lune explain TYP0005` for a detailed explanation
```

Notice the phrase “**at most** 1”. Lune’s functions can be given their arguments a few at a time (partial application, chapter 3), so too *few* is not necessarily an error, while too *many* certainly is.

More precisely — the exact specifications of what appeared in this chapter: syntax and literals in `documents/LANGUAGE_SPEC.md` §5–9, REPL display in `documents/REPL_SPEC.md` and `documents/VALUE_DISPLAY_SPEC.md`, and the diagnostic format in `documents/ERROR_DIAGNOSTICS_SPEC.md`. The code examples of this chapter live in `books/examples-en/ch01/` and are all verified against the real CLI.

Chapter 2: Values, Types and Expressions

Chapter 1 was a sprint around the language. Part II picks the pieces up one at a time and looks at them. We start with the smallest: values, their types, and the expressions that combine them.

Keep a REPL open for this chapter too. Every Lune expression can be typed straight into it, so whenever you wonder “what is the type of this?”, type it and find out.

2.1 Literals and the basic types

Lune’s literals and the types that go with them:

Literal	Type	Displayed as
42	Int	42 : Int
3.14	Double	3.14 : Double
"hello"	String	"hello" : String
true / false	Bool	true : Bool
null	Null	null : Null
()	Unit	() : Unit
(1, "a")	Tuple[Int, String]	(1, "a") : Tuple[Int, String]

A few notes.

Int does not need watching. In v0.1 `Int` is arbitrary precision, so it never overflows.

```
lune> 9223372036854775807 + 1
9223372036854775808 : Int
```

Going past the limit of a 64-bit integer is uneventful — in C this is where the number would silently turn negative.

String escapes. Inside `"` you can write `\n` (newline) and `\"`. The display form (`show` form) prints them still escaped, so do not be surprised.

```
lune> "a\nb"  
"a\nb" : String
```

That is the REPL's `show` form only. Pass the string to `println` and the raw contents come out, so `\n` becomes a real newline (as in §1.3).

Single quotes give you a String. The form `'x'` is accepted, but v0.1 has no separate character type in practice: it is the same `String` as `"x"`.

```
lune> 'x'  
"x" : String
```

`null` and `()` are different things. `null` exists solely to represent the absence of a value, and it is the star of null safety (chapter 7). `()` is the single value of type `Unit`, used as the result of functions with no meaningful value to return, such as `println`.

2.2 Operators — arithmetic, comparison, logic

The arithmetic operators are `+` `-` `*` `/` `//` `%` and unary `-` (and unary `+`). `*` `/` `//` `%` bind at the same level, more tightly than `+` `-` — mostly what school arithmetic leads you to expect, and `()` overrides it.

```
lune> 1 + 2 * 3  
7 : Int  
lune> (1 + 2) * 3  
9 : Int  
lune> 2 * -3  
-6 : Int
```

Division needs care. As chapter 1 mentioned, `/` **always returns `Double`**. Dividing an `Int` by an `Int` still gives a `Double`. For the remainder of an integer division there is `%`, which does give an `Int`.

```
lune> 7 / 2  
3.5 : Double  
lune> 7 % 2  
1 : Int
```

So what do you use when you want an integer divided by an integer to *be* an integer? **Floor division, `//`**. It keeps the type of its operands.

```
lune> 7 // 2
3 : Int
lune> 7.0 // 2.0
3.0 : Double
```

`/` and `//` look alike but are **different operators**. Remember that “division” names two things in Lune: `//` to stay in `Int`, `/` to divide as real numbers.

The rounding of `//` has one surprise in it. It rounds **towards negative infinity** (floor), not towards zero.

```
lune> -7 // 2
-4 : Int
```

If you expected `-3.5` to be “truncated” to `-3`, this is where you trip. The reason it is `-4` is that it keeps `//` and `%` consistent with each other — exercise 13-4 works through why. For now, “`//` is floor” is enough.

All three of `/` `//` `%` raise a runtime error when the right-hand side is 0 (`RUN0006`; chapter 11 covers how to read it).

And **`Int` and `Double` do not mix silently**.

```
lune> 1 + 2.0
error[TYP0003]: +: expected Int, got Double
  = help: run `lune explain TYP0003` for a detailed explanation
lune> 1 == 1.0
error[TYP0003]: ==: cannot compare Int and Double
  = help: run `lune explain TYP0003` for a detailed explanation
```

Rather than converting implicitly and handing back an answer that is roughly right, Lune asks the writer to decide which world the computation lives in. Line the literals up: `2.0` or `2`.

The comparison operators are `==` `!=` `<` `<=` `>` `>=`. `==` and `!=` work on numbers, strings and booleans. The ordering comparisons are for numbers only.

```
lune> 3 < 5
true : Bool
lune> "a" == "a"
true : Bool
lune> "a" < "b"
error[TYP0003]: <: expected numeric type, got String
  = help: run `lune explain TYP0003` for a detailed explanation
```

Compound values — tuples, lists, and the algebraic data types of chapter 5 — can be compared with `==` as well. Comparison is **structural**: separately built values are equal when their contents are.

```
lune> (1, 2) == (1, 2)
true : Bool
lune> (1, 2) == (2, 1)
false : Bool
lune> [1, 2] == [1, 2]
true : Bool
lune> Some(1) == Some(1)
true : Bool
```

But only values **of the same type** can be compared at all. Tuples of different lengths are different types, so the comparison itself is a type error, contents notwithstanding.

```
lune> (1, 2) == (1, 2, 3)
error[TYP0003]: ==: cannot compare Tuple[Int, Int] and Tuple[Int, Int, Int]
= help: run `lune explain TYP0003` for a detailed explanation
```

The logical operators are `&&` (and) and `||` (or); negation is the function `not`. `&&` binds more tightly than `||`. Thanks to lazy evaluation (chapter 4) both **short-circuit** — when the left side settles the answer, the right side is never evaluated.

```
lune> false && crash()
false : Bool
lune> true || crash()
true : Bool
lune> not(true)
false : Bool
```

Beyond these there are the null-coalescing `??` (chapter 7) and the pipeline `|>` (chapter 3). Appendix A has the full precedence table.

2.3 if is an expression — then/else and elif

To recap chapter 1: `if` is an expression that produces a value, and it has a one-line form and a block form.

```
lune> if 5 > 3 then "big" else "small"
"big" : String
```

Being an expression brings two rules with it. **The condition must be `Bool`**, and **the branches must agree in type**.

```
lune> if 1 then 2 else 3
error[TYP0003]: if condition: expected Bool, got Int
  = help: run `lune explain TYP0003` for a detailed explanation
lune> if true then 1 else "a"
error[TYP0003]: branch type mismatch: Int vs String
  = help: run `lune explain TYP0003` for a detailed explanation
```

There is no “0 is false” as in C. Write `if x % 2 == 1`, not `if x % 2`.

When there are more branches, the block form takes `elif`. Here is `grade.lune`:

```
module grade

# `if` is an expression; the block form adds branches with `elif`.
def grade(score: Int): String =
  if score >= 80:
    "pass"
  elif score >= 60:
    "retry"
  else:
    "fail"

let result = grade(75)
```

```
$ lune --eval result grade.lune
"retry"
```

However many branches there are, `grade` returns exactly one `String`. The type checker guarantees that every path produces a value — that is the comfort of an expression-oriented language.

2.4 let-in — a binding inside an expression

`let` is not only for the top level. When you want a temporary name **inside an expression**, use `let ... in ...`.

```
lune> let answer = let x = 40 in x + 2
ok
lune> answer
42 : Int
```

What follows `in` is the value of the whole expression, and `x` is visible only within it. When you want more than one name, lining up several `let` s in block form reads better.

```
module answers
```

```
# Exercise 2-3: a block-form let names the intermediate values of a BMI.
let bmi =
  let h = 1.7
  let w = 62.0
  w / (h * h)
```

```
$ lune --eval bmi ex2-3.lune
21.453287197231838
```

Put `let` s inside an indented block, and the expression at the end is the value of the block. `lune fmt` treats this shape as canonical.

2.5 Type annotations — where to write them and where to leave them out

A `let` can carry a type annotation.

```
let y: Int = 42
```

If the annotation and the actual type disagree, that is of course a type error. Here is `annot.lune`:

```
module bad

let n: Int = "hello"
```

```
$ lune --check annot.lune
```

```
error[TYP0003]: let annotation: expected Int, got String
--> annot.lune:3:14
|
3 | let n: Int = "hello"
|               ^^^^^^^^ this expression has type String
= help: run `lune explain TYP0003` for a detailed explanation
```

So where do you write annotations and where can you leave them out? The rule of thumb for v0.1:

Place	Annotation
a <code>let</code> with an obvious right-hand side	optional (inferred from the right-hand side)
the parameters and result of a <code>def</code>	write it (a recursive function's result type is required — <code>TYP0011</code>)
the parameter of a lambda <code>fn x -> ...</code>	usually inferred from context (chapter 3)

Annotations are not only for the machine. Writing `let total: Double = ...` declares your intent to the reader, and the moment the right-hand side is wrong you get the diagnostic above. **When in doubt, write it; when you want to check, use `:type`** — that is the working habit.

2.6 Tuples — a group of values of different types

The elements of a list (chapter 8) all share one type. A **tuple** bundles a fixed number of values whose types may differ.

```
lune> let pair = (1, "a")
ok
lune> pair
(1, "a") : Tuple[Int, String]
```

Values come out of a tuple by **destructuring**. There is no accessor like `.1` — you open it by naming the parts.

```
lune> let (a, b) = (10, 20)
ok
lune> a
10 : Int
lune> b
20 : Int
```

The temperature table in chapter 1 built a list of `(fahrenheit, celsius)` tuples. Carrying two values together, without ceremony, is what a tuple is for. Once you want the bundled values to have names, you want a record (chapter 6).

Break it — look at the errors from the stages *before* type checking: lexing and layout. Both belong to the **LXL** and **PRS** families of the diagnostic scheme in chapter 11.

Writing `let x = $1` uses a character Lune does not know.

```
error[LXL0001]: unexpected character '$'
--> lex.lune:3:9
|
3 | let x = $1
|           ^ unexpected character
  = help: run `lune explain LXL0001` for a detailed explanation
```

Indenting a line for no reason runs into the layout rule.

```
error[PRS0001]: expected top-level declaration, got INDENT
--> indent.lune:4:1
|
4 |   let b = 2
|   ^ unexpected token
  = help: run `lune explain PRS0001` for a detailed explanation
```

Indentation in Lune is part of the syntax, as it is in Python. Remember that a block only opens after `=` or `:`.

Summary

Fact	In one line
Basic types	<code>Int</code> (arbitrary precision) / <code>Double</code> / <code>String</code> / <code>Bool</code> / <code>Null</code> / <code>Unit</code> / <code>Tuple[...]</code>
Division	<code>/</code> is always <code>Double</code> , <code>//</code> is floor division and keeps the type, <code>%</code> is the remainder
Rounding of <code>//</code>	towards negative infinity (floor). <code>-7 // 2</code> is <code>-4</code>
<code>Int</code> and <code>Double</code>	never mix implicitly; line up the literals
<code>==</code>	compares contents (structural). Only same-type values can be compared
<code>&&</code> <code> </code>	short-circuit; <code>&&</code> binds first
<code>if</code>	condition is <code>Bool</code> , branches agree in type, <code>elif</code> for more branches
<code>let ... in</code> / block <code>let</code>	a temporary name inside an expression
Type annotations	optional on <code>let</code> , written on <code>def</code> . Check with <code>:type</code>
Tuples	<code>(a, b)</code> to bundle, <code>let (x, y) = ...</code> to open

Exercises

Exercise 2-1 (★) Predict the value and type of each expression, then check in the REPL.

```
19 / 4
19 // 4
19 % 4
1 + 2 * 3 - 4
2 * -3
true && false || true
```

Answer

`4.75 : Double` (`/` is always `Double`), `4 : Int` (`//` is floor division, so the type stays `Int`), `3 : Int`, `3 : Int`, `-6 : Int`, and `true : Bool` (`&&` goes first and gives `false`, then `|| true`).

Exercise 2-2 (★) Add a branch to `grade.lune` for “90 or above is `"excellent"`”. Check with `grade(95)` that **where** you add the branch changes the result.

Answer

Put `if score >= 90: "excellent"` **first**. Placed after `elif score >= 80: ...`, a score of 95 is caught by `score >= 80` and comes out as "pass". `if/elif` are tried from the top down — the order of the conditions is part of the specification.

Exercise 2-3 (★★) Compute the BMI ($\text{weight} \div \text{height}^2$) for a height of 1.7 m and a weight of 62.0 kg, naming the intermediate values with a block-form `let`.

Answer

```
module answers

# Exercise 2-3: a block-form let names the intermediate values of a BMI.
let bmi =
  let h = 1.7
  let w = 62.0
  w / (h * h)
```

```
$ lune --eval bmi ex2-3.lune
21.453287197231838
```

Change `1.7` to `1` and you get `TYP0003`. Keeping `Int` and `Double` apart catches this kind of unit confusion early.

Exercise 2-4 (★★) Using destructuring, build `(2, 1)` by swapping the two values of the tuple `(1, 2)`.

Answer

```
module answers

# Exercise 2-4: swap two values with a destructuring tuple binding.
let (x, y) = (1, 2)

let swapped = (y, x)
```

```
$ lune --eval swapped ex2-4.lune
(2, 1)
```

No temporary variable is needed. Open it, and bundle it back the other way round.

Exercise 2-5 (★★) Predict the result of these three expressions, then check. The third is not like the others — what happens?

```
(1, 2) == (2, 1)
[1, 2] == [1, 2]
(1, 2) == (1, 2, 3)
```

Answer

`false : Bool` (the contents are in a different order), `true : Bool` (separately built lists with the same contents are equal — structural equality), and the third is not `true` or `false` but a **type error**.

```
error[TYP0003]: ==: cannot compare Tuple[Int, Int] and Tuple[Int, Int,
Int]
= help: run `lune explain TYP0003` for a detailed explanation
```

Tuples of different lengths are different types, so asking whether they are equal is not a question that can be asked. `==` answers only between values of one type — the same design as `1 == 1.0`: what cannot be compared is rejected before the program runs.

More precisely — literals and types are in `documents/LANGUAGE_SPEC.md` §5–6; the full operator precedence table is in `documents/SYNTAX_SPEC.md` §14 (it includes future entries such as `::` and `++`, so treat `LANGUAGE_SPEC.md` as authoritative for what is implemented); `if` and `let-in` are in `documents/LANGUAGE_SPEC.md` §9. Display rules are in `documents/VALUE_DISPLAY_SPEC.md`. The code examples of this chapter live in `books/examples-en/ch02/` and are all verified against the real CLI.

Chapter 3: Functions

We have used `def` since chapter 1, but this chapter takes functions head on. There is one goal: to acquire the feeling that **a function is a value**. If a function is a value, it can be put in a variable, passed as an argument, and returned as a result. From here, Lune programs suddenly start to stretch their legs.

3.1 `def` — an anatomy of a definition

A recap, to fix the names of the parts.

```
def add(x: Int, y: Int): Int =  
  x + y
```

`add` is the function name, `x: Int`, `y: Int` are the parameters and their types, the second `Int` is the result type, and the indented part after `=` is the body. The body is an expression, and its value is the result (there is no `return` — as chapter 2 showed, everything, `if` included, is an expression).

Write the types of the parameters and the result. There are places in v0.1 where they can be left out, but leaving them out makes inference fall back to `Any` with a warning (`TYP0010`, §3.3), or become an error in a recursive function (`TYP0011`, §3.7). Above all, the type is the function's **sign over the door**.

3.2 A function is a value

A defined function can be handled like an integer or a string. It can be bound to another name, for example.

```
lune> def inc(n: Int): Int =  
...     n + 1  
...  
ok  
lune> let renamed = inc  
ok  
lune> renamed(41)  
42 : Int
```

Being a value, it has a type. Function types are written with `->`.

```
let inc: Int -> Int = fn x -> x + 1
let addA: Int -> Int -> Int = fn x y -> x + y
let addB: (Int, Int) -> Int = fn x y -> x + y
```

`Int -> Int` is “takes an `Int`, returns an `Int`”. Two parameters can be written either `Int -> Int -> Int` or `(Int, Int) -> Int`. Since `->` is right-associative, the first is `Int -> (Int -> Int)` — “takes an `Int` and returns an `Int -> Int`”. That reading leads directly into partial application in §3.5.

Ask the REPL for the type of `map`, which we have been using since chapter 1.

```
lune> :type map
map : [T, U] List[T] -> (T -> U) -> List[U]
```

The leading `[T, U]` declares “for any types `T` and `U`” (generics, chapter 5). Read out loud: “takes a list of `T` and a function from `T` to `U`, and returns a list of `U`”. A type signature makes a rather good description of a function.

3.3 Lambdas — functions without a name

`fn parameters -> expression` builds a function on the spot.

```
fn x -> x + 1           # one parameter
fn x y -> x + y         # two parameters (just list them)
fn -> 42                # no parameters
fn x: Int -> x * 2       # with a type annotation
```

The parameter’s type annotation **can be left out when the context makes it clear**. Passing a lambda to `map`, or to `applyTwice` in the next section, tells the compiler what to expect. That is why `map(range(1, 6), fn x -> x * 2)` in chapter 1 produced no warning.

Where there is no context, there is nothing to infer from.

```
lune> let f = fn x -> x * 2
warning[TYP0010]: cannot infer type of parameter x
--> <repl:5>:1:12
  |
1 | let f = fn x -> x * 2
  |           ^ parameter type falls back to Any
  = hint: add a type annotation, e.g. `fn x: Int -> ...`
  = help: run `lune explain TYP0010` for a detailed explanation
ok
```

Note that it is a **warning**, not an **error** — the binding is made and `x` falls back to `Any`. It runs, but with thinner protection from the type checker, so the habit is to do what the hint says and write `fn x: Int -> x * 2`.

3.4 Higher-order functions

A function that takes or returns a function is called a **higher-order function**. Both are ordinary to write. Here is `hof.lune`:

```
module hof

def inc(n: Int): Int =
  n + 1

# A function that takes a function.
def applyTwice(f: Int -> Int, x: Int): Int =
  f(f(x))

# A function that returns a function.
def adder(n: Int): Int -> Int =
  fn x -> x + n

let fortyTwo = applyTwice(inc, 40)

let seven = adder(3)(4)
```

```
$ lune --eval fortyTwo hof.lune
42
$ lune --eval seven hof.lune
7
```

Playing with it in the REPL:

```
lune> applyTwice(fn x -> x * 10, 4)
400 : Int
lune> adder(3)(4)
7 : Int
```

The lambda `fn x -> x + n` returned by `adder(3)` **remembers** the `n = 3` it was created with. A function that carries the environment of its definition around with it is called a **closure**. It is the same mechanism as the thunk of chapter 4, which is a parcel of “expression plus environment”: Lune’s world is made of expressions and environments.

3.5 Partial application — you may pass only some of the arguments

In the exercises of chapter 1 we met the oddity that calling a one-parameter function with no arguments, `greet()`, is not a type error. This section is the explanation.

In Lune, **a call with too few arguments is not an error; it returns a function waiting for the rest.**

```
lune> def add(x: Int, y: Int): Int =
...     x + y
...
ok
lune> let plusTen = add(10)
ok
lune> :type plusTen
plusTen : Int -> Int
lune> plusTen(32)
42 : Int
```

Remember reading `add : Int -> Int -> Int` as “give it an `Int` and an `Int -> Int` comes back”. `add(10)` just does exactly that. Passing them together as `add(10, 32)` or separately as `add(10)(32)` is the same thing — and that flexibility lets you build “the existing function with some arguments fixed” in one line.

```
lune> map([1, 2, 3], plusTen)
(11 12 13) : List[Int]
```

Passing too *many*, on the other hand, is still an error (`TYP0005` — and now the phrase “at most 1” makes sense). Arguments passed in a partial application are captured as

thunks just like those of an ordinary call: laziness holds consistently inside partial application too.

3.6 The pipeline `|>` — writing the flow of data

When function applications nest, the reading order runs against the execution order.

`double(inc(inc(5)))` means “take 5, inc, inc, double”, but you have to read it backwards. The **pipeline operator** `|>` fixes that: `x |> f` is sugar for `f(x)`.

```
lune> 5 |> inc
6 : Int
lune> 5 |> inc |> double
12 : Int
```

Left to right, in the order the data passes through the transformations. And piping into a function of several parameters gives a **partial application**. Here is

`pipeline.lune`:

```
module pipeline

def inc(n: Int): Int =
  n + 1

def double(n: Int): Int =
  n * 2

def add(x: Int, y: Int): Int =
  x + y

# `x |> f` is sugar for `f(x)`: read left to right, in the order the data
# flows.
let result = 5 |> inc |> double

# Piping into a function of several parameters gives a partial
# application.
let addFive = 5 |> add

let eleven = addFive(6)
```

```
$ lune --eval result pipeline.lune
12
$ lune --eval eleven pipeline.lune
11
```

The pipeline is the syntax for writing a recipe of transformations. It comes into its own in chapter 8, combined with list processing.

3.7 Recursion

A function that calls itself — the functional way of writing a loop. Fibonacci is the classic.

```
def fib(n: Int): Int =  
  if n <= 1 then n else fib(n - 1) + fib(n - 2)
```

```
lune> fib(10)  
55 : Int
```

A recursive function comes with one obligation: **write the result type**. Here is what happens without it, in `norettype.lune`:

```
module bad  
  
def fact(n: Int) =  
  if n == 0 then 1 else n * fact(n - 1)
```

```
$ lune --check norettype.lune
```

```
error[TYP0011]: recursive function requires a return type annotation: fact  
--> norettype.lune:3:1  
|  
3 | def fact(n: Int) =  
|   ^^ the function calls itself before its type is known  
  = hint: add a return type, e.g. `def fact(...): T = ...`  
  = help: run `lune explain TYP0011` for a detailed explanation
```

The diagnostic explains the reason too. `fact(n - 1)` appears before the type of the body has been inferred, so the type checker needs to know the type of `fact` in advance. As chapter 4 shows, only functions may recurse (a recursive *value* gives `RUN0005`), and a recursive function must carry its type on the sign over the door. Those two rules together make recursion safe to use.

One tip on writing them: **write the stopping condition first**. Build the shape `if n == 0 then ... else recursive-call` and only then fill in the recursive side, and infinite recursion becomes harder to write.

Break it — calling something that is not a function has a diagnostic of its own.

```
lune> let x = 42
ok
lune> x(1)
error[TYP0004]: value is not callable: Int
--> <repl:7>:1:1
|
1 | x(1)
  | ^ this value is not callable
    = help: run `lune explain TYP0004` for a detailed explanation
```

“An `Int` cannot be called” — you usually meet this one after a stray pair of parentheses (the reverse of meaning `f(x)(y)` and writing `f(x, y)`).

Summary

Concept	In one line
<code>def f(x: T): U</code> <code>= body</code>	a function definition; the type is the sign over the door, and a recursive function must have a result type
Function types	<code>Int -> Int</code> ; <code>-></code> is right-associative, matching how partial application reads
<code>fn x -> expr</code>	a lambda; the annotation is optional with context, <code>TYP0010</code> without
Higher-order functions	take or return functions; a returned lambda remembers its environment (a closure)
Partial application	a call with too few arguments is a function waiting for the rest
<code>x > f</code>	sugar for <code>f(x)</code> ; into a multi-parameter function it is a partial application
<code>:type f</code>	the habit of checking a function’s type

Exercises

Exercise 3-1 (★) Rewrite `double(inc(inc(5)))` as a pipeline (`inc` and `double` as in the text).

Answer

```
lune> 5 |> inc |> inc |> double
14 : Int
```

`double(inc(inc(5)))` is `14 : Int` as well. Same computation, but the piped version reads in the order it runs: $5 \rightarrow +1 \rightarrow +1 \rightarrow \times 2$.

Exercise 3-2 (★★) Write `compose`, which composes two functions: `compose(f, g)` returns a single function that applies `g` and then `f`.

Answer

```
module answers

# Exercise 3-2: function composition. compose(f, g) is "g, then f".
def compose(f: Int -> Int, g: Int -> Int): Int -> Int =
  fn x -> f(g(x))

def inc(n: Int): Int =
  n + 1

def double(n: Int): Int =
  n * 2

let incThenDouble = compose(double, inc)

let answer = incThenDouble(5)
```

```
$ lune --eval answer ex3-2.lune
12
```

The returned lambda remembers `f` and `g` — this is practice with closures (§3.4) as much as with composition.

Exercise 3-3 (★★) Write `fib` yourself and check `fib(10)`. Then delete the result type `: Int`, predict which diagnostic you will get, and run `--check`.

Answer

```
module answers

# Exercise 3-3: Fibonacci by recursion. The result type Int is required
(TYP0011).
def fib(n: Int): Int =
  if n <= 1 then n else fib(n - 1) + fib(n - 2)

let answer = fib(10)
```

```
$ lune --eval answer ex3-3.lune
55
```

Deleting the result type gives `TYP0011`, the same diagnostic as in §3.7.

Exercise 3-4 (★★★) Generalise `applyTwice` into `applyN(f, n, x)`, which applies the function `f` to `x` `n` times. You have it right when `applyN(double, 3, 1)` is `8`.

Answer

```
module answers

# Exercise 3-4: applyTwice generalised – apply f n times.
def applyN(f: Int -> Int, n: Int, x: Int): Int =
  if n == 0 then x else applyN(f, n - 1, f(x))

def double(n: Int): Int =
  n * 2

let answer = applyN(double, 3, 1)
```

```
$ lune --eval answer ex3-4.lune
8
```

Higher-order functions and recursion together. Write the stopping condition first (`n == 0` returns `x` unchanged), so that the recursive side reads as “apply once, and count down”.

Exercise 3-5 (★, in reverse) Write the smallest piece of code that produces `TYP0004` (value is not callable).

Answer

The smallest is `42(1)` (`error[TYP0004]: value is not callable: Int`). Any non-function value — a number, a string, a tuple — with `(...)` after it will do.

Conversely, if you can explain why too *few* arguments, as in `greet()`, is neither `TYP0004` nor `TYP0005`, you have graduated from this chapter.

More precisely — function definitions, lambdas and partial application are in `documents/LANGUAGE_SPEC.md §8`; the syntax of function types is in `documents/FUNCTION_TYPE_SPEC.md`; how partial application relates to laziness is in `documents/LAZY_EVALUATION_SPEC.md §7`; type inference for lambdas is in `documents/LOCAL_TYPE_INFERENCE_SPEC.md`. The code examples of this chapter live in `books/examples-en/ch03/` and are all verified against the real CLI.

Chapter 4: Lazy Evaluation — the Heart of Lune

Most programming languages compute in the order you wrote. Write `let x = expensive computation` and the expensive computation runs on that line — this is **strict evaluation**. Lune is different. **A value is not computed until it is needed**. This is **lazy evaluation**, and it lies under every other design decision in the language.

Lazy evaluation is sometimes called a trap for beginners, because you cannot see when the computation runs. Lune's answer was not to hide it but to **make it visible**. This chapter uses two instruments in the REPL, `:thunks` and `:trace`, so that you can watch the moment of evaluation with your own eyes. You do not have to take any of it on faith; all of it can be observed on the spot.

4.1 A let is a promise

In chapter 1 we saw the REPL answer nothing but `ok` to a `let`. It prints no value because **there is not a value yet**. Let us confirm that. Here we meet a built-in for observation, `crash()`: a function that always raises a runtime error when evaluated — a landmine.

```
lune> let boom = crash()
ok
lune> boom
error[RUN0006]: crash() was evaluated
  = help: run `lune explain RUN0006` for a detailed explanation
```

Nothing happened on the line that bound the mine. It went off the moment `boom` was **used**.

`let name = expr` is not an instruction but a promise: “when the value of `name` is needed, compute it with this expression”. If a strict language's `let` is a receipt, Lune's is an order slip.

4.2 Thunks — what is inside the promise

The container for such a not-yet-computed expression is called a **thunk**. A thunk holds an expression together with its environment, and is in one of three states.

State	Meaning
unevaluated	not needed even once so far
evaluated = value	computed; remembers the value
failed = error	computed and failed; remembers the failure

The REPL's `:thunks` command looks at the state of a binding **without causing any evaluation**. Let us watch the life of a thunk.

```
lune> let x = 1 + 1
ok
lune> :thunks x
x : unevaluated
lune> x
2 : Int
lune> :thunks x
x : evaluated = 2
```

`1 + 1` was computed at the instant `x` was referred to. And what became of `boom`?

```
lune> :thunks boom
boom : failed = error[RUN0006] crash() was evaluated
```

The failure is remembered too. §4.5 makes clear what that is worth.

Terminology — making a thunk evaluate is called **forcing** it. “Force `x`” = “make `x` keep its promise now”.

4.3 When does evaluation happen — watching with `:trace`

So when does forcing happen? Type `:trace on` and the REPL narrates every force as it occurs.

```
lune> :trace on
trace on
lune> let x = 1 + 1
ok
lune> x + 1
force x + 1
  force 1 + 1
  => 2
=> 3
3 : Int
```

Read it like this:

- `force expr` — evaluation of that expression's thunk has begun (indentation is nesting depth)
- `=> value` — the matching force has finished

Notice that the line `let x = 1 + 1` produced **no trace at all**. A declaration causes no forcing. The moment you typed `x + 1`, the whole expression was forced, and inside it the contents of `x`, `1 + 1`, were forced.

Use `x` a second time and a third kind of event appears.

```
lune> x
force x
  memo 1 + 1 => 2
=> 2
2 : Int
```

`memo` marks “we hit an already-computed value and used it instead of recomputing”.

Roughly, **forcing happens wherever a value is looked at**: the operands of arithmetic and comparison, the condition of an `if`, the subject of a `match`, and display in the REPL or when running a file. Appendix A has the exact list, but in practice asking `:trace` is the surest way.

The same trace is available when running a file. Here is `trace_demo.lune`:

```
module trace_demo

let x = 1 + 1

let answer = x + x
```

```
$ lune --eval answer --trace trace_demo.lune
force x + x
  force 1 + 1
    => 2
  memo 1 + 1 => 2
=> 4
4
```

`x` is used twice but computed once; the second time is a `memo`. (The trace goes to `stderr`, so redirecting separates it from the value.)

4.4 Arguments are lazy too — writing your own control structures

It is not only `let` that is lazy. **Function arguments are passed as thunks by default.**

```
lune> def first(a: Int, b: Int): Int =
...     a
...
ok
lune> first(10, crash())
10 : Int
```

We passed `crash()` and nothing exploded. `b` is never used in the body, so it never got the chance to be forced.

This means more than saving work. **Control structures can be written as ordinary functions.** Here is `myif.lune`:

```
module myif

# Arguments are lazy by default, so control structures can be defined by
hand.
def myIf(c: Bool, a: Int, b: Int): Int =
  if c then a else b

def myAnd(a: Bool, b: Bool): Bool =
  if a then b else false

let taken = myIf(true, 1, crash())

let skipped = myAnd(false, crash())
```

```
$ lune --eval taken myif.lune
1
$ lune --eval skipped myif.lune
false
```

`myIf` never evaluates the branch it did not take. `myAnd` never looks at its right operand when the left is `false` — short-circuiting. In a strict language, `&&` and `if` short-circuit only because the language treats them specially, and a user cannot write the same thing as a function. In Lune, `if` and `&&` can in principle be reproduced as plain functions. That the core of the language can stay small is thanks to lazy evaluation.

4.5 Memoisation — never computed twice

Let us pin down the `memo` of §4.3. Here is the second instrument, `tick()` : a built-in that increments an internal counter each time it is called and returns the new value. Its partner `tickCount()` reads the counter without incrementing it. Together they are a witness to how many times a computation ran.

```
lune> let t = tick()
ok
lune> tickCount()
0 : Int
lune> t
1 : Int
lune> t
1 : Int
lune> tickCount()
1 : Int
```

Binding alone leaves the counter at 0. The first use of `t` ran it exactly once, and the second use returned the remembered value. **A thunk is evaluated at most once.**

Failure works the same way. §4.2 said the `failed` state “remembers the failure”. Here is the evidence.

```
lune> let bad = tick() + crash()
ok
lune> tickCount()
0 : Int
lune> bad
error[RUN0006]: crash() was evaluated
  = help: run `lune explain RUN0006` for a detailed explanation
lune> tickCount()
1 : Int
lune> bad
error[RUN0006]: crash() was evaluated
  = help: run `lune explain RUN0006` for a detailed explanation
lune> tickCount()
1 : Int
```

The second `bad` produced the same error, but the counter stayed at 1 — **the failed computation was not re-run**. The remembered failure was re-sent.

Because of this, a Lune binding gives the same result (the same value, or the same failure) however often and whenever it is read. Nothing changes with the timing of the read, which is why laziness does not break the meaning of a program.

4.6 Only what you use — what laziness buys

So far we have looked at the *machinery* of laziness. This section is about what the machinery gives you.

Take `range(start, end)` from chapter 1 and call it with a slightly unreasonable width.

```
lune> let huge = range(1, 100000000)
ok
lune> take(huge, 5)
(1 2 3 4 5) : List[Int]
lune> take(filter(huge, fn x -> x % 7 == 0), 3)
(7 14 21) : List[Int]
```

Five elements out of a hundred-million-element list, then three multiples of seven. **Both come back instantly**, and neither costs any memory.

The trick is the one from §4.1. `range` does not build a hundred million cells and then hand them over; it returns **one cell and a promise to make the rest**. `filter` and `take` have the same shape, so the moment `take(..., 3)` has its third element,

every promise beyond it is left unforced. What is never built costs neither time nor memory.

It is not “build a list of a hundred million and then take five”. It is “`range` grows exactly as far as taking five requires”. **The consumer decides how long the list is.**

Let `tick()` from §4.5 do the counting.

```
lune> let xs = map(range(1, 100000000), fn x -> tick())
ok
lune> tickCount()
0 : Int
lune> take(xs, 3)
(1 2 3) : List[Int]
lune> tickCount()
3 : Int
```

A `map` over a hundred million elements, evaluated **three** times — and zero times until something asked. In a strict language that expression means a hundred million calls and several gigabytes. Here it meant three calls.

The boundary — a computation that needs everything still walks everything

Laziness does not make work disappear. It **defers the demand**. When an answer genuinely needs every element, the bill arrives in full.

```
lune> length(filter(huge, fn x -> x < 10))
```

That took about **seven minutes** here (Ctrl-C is a reasonable response). The answer is `9`. `filter` cannot know that nothing beyond 9 matches without walking to the hundred-millionth element, and `length` counts to the end, so the whole list gets walked. This is not a weakness of the implementation but the **correct consequence** of lazy evaluation: the same expression in Haskell walks just as far.

The condition here, `x < 10`, is monotone over an ascending list, so what we actually meant was “stop when it stops holding”. There is a different function for saying that.

```
lune> takeWhile(huge, fn x -> x < 10)
(1 2 3 4 5 6 7 8 9) : List[Int]
```

`takeWhile` gives up at the first `false`, so the **whole** list comes back at once. `filter` can never finish without looking at everything; `takeWhile` can finish. In a lazy world

that distinction shows up as a difference in speed.

For the same reason, do not ask the REPL to display `huge` itself: displaying demands every element. Put a `take` in front of anything you print. This test — **does the answer need every element?** — is treated properly in §8.5, against genuinely infinite lists, because a range wide enough is the same thing in practice.

4.7 The strictness toolbox

Lazy by default. But there are moments when you want it computed now, and Lune provides explicit opt-outs, graded by scope.

Tool	Where you write it	Meaning
<code>strict let x = expr</code>	a binding	evaluate at the moment of binding
<code>strict x: T (short form !x)</code>	a function parameter	evaluate at the moment of the call
<code>strict field: T</code>	a constructor field	evaluate at the moment of construction
<code>seq a b</code>	an expression	evaluate <code>a</code> , then return <code>b</code>
<code>deepForce x</code>	an expression	evaluate <code>x</code> all the way down
<code>lazy expr / force expr</code>	an expression	carry laziness around as the type <code>Lazy[T]</code>

strict let — evaluates at binding time. Under `:thunks` it is not even a thunk any more.

```
lune> strict let s = tick()
ok
lune> tickCount()
1 : Int
lune> :thunks s
s : value = 1
```

strict parameters — rewrite §4.4's `first` with strict parameters and even the unused argument is evaluated at the call.

```
lune> def strictFirst(strict a: Int, strict b: Int): Int =
...     a
...
ok
lune> strictFirst(10, crash())
error[RUN0006]: crash() was evaluated
= help: run `lune explain RUN0006` for a detailed explanation
```

strict fields — constructor fields are normally lazy as well (`Box(crash())` can be built without exploding), but `strict` makes them evaluate at construction.

```
module point

# A strict field is evaluated at the moment the value is constructed.
type Point =
  | Point(strict x: Int, strict y: Int)

let p = Point(crash(), 0)
```

```
$ lune --eval p point.lune
error[RUN0006]: crash() was evaluated
= help: run `lune explain RUN0006` for a detailed explanation
```

“Never let a `Point` holding an invalid value exist, even for an instant” — this returns in chapters 5 and 6 as a tool for designing data types.

seq — `seq a b` evaluates `a` and then returns `b`. Use it when you want to control the order of evaluation and nothing else.

```
lune> let a = tick()
ok
lune> let answer = seq a 42
ok
lune> answer
42 : Int
lune> :thunks a
a : evaluated = 1
```

The moment `answer` was used, `a` — irrelevant to the value — was dragged along and evaluated.

deepForce — forcing normally evaluates only the outside of a value. When you want it evaluated all the way through, use `deepForce`.

```
lune> let pair = (tick(), tick())
ok
lune> :thunks pair
pair : unevaluated
lune> deepForce pair
(2, 3) : Tuple[Int, Int]
lune> :thunks pair
pair : evaluated = (2, 3)
```

lazy / force — “if everything is lazy, why is there a `lazy ?`” `lazy expr` is the tool for carrying laziness around **as a type**.

```
lune> let delayed = lazy (40 + 2)
ok
lune> :type delayed
delayed : Lazy[Int]
lune> force delayed
42 : Int
```

`Int` and `Lazy[Int]` are different types. Writing `Lazy[T]` as a function’s result or a field of a data structure puts “this may not have been computed yet” into the type signature, and whoever receives it has to open it explicitly with `force`. It promotes implicit laziness into a contract.

4.8 Bottomless recursion — RUN0005

Here is the notorious trouble spot of lazy evaluation: what happens when you define something in terms of itself?

```
module recursive

let x = x + 1
```

To know the value of `x` you need the value of `x` — waiting will not produce an answer. Type checking rejects this straightforward form at the door (the name being defined is not yet in scope on the right-hand side, so it is an undefined name).

```
$ lune --check recursive.lune
```

```
error[TYP0001]: undefined name: x
--> recursive.lune:3:9
|
3 | let x = x + 1
|      ^ name is not defined
= help: run `lune explain TYP0001` for a detailed explanation
```

But a cycle can also slip past the static check and appear only at run time. `--eval` in v0.1 runs without type checking, so this same file can show you. Will it loop forever?

```
$ lune --eval x recursive.lune
```

```
error[RUN0005]: recursive thunk evaluation: this value's definition
depends on its own result
= hint: recursive values cannot be computed; use a recursive function
(`def`) instead, or break the reference cycle
= help: run `lune explain RUN0005` for a detailed explanation
```

It will not. **It is detected immediately.** The mechanism is in the thunk states of §4.2: on entering evaluation, a thunk marks itself “being evaluated” internally. If the same thunk is forced again during that evaluation, the definition has come back round to itself — no amount of waiting would help — so `RUN0005` is reported on the spot. `lune explain RUN0005` explains that mechanism as part of the diagnostic.

Aside: `<<loop>>` — Haskell (GHC), the home of lazy evaluation, has a similar detection, but reports it with a curt `<<loop>>`, and sometimes fails to detect it and really does loop. Lune turned the detection into teaching material with a diagnostic code, so that `explain` can also tell you *why* it need not loop forever.

Note that **recursive functions are perfectly fine.**

```
lune> def fact(n: Int): Int =
...     if n == 0 then 1 else n * fact(n - 1)
...
ok
lune> fact(5)
120 : Int
```

The body of a `def` does not run until it is called, so `fact(n - 1)` does not force the definition of `fact`. It is recursive *values* that cannot exist.

4.9 Living with laziness

The tools are all here. To finish, some guidance on when to use what.

Where laziness pays:

- **Values that may not be used** — defaults, error messages, one side of a branch. Writing them costs nothing.
- **Your own control structures and short-circuiting logic** (§4.4).
- **Infinite data structures** — the crown jewel. Chapter 8 awaits.

Where to choose strictness:

- **Side effects whose order matters** — output that runs “in the order it happened to be needed” is bewildering. Keep IO strict (chapter 9).
- **Measurement and debugging** — when you want to time “this line” and the computation keeps escaping backwards, nail it down with `strict let` or `seq`.
- **Piled-up thunks** — a loop can grow a towering stack of unevaluated computations. We revisit this with `fold` in chapter 8.

And when in doubt, **ask** `:thunks` and `:trace`. The instruments you used in this chapter are the debugging tools you will use in earnest.

Break it — bind an expression that divides by 0 and watch the life of the thunk.

```
lune> let half = 1 / 0
ok
lune> half
error[RUN0006]: division by zero
  = hint: the right operand of `/` evaluated to 0
  = help: run `lune explain RUN0006` for a detailed explanation
lune> :thunks half
half : failed = error[RUN0006] division by zero
```

Nothing at the binding (lazy), a failure on use (force), and the failure remembered (memoisation). The whole chapter in three lines.

Summary

Concept	In one line
Lazy evaluation	a value is not computed until it is needed
Thunk	the container of an unevaluated expression: <code>unevaluated</code> / <code>evaluated</code> / <code>failed</code>
Forcing	making a thunk evaluate; happens wherever a value is looked at
Memoisation	evaluated at most once; both success and failure are remembered
Only what you use	the consumer decides the length; the width of a <code>range</code> is free
Does it need everything?	if it does, it walks it all (<code>length</code> , <code>fold</code> , <code>display</code>); if not, <code>take</code> / <code>takeWhile</code>
<code>:thunks</code> / <code>:trace</code>	look at the state / narrate the forcing. When in doubt, these
<code>strict let</code> / <code>strict parameters</code> / <code>strict fields</code>	opting out of laziness
<code>seq</code> / <code>deepForce</code>	controlling the order and the depth of evaluation
<code>lazy</code> / <code>force</code> and <code>Lazy[T]</code>	making laziness explicit in the type
<code>RUN0005</code>	a recursive value cannot exist; make it a function

Exercises

Exercise 4-1 (★) Typing the following in order, what does each line print? Predict, then check.

```
lune> let u = tick()
lune> let v = tick()
lune> v
lune> :thunks
lune> u
```

Answer

```
lune> let u = tick()
ok
lune> let v = tick()
ok
lune> v
1 : Int
lune> :thunks
u : unevaluated
v : evaluated = 1
lune> u
2 : Int
```

`u`, defined first, is 2; `v`, defined second, is 1. The counter advances in the order things are **used**, not the order they are **defined**. What decides the order of evaluation is not the layout of the program but demand — the core of lazy evaluation.

Exercise 4-2 (★★) Following `myAnd` (§4.4), write `myOr`, which returns `true` without evaluating its right operand when the left one is `true`. You pass when `myOr(true, crash())` does not explode.

Answer

```
module answers

# Exercise 4-2: myOr short-circuits – a true on the left never evaluates
the right.
def myOr(a: Bool, b: Bool): Bool =
  if a then true else b

let shortCircuited = myOr(true, crash())
```

```
$ lune --eval shortCircuited ex4-2.lune
true
```

`if` evaluates only the condition `a`. When the chosen branch is the literal `true`, the thunk for `b` is never even touched.

Exercise 4-3 (★★) Define one `let t = tick()` and one `strict let s = tick()`, and use `tickCount()` and `:thunks` to explain the difference.

Answer

Reading `tickCount()` straight after the definitions gives 1 — only the `strict let` has run. Under `:thunks`, `t` : `unevaluated` against `s` : `value = 1`. `s` never went through a thunk; it became a value at the moment of binding. `t` is computed later, the first time it is used.

Exercise 4-4 (★★) List elements are lazy as well. So does `length(xs)` succeed for `let xs = [crash(), 1, 2]` ? Predict, then check.

Answer

```
lune> let xs = [crash(), 1, 2]
ok
lune> length(xs)
3 : Int
```

It succeeds. `length` counts only the **spine** of the list (the chain of cells) and never forces the **elements**. That distinction — evaluate the structure, not the contents — is exactly the mechanism behind the infinite lists of chapter 8.

Exercise 4-5 (★★★, in reverse) Produce a real `RUN0005`. Then explain why `--check` on the same file reports `TYP0001` rather than `RUN0005`.

Answer

The smallest reproduction is a file containing `let x = x + 1`, run with `--eval x` (§4.8). `--check` gives `TYP0001` because the type checker works by the rule that the name being defined is not yet in scope while its right-hand side is checked, so self-reference is rejected at the door as an undefined name. The type checker prevents cycles of names statically; the evaluator detects a cycle that appears at run time with `RUN0005`. Two layers of defence.

More precisely — the complete list of thunk state transitions and forcing boundaries is in `documents/LAZY_EVALUATION_SPEC.md`; the display specification for `:thunks` and `:trace` is in `documents/REPL_SPEC.md` §5.1–5.2. `crash`, `tick` and `tickCount` are the observation built-ins of `documents/STANDARD_LIBRARY_SPEC.md` §8.1. The code examples of this chapter live in `books/examples-en/ch04/` and are all verified against the real CLI.

Chapter 5: Algebraic Data Types and Pattern Matching

The types so far — numbers, strings, tuples — were raw material. In this chapter you make **types for your own program**. There are two tools: **algebraic data types** (ADTs), which declare the shape of data, and `match`, which takes a shape apart.

Learning these two changes how you write. Instead of checking a value and branching on a flag, you write down every possible shape in the type and let the compiler watch for missing cases. By the end of this chapter you will have seen Lune report a missing case **with a counterexample**.

5.1 Expressing a shape as a type

Consider a program about figures. A figure is either a circle or a rectangle. In Lune you write that “either” directly.

```
type Shape =  
  | Circle(radius: Double)  
  | Rect(width: Double, height: Double)
```

A value of type `Shape` is built with one of the **constructors**, `Circle` or `Rect`. No other shape is possible.

```
lune> Circle(1.0)  
Circle(1.0) : Shape  
lune> Rect(3.0, 4.0)  
Rect(3.0, 4.0) : Shape
```

Fields are declared with names (`radius: Double`). The names are there to make the declaration readable; values are built by position.

Constructors can also have no fields at all — plain alternatives, like the colours of a traffic light.

```
type Color =  
  | Red  
  | Green  
  | Blue
```

```
lune> let g = Green()  
ok  
lune> g  
Green : Color
```

In v0.1, even a field-less constructor is **called** to produce a value (`Green()`). The exceptions are the prelude's `None` and `Nil` (the end of a list, chapter 8), which are registered as values from the start and can be used bare.

Terminology — a type like `Shape`, meaning “exactly one of these”, is called a **sum type**. Its counterpart is a product type, meaning “all of these at once”, such as a tuple or a record. Types built by combining the two are what “algebraic data type” refers to. Chapter 6 ends with guidance on choosing between them.

5.2 match — taking a shape apart

The other side of an ADT is `match`. Here is a proper look at the construct chapter 1 kept hinting at.

```
def area(s: Shape): Double =  
  match s:  
    | Circle(r) -> r * r * 3.14159  
    | Rect(w, h) -> w * h
```

`match` examines the **shape** of the subject from the top down and evaluates the arm that first fits. A pattern such as `Circle(r)` both confirms the shape and **extracts** the contents under the name `r`. Checking and destructuring in a single move — that is the pleasure of pattern matching.

`match` is an expression too, so it produces a value. An arm's body can span several lines (indented).

Many kinds of pattern can go in there. All at once:

```
lune> def sign(n: Int): String =
...     match n:
...       | 0 -> "zero"
...       | x if x < 0 -> "negative"
...       | _ -> "positive"
...
ok
lune> sign(0)
"zero" : String
lune> sign(-5)
"negative" : String
lune> sign(3)
"positive" : String
```

- **literal pattern** `0` — matches exactly that value
- **name pattern** `x` — matches anything and names the value
- **guard** `x if x < 0` — an extra condition on a pattern
- **wildcard** `_` — matches anything, names nothing

Tuples can be taken apart as well.

```
lune> def where(p: Tuple[Int, Int]): String =
...     match p:
...       | (0, 0) -> "origin"
...       | (x, 0) -> "x-axis"
...       | _ -> "elsewhere"
...
ok
lune> where((3, 0))
"x-axis" : String
```

Guards come into their own together with an ADT such as `Shape`. From `shape.lune`:

```
# A guard adds an extra condition to a pattern.
def describe(s: Shape): String =
  match s:
    | Circle(_) -> "circle"
    | Rect(w, h) if w == h -> "square"
    | Rect(_, _) -> "rectangle"
```

```
$ lune --eval squareness shape.lune
"square"
```

5.3 Exhaustiveness — the compiler hands you a counterexample

Here is where Lune shows off. A `match` must be **exhaustive**: not covering every possible shape is a type error. And Lune tells you what is missing with a concrete counterexample (a *witness*).

Forget one case of `Color`:

```
lune> def label(c: Color): String =
...   match c:
...     | Red -> "warm"
...     | Blue -> "cool"
...
error[TYP0007]: non-exhaustive match: missing case Green
--> <repl:13>:2:5
|
2 |   match c:
|     ^^^^^ pattern Green is not covered
= hint: add a case for Green, or a wildcard case `| _ -> ...`
= help: run `lune explain TYP0007` for a detailed explanation
```

“`Green` is missing.” You could probably have spotted that one yourself; the real worth of a witness shows when shapes **nest**. Here is `missing.lune`:

```
module bad

# The Some(false) case is missing.
def toInt(o: Option[Bool]): Int =
  match o:
    | Some(true) -> 1
    | None -> 0
```

```
$ lune --check missing.lune
```

```
error[TYP0007]: non-exhaustive match: missing case Some(false)
--> missing.lune:5:5
|
5 |   match o:
|     ^^^^^ pattern Some(false) is not covered
= hint: add a case for Some(false), or a wildcard case `| _ -> ...`
= help: run `lune explain TYP0007` for a detailed explanation
```

You wrote `Some` and you wrote `None` — and still the compiler **synthesises the counterexample** `Some(false)`. At this point exhaustiveness checking is not a

supervisor but a tool for finding holes in a specification. It asks, with an example in hand, “did you think about this input?”

Two cautions. First, **an arm with a guard does not count towards exhaustiveness**: the compiler cannot prove that `| x if x < 0 -> ...` covers every negative number. When you use a guard, always leave an unconditional arm to catch what falls through. Second, `| _ -> ...` satisfies exhaustiveness in one stroke, but at a price: **you stop being told when a case is added to the type**. Enumerate when you can enumerate — exercise 5-1 lets you feel the difference.

5.4 Unreachable arms, and bindings that can fail

There is a check in the opposite direction as well. An arm that is **already covered and can never be chosen** is a warning. Here is `unreachable.lune`:

```
def f(c: Color): Int =  
  match c:  
    | Red -> 1  
    | Red -> 2  
    | Green -> 3
```

```
$ lune --check unreachable.lune
```

```
warning[TYP0009]: unreachable match case: Red  
--> unreachable.lune:10:9  
  |  
10 |           | Red -> 2  
   |           ^ this case can never match  
   = hint: remove this case, or move it before the cases that cover it  
   = help: run `lune explain TYP0009` for a detailed explanation  
type check OK
```

It is a `warning`, so the check itself passes (note the `type check OK` at the end). Chapter 12 covers how to make CI fail on warnings too.

One more. Patterns can appear in a `let` as well (the destructuring tuple binding of chapter 2). But only a pattern that **cannot fail** — an irrefutable one — may go there. Here is `refutable.lune`:

```
module bad  
  
let Some(x) = Some(1)
```

```
$ lune --check refutable.lune
```

```
error[TYP0008]: refutable pattern in let binding: Some(x)
--> refutable.lune:3:5
|
3 | let Some(x) = Some(1)
|   ^^^^ this pattern can fail to match
  = hint: the pattern does not cover None
  = hint: use `match` to handle all cases of Option[Int]
  = help: run `lune explain TYP0008` for a detailed explanation
```

Even though the right-hand side is plainly `Some(1)`. A value of type `Option[Int]` might be `None`, and as long as “might” is there, a `let` cannot receive it. As the hint says, open such a value with `match`. Tuple patterns are allowed in a `let` because a tuple has no other shape.

5.5 Generic ADTs — abstracting the container with a type parameter

Defining `Option` yourself looks like this (and this is in fact the structure of the prelude’s definition):

```
type Option[T] =
  | Some(value: T)
  | None

def getOrElse[T](option: Option[T], defaultValue: T): T =
  match option:
    | Some(value) -> value
    | None -> defaultValue
```

`[T]` means “for any type `T`” — one definition serves `Option[Int]` and `Option[String]` alike. The function takes a type parameter too, `def getOrElse[T](...)`, and promises in the type that the contents and the default are of the same type. It is the same mechanism as `map : [T, U] List[T] -> (T -> U) -> List[U]` from chapter 3.

5.6 Using Option and Result

The prelude comes with `Option[T]` (a value or nothing) and `Result[T, E]` (success or failure). These four are the basic tools.

```
lune> getOrElse(Some(42), 0)
42 : Int
lune> getOrElse(None, 7)
7 : Int
lune> optionMap(Some(2), fn x -> x + 1)
Some(3) : Option[Int]
lune> unwrapOr(Err("boom"), 0)
0 : Int
lune> resultMap(Ok(20), fn x -> x * 2)
Ok(40) : Result[Int, E]
```

`optionMap` transforms what is there and leaves `None` alone; `unwrapOr` gives the contents on success and a default on failure. Instead of throwing an exception, **carry the failure as a value** — the standard functional approach to error handling, used in earnest in the chapter 13 case studies.

Let us return one from a function of our own. Integer division cannot divide by 0, so wrap the result in `Option[Double]`, a type that carries failure as a value.

```
lune> def maybeDiv(x: Int, y: Int): Option[Double] =
...     if y == 0 then None else Some(x / y)
...
ok
lune> maybeDiv(1, 2)
Some(0.5) : Option[Double]
lune> maybeDiv(1, 0)
None : Option[Double]
```

Unremarkable code, but the type checker has quietly done something good. `None` on its own does not know what type its contents would be. Typed bare, you can see the type argument left undetermined:

```
lune> None
None : Option[T]
```

The `E` left over in `resultMap(Ok(20), ...)` above is the same thing — `Ok(20)` reveals only the success side. So how did the `None` inside `maybeDiv` become an `Option[Double]`? The result annotation `Option[Double]` is handed to both branches of the `if` as the **expected type**, and settles the undetermined `T` there. Expected types flow from a `let` annotation too.

```
lune> let nothing: Option[Double] = None
ok
lune> nothing
None : Option[Double]
```

They are handed to the arms of a `match` as well. To return division by zero as a failure with a reason, use `Result`. From `maybediv.lune`:

```
# Each arm of a match gets the expected type too, fixing Err's T and Ok's E.
def safeDiv(x: Int, y: Int): Result[Double, String] =
  match y:
    | 0 -> Err("div by zero")
    | _ -> Ok(x / y)
```

```
$ lune --eval ratio maybediv.lune
Ok(4.5)
```

`Err("div by zero")` reveals only the failure side and `Ok(x / y)` only the success side, but the expected type `Result[Double, String]` fills in the rest.

Looking ahead: the same holds for nullable types — this mechanism, where an expected type is handed to the branches and they meet in the middle, works for the nullable type `T?` of chapter 7 as well.

```
# Looking ahead (chapter 7): branches meet the same way for a
nullable type.
def nullDiv(x: Int, y: Int): Double? =
  if y == 0 then null else x / y
```

The `null` branch and the `Double` branch meet at `Double?` and the function type-checks. Chapter 7 sorts out when to use `Option` and when to use `T?`.

Summary

Concept	In one line
<code>type T = A(...) B(...)</code>	a sum type; a value is always exactly one constructor
field-less constructors	call them to make a value, <code>Green()</code> (only the prelude's <code>None</code> / <code>Nil</code> are bare values)
<code>match</code>	an expression that checks a shape and deconstructs it at once
patterns	literal / name / <code>_</code> / tuple / constructor / guard <code>if</code>
<code>TYP0007</code>	a missing case, reported with a counterexample
<code>TYP0009</code>	an unreachable arm (a warning)
<code>TYP0008</code>	a pattern that can fail may not go in a <code>let</code>
<code>type Option[T] = ...</code>	a generic ADT; the function side takes <code>def f[T](...)</code>
<code>Option</code> / <code>Result</code>	from the prelude: <code>getOrElse</code> / <code>optionMap</code> / <code>unwrapOr</code> / <code>resultMap</code>
expected types	a result or <code>let</code> annotation reaches the branches and fixes the type arguments of <code>None</code> / <code>Err(...)</code>

Exercises

Exercise 5-1 (★) Add `Yellow` to the `Color` of the text (leaving `label` as it is). Predict the diagnostic, run `--check`, and then consider how it would differ if exhaustiveness had been satisfied with `| _ -> ...`.

Answer

`label` reports `error[TYP0007]: non-exhaustive match: missing case Yellow`. The moment a case is added to the type, the compiler finds **every place that matches on that type** — this is why “enumerate rather than `_` when you can enumerate”. Had the code said `| _ -> "unknown"`, `Yellow` would quietly become `"unknown"` and nobody would notice.

Exercise 5-2 (★★) Write a traffic light colour `Light` (Red / Yellow / Green) and a `next` that returns the next colour, in the order red → green → yellow → red.

Answer

```
module answers

# Exercise 5-2: the next colour of a traffic light. Exhaustiveness
# checking
# guarantees that every colour has been considered.
type Light =
  | Red
  | Yellow
  | Green

def next(light: Light): Light =
  match light:
    | Red -> Green()
    | Green -> Yellow()
    | Yellow -> Red()

let afterRed = next(Red())

let afterTwo = next(next(Red()))
```

```
$ lune --eval afterTwo ex5-2.lune
Yellow
```

Bare `Red` in a pattern, `Red()` when building a value. Deleting one arm to produce `TYP0007` makes a good revision exercise.

Exercise 5-3 (★★) Build something that checks an integer and returns the value when it is 0 or above, and a reason when it is negative. The prelude's `Result[Int, String]` would do, but define a result type of your own (without type parameters) here. This is practice at feeling the ADT's advantage: constructor names can be the words of your own domain.

Answer

```
module answers

# Exercise 5-3: put the result of a check into the type, information and
all,
# for both success and failure.
type Checked =
  | Valid(value: Int)
  | Invalid(reason: String)

def check(n: Int): Checked =
  if n >= 0 then Valid(n) else Invalid("negative")

def report(c: Checked): String =
  match c:
    | Valid(v) -> "ok: " + show(v)
    | Invalid(reason) -> "rejected: " + reason

let good = report(check(5))

let bad = report(check(-3))
```

```
$ lune --eval bad ex5-3.lune
"rejected: negative"
```

When there is information worth carrying on both success and failure, express it in a type rather than with a `Bool` or a special value such as `-1`. It is the most practical pattern in ADT design. With the mechanism of §5.6 the `Result[Int, String]` version works just as well, but with a type of your own the **words of your domain**, `Valid` and `Invalid`, appear in the arms of the `match` and in the diagnostics.

Exercise 5-4 (★★★) Without using the prelude, write your own `MyOption[T]` and `myGetOrElse`, and check that both `MySome(42)` and `MyNone()` work.

Answer

```
module answers

# Exercise 5-4: build a generic container by hand – the same structure
# as the
# prelude's Option.
type MyOption[T] =
  | MySome(value: T)
  | MyNone

def myGetOrElse[T](option: MyOption[T], defaultValue: T): T =
  match option:
    | MySome(value) -> value
    | MyNone -> defaultValue

let some = myGetOrElse(MySome(42), 0)

let none = myGetOrElse(MyNone(), 0)

# Fixing type arguments from the expected type (§5.6) works for a
# generic ADT
# of your own too.
let empty: MyOption[Int] = MyNone()
```

```
$ lune --eval some ex5-4.lune
42
$ lune --eval none ex5-4.lune
0
```

Two ways of settling a type argument are on display. The `T = Int` of `MySome(42)` comes **from the argument**, while the `T` of `MyNone()` is settled by unification with the other argument, `0`. And since the expected-type mechanism of §5.6 applies to your own generic ADTs, an annotation can settle it too, as in `let empty: MyOption[Int] = MyNone()`.

Exercise 5-5 (★, in reverse) Write the smallest code that produces `TYP00008` (a refutable pattern), and explain what each of the diagnostic’s two hints is telling you.

Answer

`let Some(x) = Some(1)` (§5.4). The first hint, “the pattern does not cover None”, says **why** it can fail, naming the shape left uncovered. The second, “use `match` to handle all cases”, says **what to do instead**. Reading a diagnostic as cause → remedy is a rehearsal for chapter 11.

More precisely — the syntax of ADTs and `match` is in `documents/LANGUAGE_SPEC.md` §10–11; the algorithm for exhaustiveness checking and witnesses is in `documents/MATCH_EXHAUSTIVENESS_SPEC.md`; the full API of `Option` and `Result` is in `documents/STANDARD_LIBRARY_SPEC.md` §4–5; the rules by which an expected type fixes a constructor’s type arguments are in `documents/LOCAL_TYPE_INFERENCE_SPEC.md` §5.3 and §5.6. Note that the specification describes OR patterns (`| 0 | 1 ->`), which v0.1 does not implement. The code examples of this chapter live in `books/examples-en/ch05/` and are all verified against the real CLI.

Chapter 6: Records

A tuple (chapter 2) bundles values, but only the person who wrote `(36, "Ada")` knows whether the 36 is an age or a squad number. When you want to bundle values **with names**, you want a **record**. Where the ADT of chapter 5 expresses “exactly one of these” (a sum), a record expresses “all of these at once” (a product), with names.

6.1 Declaring and building — building by name

```
record User:
  name: String
  age: Int
```

Just list the field names and their types. Building one **always names the fields**.

```
lune> let ada = User(name = "Ada", age = 36)
ok
lune> ada
{ name = "Ada", age = 36 } : User
```

The display is `{ field = value }` too. Passing by position is not allowed.

```
lune> User("Ada", 36)
error[REC0006]: User requires named record fields
--> <repl:4>:1:6
|
1 | User("Ada", 36)
|      ^^^^^ use field = value
= help: run `lune explain REC0006` for a detailed explanation
```

This is the opposite of an ADT constructor, which takes its fields by position (chapter 5). Records tend to grow fields, and as long as the types match, neither machine nor human can tell what the 36 in `User("Ada", 36)` is — hence the design decision to insist on names. In return, every mistake around building is reported by name: `REC0003` when a field is missing, `REC0005` when the name is not declared, `REC0004` when the same name is given twice.

6.2 Field access and laziness

Read a field with `. .`.

```
lune> ada.name
"Ada" : String
lune> ada.age + 1
37 : Int
```

did-you-mean covers typos here too. From `typofield.lune`:

```
$ lune --check typofield.lune
```

```
error[REC0002]: unknown record field: User.nmae
--> typofield.lune:9:12
|
9 | let oops = ada.nmae
|           ^^^ field is not declared by this record
  = hint: did you mean `name`?
  = help: run `lune explain REC0002` for a detailed explanation
```

And after chapter 4 you can guess the rest — **fields are lazy too**.

```
lune> let ghost = User(name = "Ghost", age = crash())
ok
lune> ghost.name
"Ghost" : String
lune> :thunks ghost
ghost : evaluated = { name = "Ghost", age = <thunk> }
```

`name` was read quite happily with a landmine sitting in `age`. In the `:thunks` preview, the untouched field is still showing as `<thunk>`.

Put `strict` on a field you want evaluated at construction.

```
lune> record Tagged:
...   strict tag: Int
...   note: String
...
ok
lune> let t = Tagged(tag = crash(), note = "n")
ok
lune> t.note
error[RUN0006]: crash() was evaluated
  = help: run `lune explain RUN0006` for a detailed explanation
```

Reading `note` set off the mine in `tag`. The record is constructed the moment `t` is first forced, and that is when strict fields are evaluated. “Never let a record holding an invalid value exist” — the design tool promised in chapter 4.

6.3 Generic records

Records take type parameters, written as for the ADTs of chapter 5.

```
lune> record Box[T]:  
...   value: T  
...  
ok  
lune> Box(value = 42)  
{ value = 42 } : Box[Int]
```

`T = Int` was inferred from the field’s value. Functions can take type parameters too (exercise 6-3 builds a `swap` whose types swap with it).

6.4 Choosing between tuples, ADTs and records

Three tools for grouping values. To put them side by side:

Tool	Suits	Example
tuple	two or three values, here and now, not worth naming	the result of a <code>zip</code> , <code>(quotient, remainder)</code>
record	the same shape used repeatedly; the fields need names	<code>User</code> , configuration, a computed summary
ADT	there are alternatives, exactly one of which holds	<code>Shape</code> , <code>Option</code> , a state

Combining them is the norm. Here is a list of records processed with the tools we have had since chapter 1, in `items.lune`:

```

module items

record Item:
  name: String
  price: Int

let items = [Item(name = "pen", price = 120), Item(name = "note", price = 200)]

let total = fold(map(items, fn i: Item -> i.price), 0, fn a x -> a + x)

```

```

$ lune --eval total items.lune
320

```

“From a list of records, pull out a field with `map` and collapse it with `fold`” — the backbone of a working Lune program.

6.5 What is not there yet — update and patterns

v0.1’s records are missing two features that neighbouring languages have.

There is no **record update** (a “copy with one part changed”, as in `{ ada | age = 37 }`), so a new value is built by writing every field. There is no **record pattern** either, so a record cannot be deconstructed in a `match`.

```

lune> match ada:
...   | { name = n } -> n
...
error[PRS0001]: expected pattern, got LBRACE
--> <repl:12>:2:7
  |
2 |   | { name = n } -> n
  |   ^ unexpected token
  = help: run `lune explain PRS0001` for a detailed explanation

```

Read what you need out of a record with `.`, and branch with `if / match` on the field’s value. Both features are in the future specification (appendix E).

Break it — produce the record-building diagnostics `REC0003` (missing field), `REC0004` (the same field twice) and `REC0005` (unknown field) one at a time with `User`. Check that each is reported by name, on the construction expression. Of the code families in chapter 11, `REC` has its home ground in this chapter.

Summary

Concept	In one line
<code>record R:</code> plus field declarations	a named product; building requires <code>R(field = value)</code>
<code>r.field</code>	reading a field; a typo gives <code>REC0002</code> plus did-you-mean
laziness of fields	lazy by default; <code>strict field: T</code> evaluates at construction
<code>record Box[T]:</code>	a generic record
choosing	ad-hoc group → tuple / named product → record / one of several → ADT
not supported	record update, record patterns (read with <code>.</code>)

Exercises

Exercise 6-1 (★) Break the construction of `User` in three ways to produce `REC0003`, `REC0005` and `REC0006` (predict first which break gives which code).

Answer

`User(name = "X")` → `REC0003` (`age` is missing), `User(name = "X", years = 1)` → `REC0005` (`years` is not a declared name), `User("X", 36)` → `REC0006` (built without names). While you are there, `User(name = "X", name = "Y", age = 1)` gives `REC0004`.

Exercise 6-2 (★★) From the items of `items.lune`, build a list of **just the names** of those costing 150 or more.

Answer

```
module answers

# Exercise 6-2: take just the names of the items costing 150 or more.
record Item:
  name: String
  price: Int

let items = [Item(name = "pen", price = 120), Item(name = "note", price = 200)]

let pricey = map(filter(items, fn i: Item -> i.price >= 150), fn i: Item -> i.name)
```

```
$ lune --eval pricey ex6-2.lune
("note")
```

Narrow with `filter`, then pull out with `map`. The other order (names first) throws the price away and leaves nothing to filter on — the order of a pipeline is decided by how long each piece of information has to live.

Exercise 6-3 (★★) Write a generic record `Pair[A, B]` (`first: A`, `second: B`) and a `swap` that exchanges the two. Note that the result type of `swap` is `Pair[B, A]`.

Answer

```
module answers

# Exercise 6-3: a generic record, and a swap whose types swap with it.
record Pair[A, B]:
  first: A
  second: B

def swap[A, B](p: Pair[A, B]): Pair[B, A] =
  Pair(first = p.second, second = p.first)

let swapped = swap(Pair(first = 1, second = "a"))
```

```
$ lune --eval swapped ex6-3.lune
{ first = "a", second = 1 }
```

The type arguments of `Pair(first = p.second, ...)` are inferred from the values passed, so nothing needs writing on the construction side.

Exercise 6-4 (★★★) Build both §6.2's `Tagged` (with a strict `tag`) and an identical record without `strict`, put `crash()` in `tag` for each, and explain the difference in

behaviour, using `:thunks` as well.

Answer

Without `strict`, `l.note` returns `"n"` and `:thunks l` shows `{ tag = <thunk>, note = ... }` — the mine is harmless until touched. With `strict`, `t.note` already gives `RUN0006`, because forcing `t` runs the record's construction and construction evaluates the strict fields. The chapter 4 principle, "lazy by default plus explicit strictness", appears here directly as a data design tool.

More precisely — the syntax, typing and laziness rules of records are in `documents/RECORD_FIELD_SPEC.md` (which states in §7 and §6 that update and patterns are future work). The code examples of this chapter live in `books/examples-en/ch06/` and are all verified against the real CLI.

Chapter 7: Null Safety

`null`, standing for the absence of a value, is the king of runtime errors in many languages. Forget a single null check and one day the program simply falls over — a device its own inventor came to call his billion-dollar mistake.

Rather than abolish `null`, Lune chose to **tame it with types**. Places where null may appear are declared with the type `T?`, and the compiler will not let you use such a value before checking it. The role is close to `Option` from chapter 5, but this is the lightweight version, with syntax of its own (`??`, `?.`, narrowing). The end of the chapter sorts out which to use when.

7.1 T? — a type that can hold null

Adding `?` to a type lets it hold a value of that type **or** `null`.

```
lune> let present: Int? = 42
ok
lune> let absent: Int? = null
ok
lune> present
42 : Nullable[Int]
```

The `Nullable[Int]` in the display is the internal name of `Int?` (read it as such). An `Int` value goes where an `Int?` is wanted, but **not the other way round**.

```
lune> def wantsInt(n: Int): Int =
...     n + 1
...
ok
lune> wantsInt(present)
error[TYP0003]: expected Int, got Nullable[Int]
= help: run `lune explain TYP0003` for a detailed explanation
```

That is the whole of null safety. A value that might be null cannot be used as an ordinary value **until you have checked**. The rest of this chapter is the ways of checking.

7.2 Removing it with match — covering null, and narrowing

The most basic way is `match`. Here is `orzero.lune`:

```
module orzero

# A T? holds either a T or null.
let present: Int? = 42

let absent: Int? = null

# Covering null in a match narrows v to Int in the remaining arm.
def orZero(value: Int?): Int =
  match value:
    | null -> 0
    | v -> v
```

```
$ lune --eval unwrapped orzero.lune
42
$ lune --eval defaulted orzero.lune
0
```

Because the `| null ->` arm has taken null **first**, the `v` of the next arm is **narrowed** to `Int`. That is why `v` can be returned as an `Int` unchanged.

Exhaustiveness checking (chapter 5) counts null as a case. Here is `missingnull.lune`:

```
module bad

# A match on Bool? needs more than true and false.
def toInt(b: Bool?): Int =
  match b:
    | true -> 1
    | false -> 0
```

```
$ lune --check missingnull.lune
```

```
error[TYP0007]: non-exhaustive match: missing case null
--> missingnull.lune:5:5
|
5 |     match b:
|     ^^^^^ pattern null is not covered
  = hint: add a case for null, or a wildcard case `| _ -> ...`
  = help: run `lune explain TYP0007` for a detailed explanation
```

The classic accident of forgetting a null check has turned into `missing case null` at compile time.

One caution. Catching everything with a name pattern satisfies exhaustiveness but does **not** narrow.

```
lune> def bad(value: Int?): Int =  
...     match value:  
...         | v -> v  
...  
error[TYP0003]: branch: expected Int, got Nullable[Int]  
--> <repl:6>:3:16  
|  
3 |         | v -> v  
|         ^ this expression has type Nullable[Int]  
= help: run `lune explain TYP0003` for a detailed explanation
```

`v` becomes an `Int` only when the `null` arm comes before it.

7.3 ?? — this one, if there is nothing

“Use a default when it is null” is common enough to have its own operator: the **null-coalescing operator** `??`.

```
lune> absent ?? 7  
7 : Int  
lune> present ?? 7  
42 : Int  
lune> present ?? crash()  
42 : Int
```

It returns the left side unless it is null, and the right side when it is. The result type is `Int`, not `Int?` — after coalescing, the possibility of null is gone. Note the third example: when there is a value on the left, **the right side is not even evaluated**. Chapter 4’s lazy evaluation, doing its ordinary work here too.

7.4 ?. — following a reference safely

“Read `user.name`, but only if `user` is not null” also has syntax of its own: **safe navigation**, `?.`. Here is `nameof.lune`:

```

module nameOf

record User:
  name: String
  age: Int

# ?. short-circuits to null when the receiver is null, and reads the field
# otherwise.
def nameOf(user: User?): String? =
  user?.name

let someName = nameOf(User(name = "Ada", age = 36))

let noName = nameOf(null)

let fallback = nameOf(null) ?? "(nobody)"

```

```

$ lune --eval someName nameOf.lune
"Ada"
$ lune --eval noName nameOf.lune
null
$ lune --eval fallback nameOf.lune
"(nobody)"

```

Null receiver, null result; otherwise the field's value. The result is always a `String?`, so catch it at the end with `??` or a `match`. Carry null along with `?.` and land it with `??` — the two are used as a pair.

7.5 Narrowing with if

Where a `match` would be more than you need, an `if` condition narrows as well.

```

lune> def orOne(x: Int?): Int =
...   if x != null then x else 1
...
ok
lune> orOne(absent)
1 : Int

```

In the branch where `x != null` is true, `x` can be used as an `Int` (with `x == null`, the false branch narrows instead). This narrowing works only for the simple forms `x != null` and `x == null`; it does not extend to compound conditions joined with `&&`, or to `elif`. When the condition gets involved, switch to `match`.

7.6 Writing a function that returns null

So far we have been on the receiving end of null. On the **returning** end, the straightforward code just works.

```
lune> def maybeDiv(x: Int, y: Int): Double? =  
...     if y == 0 then null else x / y  
...  
ok  
lune> maybeDiv(7, 2)  
3.5 : Nullable[Double]  
lune> maybeDiv(7, 0)  
null : Nullable[Double]
```

Why do the arm returning `null` and the arm returning `Double` meet without complaint? This is the mechanism promised in the look-ahead box of §5.6: the result annotation `Double?` is handed to both branches of the `if` as the **expected type**, and both the `null` arm and the `x / y` arm line up at `Double?` there. The same happens for the arms of a `match`. Here it is as a file, `maybediv.lune`:

```
module maybediv  
  
# The expected type Double? reaches both branches of the if, and the null  
# arm  
# and the x / y arm meet there.  
def maybeDiv(x: Int, y: Int): Double? =  
    if y == 0 then null else x / y  
  
let some = maybeDiv(7, 2)  
  
let none = maybeDiv(7, 0)  
  
let fallback = maybeDiv(7, 0) ?? 0.0
```

```
$ lune --eval some maybediv.lune  
3.5  
$ lune --eval none maybediv.lune  
null  
$ lune --eval fallback maybediv.lune  
0.0
```

`maybeDiv(7, 0)` raises no division-by-zero error because an `if` does not evaluate the arm it did not choose (chapter 4). Leave the computation that might fail in the else arm and let null carry the other case — the basic shape of a function returning `T?`.

This works too — you can also build the parts in an annotated `let` before branching.

```
# Building the parts in an annotated let before branching works too.
# quotient is a thunk, so on the y == 0 side the division never runs.
def maybeDivLet(x: Int, y: Int): Double? =
  let quotient: Double? = x / y
  if y == 0 then null else quotient
```

`maybeDivLet(7, 0)` returns `null` as it should. The reason it “passes through” `let quotient: Double? = x / y` without a division-by-zero error is that `quotient` stays a thunk and is never forced — chapter 4 again. Both forms are safe, so the straightforward one, branching first, is normally enough.

Reading a field that may not be there (`?.`), landing on a default (`??`), and this “return null when there is nothing”. With those three, null becomes an ordinary tool, guarded by the type system.

7.7 Choosing between `Option[T]` and `T?`

Why there are two similar tools, and how to choose.

	<code>T?</code>	<code>Option[T]</code>
how “nothing” is written	<code>null</code>	<code>None</code>
dedicated syntax	<code>??</code> , <code>?.</code> , narrowing in <code>if/match</code>	none (handled with <code>match</code> and functions)
fits	optional fields, parameters and results	the prelude’s list API (<code>head / tail</code> return it), generic functions
nested “nothing”	inexpressible (<code>null</code> is one level)	<code>Some(None)</code> and <code>None</code> are distinct

The practical rule: **use `T?` when “possibly absent” is part of the shape of your data** (record fields, parameters, results). **Use `Option` inside list processing and higher-order pipelines**, because that is the language the prelude speaks. At the boundary, a `match` converts between them.

Break it — what happens if you swap the two arms of `orZero`, putting `| v -> v` first? Predict, then try.

```
lune> def swapped(value: Int?): Int =  
...   match value:  
...     | v -> v  
...     | null -> 0  
...  
error[TYP0003]: branch: expected Int, got Nullable[Int]  
--> <repl:1>:3:16  
|  
3 |           | v -> v  
|           ^ this expression has type Nullable[Int]  
= help: run `lune explain TYP0003` for a detailed explanation
```

Most people predict unreachability (`TYP0009`), but it is caught before that. The leading `v` is still an `Int?`, because `null` has not been covered yet (the caution in §7.2). The result annotation `Int` reaches it as an expected type (§7.6), and the first arm is named as unable to meet it. Narrowing works only when the `null` arm comes **first** — the order of the arms reaches all the way into the types.

Summary

Concept	In one line
<code>T?</code>	a <code>T</code> or <code>null</code> ; displayed as <code>Nullable[T]</code>
what the type guards	using a <code>T?</code> as a <code>T</code> gives <code>TYP0003</code> ; a missing null case gives <code>TYP0007</code>
<code>match</code>	once <code> null -></code> is covered, later name patterns narrow to <code>T</code>
<code>??</code>	a default when null; the right side is not evaluated until needed
<code>?.</code>	short-circuits to null, otherwise reads the field; pairs with <code>??</code>
<code>if x != null</code>	narrows for the simple form only; use <code>match</code> when it gets complicated
returning null	expected types reach the branches, so plain <code>if / match</code> works (the mechanism of §5.6)
VS <code>Option</code>	<code>T?</code> for the shape of data, <code>Option</code> for lists and higher-order functions

Exercises

Exercise 7-1 (★) Predict the result (value and type), then check. Take `absent: Int? = null` and `present: Int? = 42`.

```
absent ?? 7
present ?? 7
absent == null
present ?? crash()
```

Answer

`7 : Int, 42 : Int, true : Bool, 42 : Int`. The last does not explode because `??` short-circuits. Check also that the type is `Int` and not `Int?`.

Exercise 7-2 (★★) Rewrite §7.5's `orOne` with `match` instead of `if`.

Answer

```
lune> def orOne(x: Int?): Int =  
...     match x:  
...       | null -> 1  
...       | v  -> v  
...  
ok  
lune> orOne(null)  
1 : Int  
lune> orOne(41)  
41 : Int
```

The same skeleton as `orZero`. Both the `if x != null` version and the `match` version are correct; the rule of thumb is `if` for two branches and `match` once there are more.

Exercise 7-3 (★★) Following `nameOf`, write `ageOf`, and check that `ageOf(null) ?? 0` gives `0`.

Answer

```
lune> def ageOf(user: User?): Int? =  
...     user?.age  
...  
ok  
lune> ageOf(User(name = "Ada", age = 36))  
36 : Nullable[Int]  
lune> ageOf(null) ?? 0  
0 : Int
```

Follow the flow of types for “carry with `?.`, land with `??`” — `User? → Int? → Int` — with `:type` as well.

Exercise 7-4 (★★★) Write `average(xs: List[Int]): Double?`, the average of a list of integers, with the average of the empty list being `null`. Can you write it without ever dividing by zero?

Answer

```
module answers

# Exercise 7-4: the average of an empty list does not exist. Branching
# first
# means never touching the division by zero.
def average(xs: List[Int]): Double? =
  if isEmpty(xs) then null else fold(xs, 0, fn a x -> a + x) /
length(xs)

let some = average([2, 3, 4])

let none = average([])

let safe = average([]) ?? 0.0
```

```
$ lune --eval some ex7-4.lune
3.0
$ lune --eval none ex7-4.lune
null
$ lune --eval safe ex7-4.lune
0.0
```

Branch on emptiness **first** and the division is only ever evaluated for a non-empty list. The `null` arm and the computing arm meet at `Double?` as in §7.6. Defining the sum and the division in annotated `let` s before branching also works, since they are discarded while still thinks (the “this works too” box in §7.6).

Exercise 7-5 (★, in reverse) Write the smallest code that gets you told off by the type checker for using a possibly-null value without checking it.

Answer

```
let present: Int? = 42

let oops = present + 1
```

gives `error[TYP0003]: +: expected numeric type, got Nullable[Int]`. Passing it to a function (`wantsInt(present)` in §7.1) gives `expected Int, got Nullable[Int]`. The accident that would have been a runtime `NullPointerException` in another language has been replaced by this one line at compile time — that is what null safety is worth.

More precisely — the exact typing and narrowing rules for `T?` are in `documents/LANGUAGE_SPEC.md` §11 (null patterns in `match`) and §9.2 (narrowing in

`if`); exhaustiveness with null is in `documents/MATCH_EXHAUSTIVENESS_SPEC.md`; the rule by which an expected type reaches the branches is in `documents/LOCAL_TYPE_INFERENCE_SPEC.md` §5.3. There are further examples in `samples/nullable.lune`. The code examples of this chapter live in `books/examples-en/ch07/` and are all verified against the real CLI.

Chapter 8: Lists and Streams

This is the chapter where the lazy evaluation of chapter 4 turns into data structure design. The first half relearns properly the lists we have used since chapter 1. In the second half those same lists become **infinite** — with no new type involved. A Lune list is born able to be infinite.

8.1 What a list really is — Cons and Nil

`[1, 2, 3]` is sugar. A list is really an ordinary ADT defined in the prelude: two constructors, writable with the tools of chapter 5.

```
lune> :type Cons
Cons : [T] T -> List[T] -> List[T]
lune> :type Nil
Nil : List[T]
lune> Cons(1, Cons(2, Nil))
(1 2) : List[Int]
```

`Nil` is the empty list and `Cons(head, rest)` is “a first value, and the rest of the list”. `[1, 2]` means `Cons(1, Cons(2, Nil))` (`Nil`, like the prelude’s `None`, is registered as a value).

Now recall chapter 4: **constructor fields are lazy by default**. So the “rest” of a `Cons` is a thunk, not computed until needed. That single fact carries the whole second half of this chapter.

8.2 The basic tools — opening, counting, transforming

First, opening. `head` and `tail` return an `Option`.

```
lune> head([1, 2, 3])
Some(1) : Option[Int]
lune> head([])
None : Option[T]
lune> tail([1, 2, 3])
Some((2 3)) : Option[List[Int]]
lune> isEmpty([])
true : Bool
```

An empty list has no first element. Returning a “might not be there” answer as an `Option` is exactly the design of chapter 5. That said, the best way to open a list is not to peel an `Option` but to **destructure it directly with `match`**, as §8.6 does.

The transforming tools are the familiar `map` / `filter` / `fold`, plus `take` / `drop` / `range` / `length`. Here are their types by way of revision — note that all of them take the list first and the tool second.

Function	Type (in outline)	In a phrase
<code>map(xs, f)</code>	<code>List[T] -> (T -> U) -> List[U]</code>	<code>f</code> over every element
<code>filter(xs, p)</code>	<code>List[T] -> (T -> Bool) -> List[T]</code>	only the elements where <code>p</code> holds
<code>fold(xs, init, f)</code>	<code>List[T] -> U -> ((U, T) -> U) -> U</code>	collapse it
<code>take(xs, n) / drop(xs, n)</code>		take / discard the first <code>n</code>
<code>head / tail</code>		return an <code>Option</code>
<code>length / isEmpty / range</code>		length / emptiness / a range of integers

8.3 The tail is lazy — a list can be infinite

To the point. The tail of a `Cons` is a thunk. Which means the computation of “the rest of the list” **is allowed never to finish**. The prelude has functions that build exactly such lists.

```
naturalsFrom(n)    # [n, n+1, n+2, ...]
iterate(f, x)      # [x, f(x), f(f(x)), ...]
repeat(x)          # [x, x, x, ...]
cycle(xs)           # xs, repeated forever
```

Let us look inside an infinite list with `:thunks`, which is safe here because it causes no evaluation.

```

lune> let nat = naturalsFrom(1)
ok
lune> :thunks nat
nat : unevaluated
lune> head(nat)
Some(1) : Option[Int]
lune> :thunks nat
nat : evaluated = Cons(1, <thunk>)
lune> take(nat, 5)
(1 2 3 4 5) : List[Int]
lune> :thunks nat
nat : evaluated = Cons(1, Cons(2, Cons(3, Cons(...))))

```

`Cons(1, <thunk>)` — only the leading 1 has been computed and all the rest is still a promise. Asking for five with `take` advanced the evaluation by exactly that much. **An infinite list is a sequence that materialises only as far as it is needed**, and `:thunks` shows you how far that is. The instruments of chapter 4 have their finest hour here.

Combined with `take`, an infinite list is handled like any other value. Here is `infinite.lune`:

```

module infinite

# The tail of a List is lazy. That is how a list can be infinite.
let firstFive = take(naturalsFrom(1), 5)

let powersOfTwo = take(iterate(fn x: Int -> x * 2, 1), 6)

let threeSevens = take(repeat(7), 3)

let pattern = take(cycle([1, 2, 3]), 7)

```

```

$ lune --eval powersOfTwo infinite.lune
(1 2 4 8 16 32)
$ lune --eval pattern infinite.lune
(1 2 3 1 2 3 1)

```

8.4 Lazy combinators — transforming while still infinite

`map` and `filter` preserve the laziness of the tail, so an **infinite list can be transformed and stay infinite**. Harvest as much as you need with `take` at the end.

```
lune> take(map(naturalsFrom(1), fn n: Int -> n * n), 5)
(1 4 9 16 25) : List[Int]
lune> takeWhile(naturalsFrom(1), fn x -> x < 4)
(1 2 3) : List[Int]
lune> take(dropWhile(naturalsFrom(1), fn x -> x < 10), 3)
(10 11 12) : List[Int]
lune> take(zip(naturalsFrom(1), cycle(["a", "b"])) , 4)
((1, "a") (2, "b") (3, "a") (4, "b")) : List[Tuple[Int, String]]
lune> take(zipWith(naturalsFrom(1), naturalsFrom(10), fn a: Int b: Int ->
a + b), 3)
(11 13 15) : List[Int]
```

- `takeWhile` / `dropWhile` — take or discard while the condition holds
- `zip(a, b)` — pair two lists into a list of tuples (stopping with the shorter one)
- `zipWith(a, b, f)` — pair them and combine with `f`

Zippping two infinite lists is fine as well. The result is another infinite list, and only what you use gets computed.

8.5 Operations that do not stop — the consuming functions

Some tools, however, **consume the list to the end**: `fold` and `length`. Used on an infinite list, the end never arrives, so they **do not stop**.

```
lune> length(naturalsFrom(1))
```

The prompt does not come back (interrupt with Ctrl-C). Note that it is not an error — counting one by one forever is a perfectly correct computation that happens never to finish. The same goes for displaying an infinite list in full: `take` some of it before printing.

Telling them apart is simple: **does producing the answer require every element?** `take`, `map`, `filter`, `takeWhile`, `zip` do not (they preserve laziness). `fold` and `length` do. `drop(xs, n)` forces only the `n` cells it discards, so it is safe on an infinite list — `take(drop(naturalsFrom(1), 5), 3)` gives `(6 7 8)` — but the list it hands back is still infinite, so keep a `take` in front of the display. `filter` has a trap of its own: if no element ever satisfies the condition again, it keeps running while looking for the next one.

The same test applies to **finite** lists. `range` is lazy too (§4.6), so while `range(1, 1000000000)` is not infinite, handing it to `length` or `fold` costs a wait not meaningfully

different from one that never ends. Whether to put a `take` or `takeWhile` in front is decided by **whether the answer needs every element**, not by whether the list is infinite.

8.6 Worked examples — thinking in infinite lists

The Fibonacci sequence. Read it as “keep advancing the state `(a, b)` to `(b, a+b)`”, which is what `iterate` is for. Here is `fib.lune`:

```
module fib

# Advance the state (a, b) by one step: (0,1) -> (1,1) -> (1,2) -> (2,3) -
> ...
def step(p: Tuple[Int, Int]): Tuple[Int, Int] =
  match p:
    | (a, b) -> (b, a + b)

def fst(p: Tuple[Int, Int]): Int =
  let (a, _) = p
  a

# The Fibonacci sequence itself, as an infinite list.
let fibs = map(iterate(step, (0, 1)), fst)

let first10 = take(fibs, 10)
```

```
$ lune --eval first10 fib.lune
(0 1 1 2 3 5 8 13 21 34)
```

`fibs` is not a name for “ten Fibonacci numbers” but for **the Fibonacci sequence itself**. How many to use is decided later, by whoever uses it. Separating production from consumption is the real design benefit of infinite lists.

The sieve of Eratosthenes. An infinite list of primes. Here is `primes.lune`:

```

module primes

# The sieve of Eratosthenes. Take the head p as a prime, strain the
# multiples
# of p out of the rest, and leave the continuation to be computed on
# demand.
def sieve(xs: List[Int]): List[Int] =
  match xs:
    | Cons(p, rest) -> Cons(p, sieve(filter(rest, fn n: Int -> n % p
    != 0)))
    | Nil -> Nil

let primes = sieve(naturalsFrom(2))

let first10 = take(primes, 10)

```

```

$ lune --eval first10 primes.lune
(2 3 5 7 11 13 17 19 23 29)

```

Three lines with a lot packed into them.

- The list is destructured straight into `Cons(p, rest)` with `match` (more direct than `head / tail`).
- The second argument of the returned `Cons` — the recursive call `sieve(...)` — **goes into a lazy field**, so the endless recursion does not run away. The next prime is strained out only when somebody asks for it.
- The bare `Nil` on the right of `| Nil -> Nil` would have an undetermined type argument on its own (like `None : Option[T]` in §5.6). Here the result annotation `List[Int]` reaches the arm as an expected type and settles `T = Int`.

Break it — get the argument order of `take` wrong and the type tells you immediately.

```

lune> take(5, naturalsFrom(1))
error[TYP0003]: expected List[T], got Int
  = help: run `lune explain TYP0003` for a detailed explanation

```

List first, count second. Remember the convention from the table in §8.2: list functions take the list as their first argument.

Summary

Concept	In one line
<code>Cons</code> / <code>Nil</code>	what a list really is; <code>[1, 2]</code> is <code>Cons(1, Cons(2, Nil))</code>
a lazy tail	the “rest” is a thunk, which is how a list can be infinite
<code>head</code> / <code>tail</code>	return an <code>Option</code> ; to destructure, use <code>match on Cons(x, rest)</code>
<code>naturalsFrom</code> / <code>iterate</code> / <code>repeat</code> / <code>cycle</code>	four ways to build an infinite list
laziness-preserving tools	<code>take</code> / <code>map</code> / <code>filter</code> / <code>takeWhile</code> / <code>dropWhile</code> / <code>zip</code> / <code>zipWith</code>
consuming tools	<code>fold</code> / <code>length</code> — not for infinite lists
observing	<code>:thunks</code> shows how far a list has materialised, without evaluating it

Exercises

Exercise 8-1 (★) Predict the results, then check.

```
take(cycle([1, 2]), 5)
takeWhile(naturalsFrom(1), fn x -> x < 4)
take(zip(naturalsFrom(1), cycle(["a", "b"]))), 4)
```

Answer

`(1 2 1 2 1)`, `(1 2 3)`, `((1, "a") (2, "b") (3, "a") (4, "b"))`. The third zips an infinite sequence of numbers with an infinite alternating pattern — neither ends, but only four were asked for, so only four were computed.

Exercise 8-2 (★★) Build the infinite list of squares (1, 4, 9, 16, ...) and take the first five even ones.

Answer

```
module answers

# Exercise 8-2: five even numbers from the infinite list of squares.
let squares = map(naturalsFrom(1), fn n: Int -> n * n)

let evenSquares = take(filter(squares, fn s: Int -> s % 2 == 0), 5)
```

```
$ lune --eval evenSquares ex8-2.lune
(4 16 36 64 100)
```

`map` → `filter` → `take`: two transformations applied while still infinite, then the harvest.

Exercise 8-3 (★★) For the sensor readings `[1, 4, 7, 10]`, compute the average of each neighbouring pair (a moving average), `(2.5 5.5 8.5)`. Hint: shift the list by one and `zipWith` it against itself.

Answer

```
module answers

# Exercise 8-3: the average of each neighbouring pair (a moving
average).
# Shift the list by one and zip it with itself.
let readings = [1, 4, 7, 10]

let smoothed = zipWith(readings, drop(readings, 1), fn a: Int b: Int ->
(a + b) / 2)
```

```
$ lune --eval smoothed ex8-3.lune
(2.5 5.5 8.5)
```

`zipWith` stops with the shorter side (the dropped one), so there is no edge case to write.

Exercise 8-4 (★★★) Build the infinite list of Lucas numbers (2, 1, 3, 4, 7, ... — the same recurrence as Fibonacci, starting from `(2, 1)`) and take the first ten.

Answer

```
module answers

# Exercise 8-4: the Lucas numbers. The same recurrence as Fibonacci,
# with the
# seed changed to (2, 1).
def step(p: Tuple[Int, Int]): Tuple[Int, Int] =
  match p:
    | (a, b) -> (b, a + b)

def fst(p: Tuple[Int, Int]): Int =
  let (a, _) = p
  a

let lucas = map(iterate(step, (2, 1)), fst)

let first10 = take(lucas, 10)
```

```
$ lune --eval first10 ex8-4.lune
(2 1 3 4 7 11 18 29 47 76)
```

The only difference from `fib.lune` is the seed of `iterate`. Once you see the correspondence “recurrence = the `step` function, sequence = `iterate`”, the shape works for any recurrence.

Exercise 8-5 (★, in reverse) Write an expression that does not stop. You need not run it (if you do, have Ctrl-C ready). Explain why it does not stop, in the words of §8.5.

Answer

For example `length(naturalsFrom(1))`, `fold(repeat(1), 0, fn a x -> a + x)`, or `takeWhile(naturalsFrom(1), fn x -> x > 0)` (the condition is true forever). Each needs every element to produce its answer. A slightly meaner example is `head(filter(naturalsFrom(1), fn x -> x < 0))` — `filter` itself preserves laziness, but the first element is never found.

More precisely — the lazy field of `Cons` and the laziness of each function (how far it forces) are documented per function in `documents/STANDARD_LIBRARY_SPEC.md` §6. Even the fact that `take(list, 0)` does not evaluate `list` is specified. The code examples of this chapter live in `books/examples-en/ch08/` and are all verified against the real CLI.

Chapter 9: Writing Imperatively — var, while, for and IO

Did you notice? In eight chapters you have **never once modified a variable**. The temperature table, the infinite list of primes, the rock-paper-scissors judgement — all of it was written by naming values and combining them in expressions.

Even so, there are moments when you want to turn a counter, or update a running result. Lune's answer is pragmatic: **imperative code is a good tool as long as it is used in small doses**. This chapter covers the mutable binding `var`, the loops `while` and `for`, and `IO`, the place where output goes. The watchword is: **keep the commands inside a block and hand a value back out**.

9.1 The block idiom — an island of commands

First, look at where Lune puts imperative code. Here is `counter.lune`:

```
module counter

# var and while live inside a block; the last expression is its result.
let answer =
  var i = 0
  var total = 0
  while i < 5:
    total = total + i
    i = i + 1
  total
```

```
$ lune --eval answer counter.lune
10
```

Inside the block, variables are reassigned and a loop turns. But **from outside**, `answer` is simply an `Int`. It is the same shape as the block `let` of chapter 2, with the final expression `total` as the value of the whole block. The imperative process is sealed on its island, and outside the island it is the world of expressions as before. As long as you keep this shape, imperative code does not contaminate the rest of the program.

And as an inhabitant of chapter 4, one remark: this `let answer = ...` is lazy too. Until somebody uses `answer`, the loop does not turn even once.

9.2 var — a strict mutable binding

A binding that can be reassigned is made with `var`.

```
var count = 0
count = count + 1
```

There are two differences from `let`. It is **mutable**, and it is **strict** (the right-hand side is evaluated at the moment of binding). `tick()` from chapter 4 confirms it.

```
lune> var t = tick()
ok
lune> tickCount()
1 : Int
```

The counter advanced on the binding alone — with `let` it would still be 0. A variable whose value changes cannot be reasoned about if you also cannot say when it is evaluated, so remember that `var` is not lazy.

	<code>let</code>	<code>var</code>
reassignment	no (immutable)	yes
evaluation	lazy (not computed until used)	strict (computed at the binding)

Only a name can be assigned to. Besides `i = i + 1`, the compound assignments `+=` `-=` `*=` `/=` `//=` `%=` are available (`i += 1` means exactly `i = i + 1`). One pitfall: `/` is always true division, so `x /= 2` on an `Int` variable is a type error (the result would be a `Double`). To divide an integer and stay in integers, use floor division, `x //= 2` (`//` is in §2.2).

That “no” is a check, not an intention. Assigning to a `let` reports `TYP0013`.

```
lune> let a = 1
ok
lune> a = 2
error[TYP0013]: cannot assign to `a`: it is bound with `let` and is
immutable
  --> <repl:2>:1:1
  |
1 | a = 2
  | ^ this binding cannot be assigned to
  | = hint: declare it with `var a = ...` if it has to change
  | = help: run `lune explain TYP0013` for a detailed explanation
lune> a
1 : Int
```

Files are no different. Running `lune --check` on `letassign.lune` reports the same `TYP0013`, pointing at the assignment.

`let` is not the only immutable thing. **Function parameters**, **for variables** and names bound by a pattern cannot be assigned to either. The only binding that may change is a `var`. The check looks at the nearest binding, so a `let x` inside a block stays immutable even when an outer `var x` exists.

9.3 while — the smallest loop

`while condition:` repeats a block while the `Bool` condition holds. `counter.lune` in §9.1 is all there is to it — the condition is evaluated **every time round**, and the whole `while` expression is always `Unit`. In other words `while` is not a tool for producing a value but a tool for reassigning a `var`.

There is no `break` or `continue`. To leave early, weave the condition into the loop condition (as in `while i < 5 && not(done):`). When that starts to get tiresome, it is usually a sign that the code **wants to be recursion or a fold** (chapters 3 and 8).

9.4 for — walking a list

When all you want is to do something to every element of a list, `for` is more concise. Here is `fortotal.lune`:

```

module fortotal

let pairs = [(1, 10), (2, 20)]

# A for pattern may be any irrefutable one, as in a let.
let total =
  var result = 0
  for (left, right) in pairs:
    result = result + left + right
  result

```

```

$ lune --eval total fortotal.lune
33

```

The form is `for pattern in list:`, and the pattern follows the same rule as in a `let` (chapter 5): an **irrefutable** pattern such as a tuple can destructure, and a pattern that could fail is rejected.

```

lune> for Some(x) in [Some(1), Some(2)]:
...   println(x)
...
error[TYP0008]: refutable pattern in for binding: Some(x)
--> <repl:1>:1:5
|
1 | for Some(x) in [Some(1), Some(2)]:
|   ^^^^ this pattern can fail to match
= hint: the pattern does not cover None
= hint: use `match` to handle all cases of Option[Int]
= help: run `lune explain TYP0008` for a detailed explanation

```

Only a `List[T]` can be walked. Passing something else, as `badfor.lune` does, has a diagnostic of its own.

```

error[TYP0006]: for iterable must be List, got Int
--> badfor.lune:5:14
|
5 |   for x in 42:
|           ^^ iterable must be List[T]
= help: run `lune explain TYP0006` for a detailed explanation

```

`for` is `Unit` as well. And since it advances by forcing the spine one step at a time, a **for over an infinite list does not come back** — think of it as one of the consuming tools of chapter 8.

9.5 IO — output and laziness

We have used `println` many times, but output has a trap of its own in a lazy world. Watch.

```
lune> let p = println("hi")
ok
lune> p
hi
() : Unit
```

The `let` line printed **nothing**. `println("hi")` was wrapped in a thunk and ran only when `p` was used. In a lazy world, then, *when* output happens is decided by *when* a value is needed — an awkward property for a program that wants characters on the screen in a particular order.

So output is **collected into an IO: block**. Here is `io.lune`:

```
module io

# Collect output in an IO block, which runs top to bottom.
def report(): Unit =
  IO:
    println("one")
    println("two")

let run = report()
```

```
$ lune --eval run io.lune
one
two
()
```

The statements in an `IO:` block run **top to bottom** when the block runs. It is also a declaration that “the side effects whose order matters are here”, and a marker for the reader. (In v0.1, `IO:` is that placement convention rather than strict effect tracking in the type system; the future specification plans to strengthen it.)

As a matter of style, remember: **computation in pure functions, output inside IO**. Split the decision logic into a function returning a `String` and pour it through a `for` inside `IO:` — exercise 9-2 (FizzBuzz) is practice at that separation.

9.6 raise / throw — the last resort

There is an expression for raising a runtime error yourself.

```
lune> let bad = raise "failed"
ok
lune> bad
error[RUN0006]: failed
  = help: run `lune explain RUN0006` for a detailed explanation
```

`raise expr` (and its synonym `throw`) raises a runtime error when evaluated. As ever, `let` is lazy, so binding it does nothing and using it sets it off — think of it as your own version of `crash()` and you will be about right.

And an important caution: **there is no `try / catch`**. There is no way to catch a raised error inside the program. Express recoverable failure with the tools that **return failure as a value**: a monomorphic result type or `Result` from chapter 5, or `T?` from chapter 7. `raise` is for the genuinely exceptional case where stopping the program is the correct response.

Summary

Concept	In one line
the block idiom	seal imperative code in a <code>let name = block</code> and return a value from the last expression
<code>var</code>	mutable and strict ; only a name can be assigned to
<code>while condition:</code>	the condition is evaluated each time round; the expression is <code>Unit</code> ; no break/continue
<code>for pattern in list:</code>	<code>List[T]</code> only (<code>TYP0006</code>); the pattern must be irrefutable (<code>TYP0008</code>)
order of output	collect it in an <code>IO:</code> block so laziness cannot shuffle it
the separating habit	computation in pure functions, output inside <code>IO</code>
<code>raise / throw</code>	nothing can catch it; return recoverable failure as a value

Exercises

Exercise 9-1 (★) Compute the product of 1 to 10 (10!) with `while`.

Answer

```
module answers

# Exercise 9-1: the product of 1 to 10 (10!) with while.
let product =
  var i = 1
  var acc = 1
  while i <= 10:
    acc = acc * i
    i = i + 1
  acc
```

```
$ lune --eval product ex9-1.lune
3628800
```

Exercise 9-2 (★★) Write FizzBuzz. For 1 to 15, print `Fizz` for multiples of 3, `Buzz` for multiples of 5, `FizzBuzz` for both, and the number otherwise, one per line. Keep the decision and the printing separate.

Answer

```
module answers

# Exercise 9-2: FizzBuzz. The decision is a pure function; the printing
# is a
# for inside IO.
def fizzbuzz(n: Int): String =
  if n % 15 == 0:
    "FizzBuzz"
  elif n % 3 == 0:
    "Fizz"
  elif n % 5 == 0:
    "Buzz"
  else:
    show(n)

def main(): Unit =
  IO:
    for i in range(1, 16):
      println(fizzbuzz(i))

let run = main()
```

`fizzbuzz` is a pure function, so it can be unit-tested in the REPL (`fizzbuzz(15)` → `"FizzBuzz"`). The needs of output never mix with the decision logic — the separating habit of §9.5.

Exercise 9-3 (★★) Define one `let t = tick()` and one `var v = tick()`, and use `tickCount()` to explain the difference in when `let` and `var` evaluate.

Answer

`tickCount()` straight afterwards is 1 — only the `var` advanced it. The thunk of `let t` is still asleep and becomes 2 the first time `t` is used. It is the same timing as the `strict let` of chapter 4 (`var` = mutable + strict, `strict let` = immutable + strict).

Exercise 9-4 (★★★) Rewrite §9.1's `counter.lune` (the sum of 0 to 4) without `var` or a loop. Then argue in your own words which is easier to read.

Answer

```
lune> fold(range(0, 5), 0, fn a x -> a + x)
10 : Int
```

One line. Accumulating computations are `fold`'s home ground, and the loop version's four lines (two counters and two updates) carried a responsibility that has now vanished: not mixing up `i` and `total`. On the other hand, control that breaks out partway on a condition is sometimes more natural with `while`. A rule of thumb: **`fold` to walk data once and aggregate, `while` to turn a state machine.**

Exercise 9-5 (★, in reverse) Produce `TYP0006` and `TYP0008`, each with the smallest `for` you can write.

Answer

```
for x in 42: println(x) (TYP0006 — 42 is not a List) and
for Some(x) in [Some(1)]: println(x) (TYP0008 — Some(x) is a pattern that can fail). Both
appear in §9.4. Notice too that TYP0008's hints are worded exactly as they were for
let in chapter 5 — a for binding is a relative of a let.
```

More precisely — `var` and assignment are in `documents/LANGUAGE_SPEC.md` §7.3; `while` is in `documents/WHILE_LOOP_SPEC.md`; `for`, including the rule for forcing the spine, is in `documents/FOR_LOOP_SPEC.md`; `raise / throw` and `IO:` are in `documents/LANGUAGE_SPEC.md` §9.6–9.7. The code examples of this chapter live in `books/examples-en/ch09/` and are all verified against the real CLI.

Chapter 10: Modules

As a program grows it stops fitting in one file. This chapter covers splitting a Lune program across files — **modules**.

The module system of v0.1 is deliberately small. There is no package management, no public/private control, no namespace access. There is only “split into files and load them”. Even so, this minimal set of tools has four rules worth knowing, and getting them wrong produces diagnostics of its own (the `MOD` family, chapter 11).

10.1 The module declaration — the file’s name tag

The first line you have been copying since chapter 1 finally takes the lead.

```
module geometry
```

A Lune source file declares which module it is. And **the declaration must agree with the file’s location**: `module geometry` for `geometry.lune`, `module util.text` for `util/text.lune`. A dot corresponds to a directory separator.

```
module util.text

def shout(s: String): String =
  s + "!"
```

A module name, then, is the route to the file. When the name tag disagrees with where it sits, the importing side complains (§10.5).

One exception: the **entry file**, the one passed directly to `--eval` or `--check`, is not required to match. (That is how this book has been able to put `module hello` in all sorts of places.)

10.2 import — bringing names in

The reading side uses `import`. Here is `main.lune`:

```
module main
import geometry
import util.text

let area = circleArea(1.0)

let banner = shout("area")
```

```
$ lune --eval area main.lune
3.14159
$ lune --eval banner main.lune
"area!"
```

Writing `import geometry` brings the top-level declarations of `geometry.lune` (`def`, `let`, `type`, `record`) **straight into your own environment**. Note that the call is the unqualified `circleArea(...)`, not `geometry.circleArea(...)`.

Qualified calls are not possible in v0.1.

```
lune> let a = geometry.circleArea(1.0)
error[TYP0001]: undefined name: geometry
  = help: run `lune explain TYP0001` for a detailed explanation
```

There is no value named `geometry` — an import carries the names of the contents and nothing else. The flip side is that **two modules defining the same name will collide**. The defence in v0.1 is not to make names too short (`cartTotal` rather than `total`).

The canonical form puts the `module` declaration and the `import` lines together with no blank line between them (`lune fmt` will arrange them that way; chapter 12).

10.3 How to divide a program

How should you split it? Two ideas actually pay off with the tools of v0.1.

Keep data and its operations together. Put a record or ADT definition and the functions that work on it in the same module. The `shop/items.lune` of exercise 10-2 has this shape.

```

module shop.items

record Item:
  name: String
  price: Int

def priceOf(item: Item): Int =
  item.price

def total(items: List[Item]): Int =
  fold(map(items, priceOf), 0, fn a x -> a + x)

```

Extract general-purpose tools. Utilities for strings or numbers, called from everywhere, go somewhere like `util.text`.

There are also divisions that are **better avoided** in v0.1. “Split to hide the implementation” — without public/private control, everything stays visible. “Split apart things that reference each other” — as the next section shows, cycles are forbidden.

10.4 Cyclic imports are detected

What happens when `cycle_a` imports `cycle_b` and `cycle_b` imports `cycle_a`?

```
$ lune --check cycle_a.lune
```

```

error[MOD0002]: cyclic module import detected: cycle_a.lune ->
cycle_b.lune -> cycle_a.lune
  --> cycle_b.lune:2:1
  |
2 | import cycle_a
  | ^^^^^^^ this import closes the cycle
  = help: run `lune explain MOD0002` for a detailed explanation

```

The diagnostic shows **the path of the cycle itself** (`cycle_a → cycle_b → cycle_a`) and names the import that closed the loop. You can see at a glance where to cut.

Why forbid it? Lune type-checks and evaluates a module’s dependencies **first** (`geometry` before `main`). With a cycle there is no “first”. It is the same reasoning that made `let x = x + 1` a `RUN0005` in chapter 4, appearing at the granularity of files.

There are three fixes: extract the common part into a third module, make the dependency point one way only, or put the two back into one file (mutual reference inside a single file is fine).

10.5 When a module is not found — the search order

`import foo.bar` looks for `foo/bar.lune`. It looks in three places, in this order.

1. the directory of the entry file
2. the current directory
3. directories added with `--module-path` (which may be repeated)

The third is useful when a shared library lives in a directory of its own. Here is `lib/shared.lune` (module `shared`) used from `usesshared.lune`:

```
$ lune --check usesshared.lune
error[MOD0001]: module not found: shared
$ lune --module-path lib --check usesshared.lune
type check OK
$ lune --module-path lib --eval answer usesshared.lune
42
```

When it is not found, the diagnostic **tells you where it looked**.

```
error[MOD0001]: module not found: nowhere
--> missing.lune:2:1
|
2 | import nowhere
  | ^^^^^^^ no matching .lune file was found
  = hint: searched: .
  = help: run `lune explain MOD0001` for a detailed explanation
```

(`searched:` really prints absolute paths; the book abbreviates them.)

When the file is found but the name tag differs, the diagnostic is a different one. With `mismatch.lune` containing `module totally.different`:

```
error[MOD0003]: module declaration mismatch: expected mismatch, got
totally.different
--> mismatch.lune:1:1
|
1 | module totally.different
  | ^^^^^^^ module name does not match the import path
  = hint: change the declaration to `module mismatch` or import it as
`totally.different`
  = help: run `lune explain MOD0003` for a detailed explanation
```

Note that the hint offers **the fix in both directions** (change the tag, or change the import). Which one is right is a design decision, so the machine does not choose; it

lays out both. A good example of “a hint gives cause and remedy”, which chapter 11 covers.

10.6 External imports — `java.*` and `std.*`

Finally, there are imports that do not point at a Lune file.

```
module basics
import java.time.LocalDate

def today(): String =
  LocalDate.now().toString()
```

An import beginning with `java.`, `javax.`, `kotlin.` or `std.` is treated as an **external import** and is not resolved to a file. The type checker merely registers the last name (`LocalDate`) as `Any`, so this definition type-checks — and fails at run time if you actually call it. JVM interoperation is future work (appendix E); v0.1 only accepts the syntax.

Since everything passes as `Any`, external imports are a region where type checking does not protect you. It is wise not to use them in programs written for v0.1.

Break it — produce two of this chapter’s `MOD` diagnostics yourself: `MOD0001` (not found) and `MOD0003` (mismatched tag). You can also move from one to the other: keep `import nowhere`, create `nowhere.lune`, and write `module wrong.name` in it — `MOD0001` becomes `MOD0003`.

```
error[MOD0003]: module declaration mismatch: expected nowhere, got
wrong.name
  --> nowhere.lune:1:1
    |
1  | module wrong.name
    | ^^^^^^^ module name does not match the import path
    | = hint: change the declaration to `module nowhere` or import it as
`wrong.name`
    | = help: run `lune explain MOD0003` for a detailed explanation
```

You can feel that “find the file” and “check the name tag” are separate stages. Note also that `module something.else` is a syntax error (`PRS0002`) because `else` is a keyword — module names cannot contain keywords.

Summary

Concept	In one line
<code>module a.b</code>	the name tag; must match <code>a/b.lune</code> (the entry file is exempt)
<code>import a.b</code>	the other module's top-level declarations enter your environment (used unqualified)
namespace access	<code>a.b.f()</code> is not supported; watch out for name collisions yourself
canonical form	<code>module</code> and the <code>import</code> lines run together, with no blank line
cycles	<code>MOD0002</code> ; dependencies are evaluated first, so no loops
search order	the entry file's directory → the current directory → -- <code>module-path</code>
diagnostics	<code>MOD0001</code> not found (with where it looked) / <code>MOD0003</code> mismatched tag (hints both ways)
external imports	<code>java.*</code> , <code>std.*</code> and friends are registered as <code>Any</code> only; effectively unusable in v0.1

Exercises

Exercise 10-1 (★) Create a file containing `import nowhere` to produce `MOD0001`, then create `nowhere.lune` containing `module wrong.name` to turn it into `MOD0003`.

Answer

Exactly §10.5 and the “Break it” box. `MOD0001`'s hint is where it looked; `MOD0003`'s hint is the two-way choice (fix the tag or fix the import). The change from one diagnostic to the other shows that checking proceeds in the order: resolve the file, then check the name tag.

Exercise 10-2 (★★) Move chapter 6's `Item` record and its aggregation into `shop/items.lune` (`module shop.items`), and compute the total from a `main` in another file.

Answer

```
module shop.items

# Exercise 10-2: split the data and its operations out into a module.
record Item:
  name: String
  price: Int

def priceOf(item: Item): Int =
  item.price

def total(items: List[Item]): Int =
  fold(map(items, priceOf), 0, fn a x -> a + x)
```

```
module shop_main
import shop.items

# Imported names are used unqualified: Item, priceOf and total alike.
let cart = [Item(name = "pen", price = 120), Item(name = "note", price =
200)]

let sum = total(cart)
```

```
$ lune --eval sum shop_main.lune
320
```

`record Item` is carried across by the import too, so the using side builds it normally with `Item(name = ..., price = ...)`. Types, functions and values all come across alike: every top-level declaration does.

Exercise 10-3 (★★) Put a `module shared` file in a `lib/` directory and load it with `--module-path`. Check what happens without the flag as well.

Answer

As in §10.5: without the flag, `MOD0001`; with it, it passes. Note that `lib/shared.lune` declares `module shared`, not `module lib.shared` — `--module-path lib` says “start the search at `lib`”, so `lib` is not part of the module name.

Exercise 10-4 (★★★) Create a cycle between two modules to produce `MOD0002`, then resolve it by extracting the shared part into a third module.

Answer

`cycle_a` $\rightleftharpoons$ `cycle_b` makes the cycle (§10.4). To resolve it, move the definitions both of them need into `common.lune` and have `cycle_a` and `cycle_b` import only `common`. The dependency graph turns from a loop into a tree, so an evaluation order exists again. The diagnostic shows the path (`a` $\rightarrow$ `b` $\rightarrow$ `a`), which is where to start when deciding which edge to cut.

Exercise 10-5 (★) A file containing `import java.time.LocalDate` passes the type check. Why? And why is that not safe?

Answer

An external import merely registers the last name, `LocalDate`, as `Any`, without resolving anything (§10.6). `Any` can be used as anything, so a mistake such as `LocalDate.nonexistentMethod()` slips through the type check and is not found until run time. Type checking does not protect you there — which is why v0.1 programs should avoid external imports.

More precisely — the determination of search roots, the test for external imports, and the normative loading order and cycle detection are in `documents/MODULE_LOADING_SPEC.md`. The `MOD` family of diagnostics is also collected in `documents/ERROR_INDEX.md`. The code examples of this chapter live in `books/examples-en/ch10/` and are all verified against the real CLI.

Chapter 11: Talking to the Compiler

Across the previous chapters you have met nearly thirty diagnostics. You have been told which name was a typo, been handed a counterexample for a missing `match` case, and been stopped at compile time for forgetting about null. This chapter turns that experience into **a system**.

To Lune, a diagnostic is not a notice of failure but half of the language's user interface. Every diagnostic code comes with an explanation written to teach, and a test guarantees that it does. Riding on that design, this chapter's goal is to turn the compiler from a noisy gatekeeper into a tutor sitting next to you.

11.1 The anatomy of a diagnostic — six parts

Chapter 1 touched on the structure of a diagnostic; here it is dissected completely. Our specimen is `TYP0008` (chapter 5), which has every part.

```
error[TYP0008]: refutable pattern in let binding: Some(x)
--> refutable.lune:3:5
|
3 | let Some(x) = Some(1)
  |      ^^^^ this pattern can fail to match
    = hint: the pattern does not cover None
    = hint: use `match` to handle all cases of Option[Int]
    = help: run `lune explain TYP0008` for a detailed explanation
```

Part	Here	Role
severity	<code>error</code>	<code>error</code> (the check fails) or <code>warning</code> (it passes, but look)
code	<code>TYP0008</code>	the diagnostic's number — the key to looking it up (§11.3)
summary	<code>refutable pattern in let...</code>	one line on what is wrong
location and source	<code>--> refutable.lune:3:5</code> and <code>^^^^</code>	where, and which part
label	<code>this pattern can fail to match</code>	what the <code>^^^^</code> is pointing at
hint / help	<code>= hint: ... x2</code>	suggested fixes, and the way to learn more

The knack is to read **summary** → **label** → **hint**. The summary tells you what this is about, the label pins down where, the hint gives the next move. When there are several hints they are ordered **cause** → **remedy** (here, “does not cover None” is the cause and “use `match`” the remedy).

A `warning` does not stop the check. As `TYP0009` in chapter 5 showed, with only warnings you still get `type check OK` at the end and the check counts as successful. But a warning marks “it runs, but probably not as you intended” — make a habit of not leaving them.

11.2 The code scheme — seven families and the stages of checking

The first three letters of a code give its **family**, and the family corresponds directly to a **stage of the compiler**. Source code passes checkpoints in order — layout, lexing, parsing, module resolution, type checking — and only what clears them all is run.

Family	Stage	Example	Mostly met in
LAY	layout (indentation)	inconsistent indentation	chapter 2
LXL	lexing	an unknown character, an unterminated string	chapter 2
PRS	parsing	an unexpected token	chapters 2 and 6
MOD	module resolution	an import not found, a cycle	chapter 10
TYP	type checking	undefined names, type mismatches, exhaustiveness	throughout
REC	record checking	missing, unknown or duplicate fields	chapter 6
RUN	run time	division by zero, recursive thunks	chapter 4

Two things follow from the table. First, **the earlier the family, the earlier it catches you** — while an `LXL` error stands, the conversation about types has not even begun. Second, only `RUN` is at run time. A `RUN` diagnostic can appear even after `--check` passes (chapter 4’s `crash()`, or division by zero): type checking guards a great deal, but not everything.

There are 31 codes in all today. Appendix C has the list, and the full catalogue of explanations is `documents/ERROR_INDEX.md`.

11.3 explain — an error number is for looking up

There are three doors to a diagnostic’s explanation.

```
$ lune explain TYP0007          # from the shell
```

```
lune> :explain TYP0007          # from inside the REPL
```

And to read them all at once, `lune explain --index` — which is what `ERROR_INDEX.md` actually is (the file is the output of that command, saved; a test fails when it goes stale).

Each explanation has three parts — what happened, the smallest example that reproduces it, and how to fix it — as you read for `TYP0001` in chapter 1 and `RUN0005` in chapter 4. Ask about a code that does not exist and you get the list of codes you can ask about.

```
$ lune explain ZZZ9999
error: no explanation for diagnostic code 'ZZZ9999'
available codes: LAY0001, LAY0002, LXL0001, LXL0002, LXL0003, LXL0004,
MOD0001, MOD0002, MOD0003, PRS0001, PRS0002, REC0001, REC0002, REC0003,
REC0004, REC0005, REC0006, RUN0005, RUN0006, TYP0001, TYP0003, TYP0004,
TYP0005, TYP0006, TYP0007, TYP0008, TYP0009, TYP0010, TYP0011, TYP0012,
TYP0013
```

“An error number is not to be memorised; it is **the key to looking it up**” — that is how to use the code scheme.

11.4 Errors a machine can fix — did-you-mean and lune fix

Some diagnostics have exactly one obvious fix. Typos are the archetype. Here is `typos.lune`, with two planted deliberately:

```
module stats

let values = [3, 1, 4, 1, 5]

let total = fold(valeus, 0, fn a x -> a + x)

let count = lenght(values)
```

```
$ lune --check typos.lune
```

```
error[TYP0001]: undefined name: valeus
--> typos.lune:5:18
|
5 | let total = fold(valeus, 0, fn a x -> a + x)
|                   ^^^^^^^ name is not defined
  = hint: did you mean `values`?
  = help: run `lune explain TYP0001` for a detailed explanation
```

`--check` reports only the first, but `lune fix` **applies the hints mechanically, re-checks the result**, and repeats until nothing fixable remains.

```
$ lune fix typos.lune
module stats

let values = [3, 1, 4, 1, 5]

let total = fold(values, 0, fn a x -> a + x)

let count = length(values)
```

Both `valeus` and `lenght` were fixed in one go. `--write` edits the file in place; `--check` turns it into a CI check that fails when anything fixable is left.

```
$ lune fix --check typos.lune
typos.lune: 2 auto-fixable issue(s)
```

`fix`, on the other hand, **does not touch what it cannot decide**. Run it on chapter 2's `annot.lune` (`let n: Int = "hello"`) and the source comes back unchanged. Whether the `Int` or the `"hello"` should change is a design decision, and a machine cannot make it. Remember that line: **if the hint is unique it is the machine's job; if judgement is required it is yours**.

11.5 Error-driven development — writing by following the witness

To finish, here is the development style this book has been promising all along, in one pass: **write the skeleton first, and ask the compiler what is missing**.

The subject is rock-paper-scissors. We judge whether `a` beats `b`. Start with the type and the one case we are sure of.

```

lune> type Hand =
...     | Rock
...     | Paper
...     | Scissors
...
ok
lune> def beats(a: Hand, b: Hand): Bool =
...     match (a, b):
...         | (Rock, Scissors) -> true
...
error[TYP0007]: non-exhaustive match: missing case (Paper, Rock)
--> <repl:2>:2:5
|
2 |     match (a, b):
|     ^^^^^ pattern (Paper, Rock) is not covered
= hint: add a case for (Paper, Rock), or a wildcard case `| _ -> ...`
= help: run `lune explain TYP0007` for a detailed explanation

```

The compiler has brought us **the next winning case to write**: (Paper, Rock) — paper beats rock. Do as it says and write all the winning cases.

```

lune> def beats(a: Hand, b: Hand): Bool =
...     match (a, b):
...         | (Rock, Scissors) -> true
...         | (Paper, Rock) -> true
...         | (Scissors, Paper) -> true
...
error[TYP0007]: non-exhaustive match: missing case (Rock, Rock)
--> <repl:2>:2:5
|
2 |     match (a, b):
|     ^^^^^ pattern (Rock, Rock) is not covered
= hint: add a case for (Rock, Rock), or a wildcard case `| _ -> ...`
= help: run `lune explain TYP0007` for a detailed explanation

```

This time it is (Rock, Rock) — we are being told that **draws exist**. Everything that is not a win is not a win, so catch it with a wildcard.

```

lune> def beats(a: Hand, b: Hand): Bool =
...     match (a, b):
...         | (Rock, Scissors) -> true
...         | (Paper, Rock) -> true
...         | (Scissors, Paper) -> true
...         | _ -> false
...
ok
lune> beats(Rock(), Scissors())
true : Bool
lune> beats(Scissors(), Rock())
false : Bool

```

Look back at what happened. We did not remember a single one of the gaps in the specification — paper's win, the existence of draws — **by ourselves**. The compiler brought each of them, in the form of a concrete counterexample. A diagnostic is not something to read but **a TODO list to work through**. That is error-driven development.

The more of the types you write first, the better the questions the compiler asks. Enumerate the shapes with an ADT, build the skeleton with `match`, run `--check`. When the gaps are filled, finish with `fix` and `fmt`. That cycle is the basic rhythm of writing Lune.

Break it — delete `| _ -> false` from `rps.lune`, run `--check` to see the witness, and then cover the cases by **enumerating the three draws and the three losses** instead of using a wildcard. Both styles have a virtue: the wildcard is shorter, while the enumeration protects you when a case is added to `Hand` (the lesson of exercise 5-1).

Summary

Concept	In one line
the parts of a diagnostic	severity / code / summary / location + label / hint / help; read summary → label → hint
the order of hints	cause → remedy
seven families	LAY, LXL, PRS, MOD, TYP, REC before running; only RUN at run time
<code>lune explain / :explain / --index</code>	all 31 codes have an explanation; the number is a key for looking up
<code>lune fix</code>	when the hint is unique, the machine fixes it (repeatedly); <code>--write</code> / <code>--check</code>
where fix stops	it does not touch fixes a machine cannot decide
error-driven development	skeleton → <code>--check</code> → work through the witnesses as a TODO list

Exercises

Exercise 11-1 (★) About the specimen in §11.1 (`TYP0008`): which hint states the cause and which the remedy? And why does `--check` come back as a failure (a non-zero exit)?

Answer

The cause is the first (“the pattern does not cover None”) and the remedy the second (“use `match` to handle all cases”). It fails because the severity is `error`. With only warnings (`TYP0009` and the like) the check succeeds and prints `type check OK`.

Exercise 11-2 (★★) Predict what `lune fix` does to `typos.lune` and to chapter 2’s `annot.lune`, then check.

Answer

`typos.lune` comes back as complete source with both typos fixed. `annot.lune` comes back **unchanged** — `TYP0003` has no unique, mechanically applicable hint. With `fix --check`, the former fails (non-zero exit) with “2 auto-fixable issue(s)” and the latter succeeds, having nothing to fix.

Exercise 11-3 (★★) Using `beats` as a building block, write `judge(a, b)`, returning the three values `"win"` / `"lose"` / `"draw"`.

Answer

```
def judge(a: Hand, b: Hand): String =  
  if beats(a, b):  
    "win"  
  elif beats(b, a):  
    "lose"  
  else:  
    "draw"
```

```
$ lune --eval tied ex11-3.lune  
"draw"
```

The **rule** for who wins is written in exactly one place, `beats`, and `judge` merely asks it twice. Not writing the rule twice is the point.

Exercise 11-4 (★★★, in reverse, comprehensive) Write the smallest code that produces a diagnostic from each of the five families `LXL`, `PRS`, `TYP`, `REC` and `RUN` (bonus points for a code this book has not yet shown you).

Answer

For example `let x = $1` (LXL0001), `let a = (1` (a PRS code — an unclosed parenthesis), `1 + true` (TYP0003), `User(name = "X", name = "Y", age = 1)` (REC0004, chapter 6), and evaluating a binding containing `1 / 0` with `--eval` (RUN0006). A bonus: ending a file with an unclosed `###` gives LXL0003 (unterminated block comment). Check your finds against the index of all 31 codes (appendix C / `ERROR_INDEX.md`).

Exercise 11-5 (★) Skim the output of `lune explain --index`, find a diagnostic code this book has not shown you, and read its explanation.

Answer

`MOD0002` (a cyclic import), for instance, has not appeared yet — that is chapter 10's treat. `LAY0002` (an unmatched closing bracket) and `REC0001` (a duplicated field in a record declaration) also take some contriving to produce on purpose.

More precisely — the normative diagnostic model and display format are in `documents/ERROR_DIAGNOSTICS_SPEC.md`; the catalogue of every code is `documents/ERROR_INDEX.md` (regenerated with `lune explain --index`); the rules for applying fixes are in `documents/ERROR_DIAGNOSTICS_SPEC.md` §9.5. The code examples of this chapter live in `books/examples-en/ch11/` and are all verified against the real CLI.

Chapter 12: The REPL, Formatting and Checking

Chapter 11 was about the conversation with diagnostics. This chapter is about the tools themselves — the REPL’s commands, the canonical formatter `lune fmt`, the CLI’s checking flags, and a recipe for lining them up in CI.

One thing worth saying first. Every Lune tool is **a tool for checking your answer**. Unsure of a type? `:type`. Suspicious about the order of evaluation? `:trace`. Arguing about formatting? `fmt`. Get into the habit of asking the machine instead of thinking it through in your head, and Lune becomes a remarkably straightforward language.

12.1 Every REPL command

`:help` tells you all of them.

```
lune> :help
commands: :help, :quit, :q, :env, :type NAME, :thunks [NAME], :trace
[on|off], :lang [en|ja], :explain CODE [en|ja]
```

Command	What it does	Learned in
<code>:help</code>	this list	—
<code>:quit</code> / <code>:q</code>	quit	chapter 1
<code>:env</code>	list every name in scope, with its type	this chapter
<code>:type NAME</code>	give the type of a name	this chapter
<code>:thunks [NAME]</code>	show a binding’s evaluation state without evaluating it	chapter 4
<code>:trace [on off]</code>	turn the running commentary on forcing on and off	chapter 4
<code>:lang [en ja]</code>	switch the diagnostic language	chapter 1
<code>:explain CODE</code>	explain a diagnostic code	chapter 11

Multi-line input continues when a line ends in `=`, `:` or `->` (the `...` prompt) and finishes at a blank line — the behaviour you have used since chapter 1.

Started in a terminal, readline-style line editing and history (the up and down arrows) work as well. Where the environment supports it, history is kept in `~/.lune_history`, so an experiment from a previous session can be called back.

`import` works in the REPL too: modules are resolved from the directory you started in, plus any `--module-path` (chapter 10).

12.2 `:type` and `:env` — asking the types

`:type` may be the most-used tool in this book. It works on your own functions and on the prelude's alike.

```
lune> def add(a: Int, b: Int): Int =  
...     a + b  
...  
ok  
lune> :type add  
add : Int -> Int -> Int
```

Ask it about a record constructor and the answer includes the field names.

```
lune> :type User  
User : (name: String, age: Int) -> User
```

Chapter 6 taught that records must be built with named fields; here that fact is written into the type itself.

`:env` lists **every** name in scope. It includes the prelude, so the list is long, but the interesting part is seeing your own definitions mixed in at the end.

```
lune> let x = 40  
ok  
lune> :env  
Cons : [T] T -> List[T] -> List[T]  
Err : [T, E] E -> Result[T, E]  
...  
take : [T] List[T] -> Int -> List[T]  
takeWhile : [T] List[T] -> (T -> Bool) -> List[T]  
tick : () -> Int  
x : Int  
zip : [T] List[T] -> List[U] -> List[Tuple[T, U]]
```

It is a quick index for “what was in the prelude again?” (the formal list is appendix B).

Ask about a name it does not know and you are told off, naturally.

```
lune> :type nosuch
error[TYP0003]: undefined name: nosuch
  = help: run `lune explain TYP0003` for a detailed explanation
```

(An aside: undefined names have a code of their own, `TYP0001`, so `TYP0003` here is a small oversight in the implementation. Consistency of diagnostic codes is the sort of thing that ought to be guarded by a test, too.)

12.3 lune fmt — the formatter that ends the argument

Two spaces or four? A space around `fn a x -> a+x`? Lune reduces the time spent on that kind of argument to zero. **There is one canonical form**, and `lune fmt` moves code to it.

Here is a messy file, `messy.lune`:

```
module messy

let  nums = [ 1,2 ,3 ]

# sum them up
def total( xs : List[Int] ) : Int =
    fold( xs, 0, fn a x -> a+x )

let answer=total(nums)
```

```
$ lune fmt messy.lune
module messy

let nums = [1, 2, 3]

# sum them up
def total(xs: List[Int]): Int =
    fold(xs, 0, fn a x -> a + x)

let answer = total(nums)
```

The extra blank lines became one, the spacing around brackets was tidied, indentation became four spaces, and the `#` comment was kept where it was.

This formatting comes with two guarantees.

Idempotence — formatting an already-formatted result changes nothing. It never oscillates, so it is safe to put in CI.

It does not change meaning — `fmt` prints the formatted AST and then **re-reads its own output to check that the syntax tree still matches** before handing the result back. If it does not match, it gives up formatting and reports an error. (That is exactly how the nested `let-in` bug mentioned in chapter 2 was caught.) Formatting will never quietly break your program.

There are three modes.

```
$ lune fmt FILE           # print the formatted result (the file is
untouched)
$ lune fmt --write FILE   # format the file in place
$ lune fmt --check FILE   # fail if formatting is needed (for CI)
```

`--check` names the files that differ and exits with 1.

```
$ lune fmt --check messy.lune
would reformat messy.lune
```

One limitation: a file containing `###` block comments cannot be formatted yet.

```
$ lune fmt bc.lune
error: bc.lune: lune fmt does not support `###` block comments yet
```

It merely declines to format (and so cannot break anything), which is the safe behaviour, but if `fmt` is in your CI the current practice is to avoid `###` and use a run of `#` instead.

12.4 The CLI's checking flags

The main flags of the `lune` command, in the order you meet them.

Command	What it does
<code>lune --repl</code>	start the REPL
<code>lune --check FILE</code>	type-check only (no evaluation)
<code>lune --eval NAME FILE</code>	evaluate the binding NAME and print it
<code>lune --eval NAME --trace FILE</code>	as above, plus a trace of forcing on stderr (chapter 4)
<code>lune --module-path DIR</code>	add a module search root (chapter 10, repeatable)
<code>lune --tokens FILE</code>	print the result of lexing (the token stream)
<code>lune explain CODE</code>	explain a diagnostic code (chapter 11; <code>--index</code> for all)
<code>lune fmt / lune fix</code>	format / auto-fix (§12.3, chapter 11)
<code>--lang en ja</code>	the diagnostic language. The default comes from <code>LUNE_LANG</code> ; the flag wins

`--tokens` is not an everyday tool, but it is handy for peering at how layout (indentation) is handled. It shows you the reality behind “indentation is part of the syntax” from chapter 2.

```
$ lune --tokens geometry.lune
1:1    MODULE  'module'      None
1:8    IDENT   'geometry'    'geometry'
1:16   NEWLINE ''          None
3:1    DEF     'def'        None
3:5    IDENT   'circleArea'  'circleArea'
...
```

Note also the relationship between `--check` and `--eval`. **`--eval` does not type-check** — which is how chapter 4 was able to produce `RUN0005` (a recursive thunk). To check before running, run `--check` first.

Evaluating a binding that does not exist is a runtime error.

```
$ lune --eval nosuch geometry.lune
error[RUN0006]: undefined variable: nosuch
```

12.5 A small CI recipe

Line the tools up and you have a check for the project. Exit codes are uniform — non-zero when something is wrong — so a shell `&&` makes them one pipeline.

```
lune fmt --check src/*.lune && lune --check src/main.lune && lune fix --check src/*.lune
```

The three mean:

1. `fmt --check` — is every file in canonical form (has any formatting been missed)?
2. `--check` — does it type-check?
3. `fix --check` — is anything the machine could fix (a typo, say) still there?

The exit code rules, collected:

Situation	Exit code
the check passes	0
warnings only (<code>TYP0009</code> and the like)	0
there are errors	1
<code>fmt --check</code> finds a difference	1
<code>fix --check</code> finds something fixable	1

Watch the second row. As chapter 5 showed, warnings alone still count as success. If you do not want warnings slipping past either, one more step is needed: fail when `warning` appears in the output.

```
lune --check src/main.lune 2>&1 | tee /tmp/out; ! grep -q warning /tmp/out
```

Break it — format `messy.lune` with `fmt --write` and then run `fmt --check` again. This time it should say nothing (a demonstration of idempotence). While you are there, read the formatted result and count the differences between your own habits and the canonical form.

Summary

Tool	In one line
<code>:help</code>	the list of REPL commands; start here when stuck
<code>:type / :env</code>	ask the types / list every name in scope
<code>lune fmt</code>	format to the canonical form; idempotent and meaning-preserving (self-checked by re-parsing)
the three <code>fmt</code> modes	<code>stdout / --write / --check</code> (CI)
<code>fmt</code> 's limitation	<code>###</code> block comments are unsupported (it declines, safely)
<code>--check</code> VS <code>--eval</code>	<code>--eval</code> does not type-check; run <code>--check</code> first if you want checking
<code>--tokens</code>	a window onto lexing and layout
the CI recipe	<code>fmt --check && --check && fix --check ; warnings alone still exit 0</code>

Exercises

Exercise 12-1 (★) Predict the result of running `fmt` on `messy.lune`, then check. Where were you wrong?

Answer

As in §12.3. The usual surprises are that runs of blank lines are squeezed to one, the spacing around the brackets and the colons of `def total( xs : List[Int] ) : Int =`, and the spaces appearing in `a+x`. Check too that the `#` comment survives.

Exercise 12-2 (★) Use `:env` to confirm that `zipWith` is in the prelude, and what its type is. Does it match your memory of using it in chapter 8?

Answer

`zipWith : [T, U, V] List[T] -> List[U] -> (T -> U -> V) -> List[V]`. “Takes two lists and a two-argument function, and returns one list” — what you did for the moving average in exercise 8-3, written in the type. `:type zipWith` gives the same answer.

Exercise 12-3 (★★) Checking with `:type` as you go, build a “multiply by three” function from `scale(factor, n)` by partial application, and `map` it over a list. Check

the type of the partial application itself along the way, as in `:type scale(3)`.

Answer

```
module answers

# Exercise 12-3: tools built by partial application, checked with :type
# as they were assembled.
def scale(factor: Int, n: Int): Int =
  factor * n

let double = scale(2)

let tripled = map([1, 2, 3], scale(3))

let doubled = map([1, 2, 3], double)
```

```
$ lune --eval tripled ex12-3.lune
(3 6 9)
$ lune --eval doubled ex12-3.lune
(2 4 6)
```

In the REPL, `:type scale` is `Int -> Int -> Int`, and after `let triple = scale(3)`, `:type triple` is `Int -> Int`. The right-associativity of `->` from chapter 3, observed directly.

Exercise 12-4 (★★) Run the CI recipe of §12.5 against your own project (the exercise files of this book will do). If you find a file that fails, explain the cause from the diagnostic.

Answer

Trying it on this book's `books/examples-en/`, the deliberately broken files (`ch02/annot.lune`, `ch11/typos.lune` and so on) are caught. `typos.lune` fails both `--check` and `fix --check`, but for different reasons — the first is “it does not type-check”, the second “something fixable is still there”. Reading that the same file fails for **two different reasons** is the point.

Exercise 12-5 (★, in reverse) Make `lune fmt` give up formatting.

Answer

Run `fmt` on a file containing a `###` block comment and it gives up with `lune fmt does not support ### block comments yet` (§12.3). The other way to make it give up is a bug in the formatter itself failing the meaning-preservation self-check (`formatter changed the program's meaning (internal bug)`). That one is hard to produce on purpose, but if you find one it is a bug worth reporting — one was in fact found while this book was being written.

More precisely — the REPL is specified in `documents/REPL_SPEC.md` (including the display format of `:thunks` and `:trace`); the formatter's guarantees and limitations are in `documents/FORMATTER_SPEC.md`; every CLI flag is in appendix D. The code examples of this chapter live in `books/examples-en/ch12/` and are all verified against the real CLI.

Chapter 13: Building a Program — Case Studies

Every part is now in hand. This chapter writes three small programs **from design to completion**. No new syntax appears; instead we see how the tools learned separately fit together.

All three follow the same procedure: **decide the types** → **write the skeleton** → **fill it in guided by the compiler's witnesses** → **finish with `fmt` and `--check`**. In other words, three real runs of the error-driven development of chapter 11.

13.1 Text statistics — one pass with fold and a record

K&R's introduction starts with a program that counts characters. In tribute, let us build statistics about words.

One caveat first. v0.1's strings have only `length` and `+` (concatenation), with no tool for splitting a string into words. So the input is a list of already-split words, `List[String]`. Splitting is a job for outside the program — that is the design.

Decide the types. The result we want is a triple: the number of words, the total number of characters, and the longest word. We want to carry it with names, so: a record (chapter 6).

```
record Stats:
  count: Int
  totalChars: Int
  longest: String
```

Write the skeleton. We are collapsing a list into a single value, so `fold` (chapter 8). Which means all we need is a function that makes a new `Stats` from the running `stats` and the next word. Here is `stats.lune`:

```

module stats

record Stats:
  count: Int
  totalChars: Int
  longest: String

let empty = Stats(count = 0, totalChars = 0, longest = "")

def step(s: Stats, w: String): Stats =
  let n = length(w)
  let longer = if n > length(s.longest) then w else s.longest
  Stats(count = s.count + 1, totalChars = s.totalChars + n, longest =
longer)

def summarize(words: List[String]): Stats =
  fold(words, empty, step)

def averageLength(s: Stats): Double? =
  if s.count == 0 then null else s.totalChars / s.count

let words = ["the", "quick", "brown", "fox", "jumps"]

let summary = summarize(words)

let average = averageLength(summary)

let emptyAverage = averageLength(summarize([]))

```

```

$ lune --eval summary stats.lune
{ count = 5, totalChars = 21, longest = "quick" }
$ lune --eval average stats.lune
4.2
$ lune --eval emptyAverage stats.lune
null

```

There are three design decisions in there.

The list is walked only once. Computing the three statistics separately would mean three `fold`s; making the accumulator a record does it in one. The running value of a `fold` need not be a number — that is what `fold` is really for.

The average is a `Double?` . With zero words there is no average. Returning 0 would be a lie, so it returns `null` (chapter 7). The caller chooses whether to land on `?? 0.0` or to branch with a `match` .

`step` is its own function. It could have been a lambda, but giving it a name means it can be tried on its own in the REPL (`step(empty, "hi")`). Name the things you want to test — a practical instinct.

13.2 A ledger — ADTs, Result and splitting into modules

Next, something more like real work: record income and expenses and produce a balance.

Decide the types. An entry is either income or an expense — a job for a sum type (chapter 5). The data and its operations go into a module together (chapter 10). Here is `ledger/entry.lune`:

```
module ledger.entry

type Entry =
  | Income(label: String, amount: Int)
  | Expense(label: String, amount: Int)

def signedAmount(e: Entry): Int =
  match e:
    | Income(_, amount) -> amount
    | Expense(_, amount) -> 0 - amount

def labelOf(e: Entry): String =
  match e:
    | Income(label, _) -> label
    | Expense(label, _) -> label

def validate(e: Entry): Result[Entry, String] =
  if signedAmount(e) == 0 then Err("amount must not be zero: " +
    labelOf(e)) else Ok(e)
```

This is the heart of the design. We decided **not to carry the sign in the type**. An `Expense` holds a positive `amount`, and `signedAmount` applies the sign. That way the question “do we record expenses as negative?” never arises — a small but essential choice about whether meaning lives in the type or in a function.

`validate` returns failure **as a value** (chapter 5). Nothing is thrown, so the caller cannot ignore the failure. Here is `ledger_main.lune`:

```

module ledger_main
import ledger.entry

let entries = [Income("salary", 3000), Expense("rent", 1200),
Expense("food", 400)]

let balance = fold(map(entries, signedAmount), 0, fn a x -> a + x)

let expenses = filter(entries, fn e: Entry -> signedAmount(e) < 0)

let rejected = validate(Expense("zero", 0))

let accepted = validate(Income("bonus", 500))

```

```

$ lune --eval balance ledger_main.lune
1400
$ lune --eval expenses ledger_main.lune
(Expense("rent", 1200) Expense("food", 400))
$ lune --eval rejected ledger_main.lune
Err("amount must not be zero: zero")
$ lune --eval accepted ledger_main.lune
Ok(Income("bonus", 500))

```

“map to signed amounts and fold them” — chapter 6’s aggregation pattern, working unchanged. Pulling out the expenses with filter is the same idea.

Experience the benefit of witnesses here. Add a third case to Entry — a Transfer, say — and TYP0007 appears in both signedAmount and labelOf. The compiler points out, exhaustively, that you have not decided what sign a transfer has. That is what chapter 11 taught, and it is what a design robust to changing requirements actually means.

A note on v0.1 — ADT constructors are called with **positional** arguments (Income("salary", 3000)). Writing them by name, as Income(label = "salary", amount = 3000), is rejected with TYP0012. Constructors can be partially applied (chapter 3), so Income(amount = 3000) would mean “a partial application with the first argument unfilled”, leaving nothing for the names to correspond to. **It is records that require names**, and there positional arguments give REC0006 instead (chapter 6) — the two are exactly mirror images.

13.3 A laboratory of sequences — infinite lists and counting evaluations

The last is a comprehensive exercise in lazy evaluation. We build the Collatz sequence (halve an even number; treble an odd one and add one; continue until it reaches 1) as an infinite list.

Writing “halve if even” needs care about **which** division. `/` always returns a `Double`, so `n / 2` is not an `Int`. To divide and stay in integers, use floor division `//` (§2.2).

```
lune> 7 // 2
3 : Int
lune> 7 / 2
3.5 : Double
```

Here is `collatz.lune`:

```
module collatz

# `//` is integer floor division. `/` always returns a Double, so staying
in
# Int means `//`.
def next(n: Int): Int =
  if n % 2 == 0 then n // 2 else 3 * n + 1

let fromSix = take(iterate(next, 6), 9)

let fromSeven = takeWhile(iterate(next, 7), fn n: Int -> n != 1)
```

```
$ lune --eval fromSix collatz.lune
(6 3 10 5 16 8 4 2 1)
$ lune --eval fromSeven collatz.lune
(7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2)
```

`iterate(next, 6)` is “the infinite Collatz sequence starting from 6” (chapter 8). How much of it to look at is the caller’s decision — nine with `take`, or “up to just before 1” with `takeWhile`. Separating the definition of a sequence from how much of it you use, once again.

And finally, let us confirm **with a number** that laziness really is doing its work. Prepare an “expensive” computation with a `tick()` (chapter 4) inside, and measure how many times it runs when two elements are taken from a five-element list. Here is `counted.lune`:

```

module counted

# An "expensive" computation with a tick() in it: the counter records how
# many times it was called.
def costly(n: Int): Int =
  seq tick() (n * 2)

let doubled = map([1, 2, 3, 4, 5], costly)

# Take just the first two, and evaluate them all the way down.
let firstTwo = deepForce take(doubled, 2)

# How many times costly ran while doing so.
let cost = seq firstTwo tickCount()

```

```

$ lune --eval firstTwo counted.lune
(2 4)
$ lune --eval cost counted.lune
2

```

We mapped over five elements and `costly` ran twice. A strict language would run it five times and then throw three results away. The benefit of laziness, tracked since chapter 4, finally appears as a number.

Notice also why `deepForce` is needed. `take(doubled, 2)` alone materialises only the spine (the chain of `Cons` cells), leaving the elements as thunks — with `:thunks` you can see it stop at `Cons(<thunk>, <thunk>)`, the same view as in chapter 8. That you can choose *how far* to evaluate is itself an expression of Lune’s design.

13.4 What the three cases have in common

Put side by side, a common pattern emerges.

1. **Turn the shape into a type** (record or ADT — product or sum).
2. **Write with immutable transformations** (`map` / `filter` / `fold`; if an imperative part is needed, seal it in a block).
3. **Express “nothing” and “failed” as values** (`T?` / `Result` / a monomorphic result type).
4. **Let the compiler watch for gaps** (exhaustiveness in `match`, filling in guided by witnesses).
5. **Build your own instrument to check with** (as `costly` and `tick()` do — confirm with a number, not with “it should be fast”).

And every one of the programs was formatted with `lune fmt` and passed through `lune --check` before being called finished (the CI recipe of chapter 12).

Exercises

Exercise 13-1 (★★) Add “the shortest word” to `Stats`, without breaking on an empty list.

Answer

Initialising it to `""` would make the shortest always the empty string, so express “not yet any” with `null` (chapter 7).

```
record Stats:
  count: Int
  totalChars: Int
  longest: String
  shortest: String?

def pickShortest(current: String?, w: String): String =
  if current == null:
    w
  elif length(w) < length(current):
    w
  else:
    current
```

```
$ lune --eval summary ex13-1.lune
{ count = 5, totalChars = 21, longest = "quick", shortest = "the" }
```

After the first line of `pickShortest` checks `current == null`, `current` is narrowed to `String`, so `length(current)` can be written (the narrowing of chapter 7). Note also that the one-line form `if ... then ... elif` does not exist — `elif` belongs to the block form only.

Exercise 13-2 (★★) Add `Transfer(label: String, amount: Int)` to `Entry` (a transfer between accounts, which does not affect the balance). Observe what the compiler demands.

Answer

Adding one line to the `type` produces `TYP0007` (non-exhaustive match) in **both** `signedAmount` and `labelOf`. Adding `| Transfer(_, _) -> 0` (no effect on the balance) and `| Transfer(label, _) -> label` makes it pass.

Change a type in one place and the compiler lists every place affected — this is why the wildcard `_ -> ...` is worth avoiding (exercise 5-1). Had it been written with a wildcard, `Transfer` might quietly have been treated as an expense.

Exercise 13-3 (★★) What does `cost` become if `take(doubled, 2)` in `counted.lune` becomes `take(doubled, 5)`? And what if `deepForce` is removed? Predict, then check.

Answer

It becomes `5` (all five elements are evaluated). Without `deepForce`, `cost` is `0` — `take` materialises only the spine and never touches the elements' thunks. The chapter 8 lesson that “having a list” and “having computed its contents” are different things, confirmed with a number.

Exercise 13-4 (★★★) `-7 // 2` is `-4`, not `-3`. Explain why, together with the value of `-7 % 2`. Predict first, check in the REPL, then answer.

Answer

Start by looking at both values.

```
lune> -7 // 2
-4 : Int
lune> -7 % 2
1 : Int
```

// rounds **towards negative infinity** (floor), not towards zero (truncation). $-7 / 2$ is -3.5 , and the floor of that is -4 .

Why not -3 ? Because // and % are not two unrelated operations: they **have to add up as a pair**. Whatever the signs, quotient and remainder are tied by this identity.

```
a == (a // b) * b + (a % b)
```

Check it:

```
lune> (-7 // 2) * 2 + (-7 % 2)
-7 : Int
```

Suppose // truncated towards zero and returned -3 . The % side (1) is unchanged, so:

```
lune> (-3) * 2 + (-7 % 2)
-5 : Int
```

which is not -7 . Choosing -3 makes // and % contradict each other, and they stop deserving the names quotient and remainder. Once % is defined to return a remainder matching the sign of the divisor ($-7 \% 2$ is 1 , not -1), // must be floor. Neither can be decided by looking at it alone — which is the point of this exercise.

Only positive numbers appear in the Collatz sequence, so writing `next` never exposes the difference. It starts to matter the moment you divide a negative number. The rounding direction is specified in §9.1 of

`documents/LANGUAGE_SPEC.md`.

Exercise 13-5 (★★★, comprehensive) Pick one of the three case studies and extend it. For example: add a filter for “words of n characters or more” to the text statistics; add a monthly summary to the ledger (a `record Month` combined with `filter`); or write a function giving the number of steps a Collatz sequence takes to reach 1.

Answer

Taking the third: the number of steps is the `length` of the `takeWhile` result, plus one.

```
lune> length(takeWhile(iterate(next, 7), fn n: Int -> n != 1)) + 1  
17 : Int
```

`length` is being used on an infinite list, but `takeWhile` has made it finite first, so it terminates (§8.5). “Transform while infinite, make it finite, then consume” — whether you keep that order is the test of whether lazy evaluation is under your control.

More precisely — this chapter uses no new features. The specifications each section rests on: records in `documents/RECORD_FIELD_SPEC.md`, ADTs and `match` in `documents/MATCH_EXHAUSTIVENESS_SPEC.md`, lists and the lazy combinators in `documents/STANDARD_LIBRARY_SPEC.md` §6, and `seq / deepForce` in `documents/LAZY_EVALUATION_SPEC.md` §9–10. The code examples of this chapter live in `books/examples-en/ch13/` and are all verified against the real CLI.

Appendix A: Language Reference Manual

A normative list for Lune v0.1, readable independently of the main text. This is where you come to check “what was the rule again?”; why it is that way is in the chapters (and in appendix E).

The specifications in `documents/` are authoritative. This appendix draws on them, but the contents of the tables were either extracted mechanically from the implementation or confirmed by running them through the real compiler (see “Where this appendix comes from” at the end).

A.1 Lexical structure

Keywords

48 words are reserved. **Many of them are for the future and are syntax errors in v0.1** (see A.6).

IO	abstract	as	catch	class	
deepForce					
def	elif	else	extends	false	
final					
finally	fn	for	force	if	
implements					
import	in	init	interface	internal	lazy
let	match	module	new	null	
override					
private	protected	public	raise	record	seq
static	strict	super	then	this	
throw					
throws	true	try	type	var	
while					

`break`, `continue`, `return`, `export` and `use` are **not reserved**. They are treated as ordinary identifiers, so writing one gives “undefined name” (`TYP0001`). See A.6.

Identifiers

```
IDENT_START = [A-Za-z_]
IDENT_PART  = [A-Za-z0-9_]
IDENT       = IDENT_START IDENT_PART*
```

A lone `_` is the wildcard pattern; `_name` is an identifier.

Literals

Kind	Example	Type
integer	<code>42</code> , <code>1_000_000</code>	<code>Int</code> (arbitrary precision)
floating point	<code>3.14</code> , <code>1e-3</code>	<code>Double</code>
string	<code>"hello"</code>	<code>String</code>
character	<code>'x'</code>	<code>String</code> (see below)
boolean	<code>true</code> , <code>false</code>	<code>Bool</code>
null	<code>null</code>	<code>Null</code>
unit	<code>()</code>	<code>Unit</code>
list	<code>[1, 2, 3]</code>	<code>List[T]</code>
tuple	<code>(1, "a")</code>	<code>Tuple[Int, String]</code>

The `_` in an integer literal is a digit separator and is ignored. The string escapes are `\n` `\r` `\t` `\\` `\"` `\'` `\0`.

A character literal does not give a character type — `'x'` is a token of its own lexically, and anything other than exactly one character inside gives `LXL0002`. But **its type is `String`**. v0.1 has no character type (`Char`) and no standard library functions for characters. `let c: Char = 'x'` is treated exactly as a misspelled type name would be, giving `TYP0003`. `Char` belongs to the future specification (`LANGUAGE_FUTURE_SPEC.md`).

Comments

```
# a line comment
```

```
###
```

```
a block comment (they do not nest)
```

```
###
```

`//` is **not a comment** — it is floor division (§2.2). `/* ... */` is not a comment either.

Operator table

Highest precedence first. Operators on one line share a precedence.

Precedence	Operators	Associativity
1	<code>.</code> <code>?.</code> <code>()</code> <code>[]</code>	left
2	unary <code>!</code> , unary <code>-</code> , unary <code>+</code>	right
3	<code>*</code> <code>/</code> <code>//</code> <code>%</code>	left
4	<code>+</code> <code>-</code>	left
5	<code>::</code> <code>++</code>	right
6	<code>==</code> <code>!=</code> <code><</code> <code><=</code> <code>></code> <code>>=</code>	non-associative
7	<code>&&</code>	left
8	<code> </code>	left
9	<code>??</code>	right
10	<code> ></code>	left
11	<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>//=</code> <code>%=</code>	right

Non-associative means they cannot be chained. What stops it is the type checker rather than the parser, though: `1 < 2 < 3` parses, and fails when the left operand of the second `<` turns out to be a `Bool` (`<: expected numeric type, got Bool`).

`::` and `++` exist in the parser but are out of practical scope for v0.1's type checking and evaluation.

A.2 Layout rules

Blocks are expressed by indentation. Only **spaces** may begin a line; a tab gives the lexical error `LXL0004` (tabs are not allowed in indentation). The caret points at the tab itself, so you can see where it crept in.

```
error[LXL0004]: tabs are not allowed in indentation
--> tabs.lune:4:1
  |
4 |     1
  | ^ use spaces for indentation
  = hint: replace tabs with spaces
```

Blank lines and comment-only lines are **not subject to the indentation check** (see the rules below), so a tab in one of those is not reported. Nor are tabs inside strings or after the code on a line.

After lexing, layout processing produces `NEWLINE` / `INDENT` / `DEDENT`.

- Compare the width `n` of the leading whitespace with the top of the indentation stack, `top`.
- `n == top` → emit only `NEWLINE`.
- `n > top` → emit `INDENT` and push `n`.
- `n < top` → emit `DEDENT` until a matching value appears. If none matches, an indentation error (`LAY0001`).
- At end of file, emit `DEDENT` until the stack is empty, then `EOF`.

No layout tokens are produced inside brackets. `(` `[` `{` increase the depth by one and `)` `]` `}` decrease it, and while the depth is above zero a physical newline is just whitespace. That is why you may break lines freely inside brackets.

```
let xs = [
    1,
    2,
]
```

For constructs that expect a block after `:` (`if`, `while`, `for`, `match`, `record` and so on), the next significant tokens must be `NEWLINE` `INDENT`. After `=`, either one line or a block is allowed.

```
def f(): Int = 1

def g(): Int =
  1
```

Multi-line input in the REPL continues while a line ends in `=`, `:` or `->`, and finishes at a blank line (appendix D).

A.3 Grammar

Below, `NL` means one or more `NEWLINE`s. The complete EBNF is in §5–§10 of `documents/LEXER_PARSER_SPEC.md`; this is the skeleton drawn from it.

Files and declarations

```
file      ::= NL* module_decl? import_decl* top_decl* EOF
module_decl ::= "module" qualified_name NL+
import_decl ::= "import" qualified_name ("as" IDENT)? NL+
qualified_name ::= IDENT ("." IDENT)*

top_decl   ::= function_decl | type_decl | record_decl | let_decl |
var_decl

function_decl ::= "def" IDENT type_params? "(" params? ")" (":" type)? "="
body
let_decl      ::= ("strict" "let" | "let") pattern (":" type)? "=" body
var_decl      ::= "var" IDENT (":" type)? "=" body
type_decl     ::= "type" IDENT type_params? "=" NL INDENT constructor+
DEDENT
record_decl   ::= "record" IDENT type_params? ":" NL INDENT field+ DEDENT

constructor  ::= "|" IDENT "(" ctor_fields? ")"? NL
ctor_fields  ::= ctor_field ("," ctor_field)*
ctor_field   ::= ("strict" | "!=")? IDENT ":" type
field        ::= ("strict" | "!=")? IDENT ":" type NL
params       ::= param ("," param)*
param        ::= ("strict" | "!=")? IDENT ":" type
type_params  ::= "[" IDENT ("," IDENT)* "]"
body         ::= expr | NL INDENT block DEDENT
```

Expressions

Expressions are parsed with a Pratt parser. The precedences are those in A.1.

```

expr    ::= assign
assign  ::= pipeline (assign_op assign)?
primary ::= literal | IDENT | "(" expr ")" | list | tuple
         | if_expr | match_expr | lambda | block_expr
         | "lazy" expr | "force" expr | "deepForce" expr
         | "seq" expr expr | "while" expr ":" block | "for" pattern "in"
expr ":" block
         | "IO" ":" block | "raise" expr

if_expr  ::= "if" expr ":" NL INDENT block DEDENT elif* else?
         | "if" expr "then" expr "else" expr
match_expr ::= "match" expr ":" NL INDENT match_case+ DEDENT
match_case ::= "|" pattern ("if" expr)? "->" body NL
lambda    ::= "fn" lambda_param* "->" body
list      ::= "[" (expr ("," expr)* ",")? "]"
tuple     ::= "(" expr ("," expr)* ")"

```

Building a `record` is a special form of call syntax in which **field names are required** (`User(name = "Ada", age = 36)`). Functions and ADT constructors are called positionally (naming them gives `TYP0012`).

Patterns

```

pattern ::= "_"                # wildcard
         | "null"              # null
         | IDENT                # name binding
         | literal              # literal
         | "(" pattern ("," pattern)+ ")" # tuple
         | IDENT "(" pattern ("," pattern)* ")" # constructor
         | pattern "|" pattern  # or pattern
         | pattern ":" type     # with a type annotation

```

The pattern of a `let` must be **irrefutable**. `let Some(x) = ...` gives `TYP0008` (use `match`).

A.4 Types

Basic types

```

Bool  Int  Double  String  Unit  Any  Nothing  Null

```

`Int` is arbitrary precision. `Nothing` is the type with no values (the result type of `crash()`).

That is all of the basic types. `Long`, `Float` and `Char`, which other languages might lead you to expect, do not exist in v0.1. There is one integer type, `Int` (arbitrary precision, so there is no reason to separate a `Long`), one fractional type, `Double`, and characters are `String`. Writing one in a type annotation is treated exactly as a misspelled type name, giving `TYP0003`.

Compound types

```
Option[Int]      Result[Int, String]  List[Int]
Lazy[Int]        IO[String]           Tuple[Int, String]
String?
```

`T?` is `Nullable[T]` in the AST and in type representations.

Function types

The curried notation is canonical, and `->` is **right-associative**.

```
Int -> Int
Int -> Int -> Int
```

`(Int, Int) -> Int` is treated as sugar for `Int -> Int -> Int`. A function taking one tuple is written `Tuple[Int, Int] -> Int`.

Null safety

Both `null` and a non-null `T` can be passed where a `T?` is wanted, but **`null` cannot be passed where a non-null type is wanted, and a `T?` cannot be used as-is where a `T` is required**. The ways to unwrap:

Means	Written as
a null pattern in <code>match</code>	<code> null -> ...</code> ; once covered, later names narrow to non-null
null coalescing	<code>a ?? b</code> (<code>b</code> if <code>a</code> is null; short-circuits)
safe navigation	<code>x?.m</code> (short-circuits to null; the result is nullable)
flow narrowing	in the branch of <code>if x != null:</code> , <code>x</code> is a <code>T</code>

Narrowing applies only to the simple `if x != null` / `if x == null` where `x` is a variable. Narrowing through compound conditions (`&&` and so on), `elif` or `while`, and a `!!` assertion operator, do not exist yet.

Local type inference

Parameters of a `def` require type annotations. The result type may be omitted (but is **required for a recursive function** — `TYP0011`).

An unannotated lambda parameter takes its type from the context. The contexts are the type annotation of a `let` or `var`, the result annotation of a `def`, and the argument position of a call.

```
let inc: Int -> Int = fn x -> x + 1          # x : Int
let doubled = map([1, 2, 3], fn x -> x * 2) # x : Int
```

Without a context it falls back to `Any` with the warning `TYP0010`.

A.5 Evaluation

Lune is **lazy by default**.

What is lazy

- the right-hand side of an ordinary `let`
- ordinary function arguments
- ordinary constructor fields
- ordinary record fields
- the body of `lazy expr`

Where evaluation happens

- a variable reference
- `force` / `deepForce`
- the first argument of `seq`
- the condition of an `if`, and of a `while`
- the operands a binary operation needs
- the outer constructor of a `match` scrutinee
- comparison against a literal pattern
- strict parameters (`!x: Int` / `strict x: Int`)
- `strict let` (and `!let`, which is the same)
- strict constructor fields and strict record fields

Memoisation

A thunk memoises **both success and failure**. Forcing a thunk that has failed once produces the same error again (the computation is not repeated).

Re-entrant evaluation is a runtime error (`RUN0005`). A value that depends on its own result, such as `let x = x + 1`, comes back to itself the moment it is forced, so it is reported on the spot rather than waited on (§4.8).

Running `--check` on `let x = x + 1` reports `TYP0001` (undefined name `x`) instead, because the `x` on the right is resolved before that `let` enters the environment.

`RUN0005` appears when `--eval` actually forces it — an example of the same code producing different diagnostics at the checking and running stages.

The instruments for observing this are `:thunks` (state without evaluation) and `:trace` / `--trace` (a commentary on forcing). See chapter 4 and appendices B and D.

A.6 Not supported in v0.1

Some things are reserved as keywords but do not work yet. **What comes back is listed alongside**, so that you can look it up from the diagnostic code.

What you wrote	The diagnostic
<code>class</code>	PRS0001 expected top-level declaration, got CLASS
<code>interface</code>	PRS0001 expected top-level declaration, got INTERFACE
<code>new Foo()</code>	TYP0003 unsupported expression: NewExpr
<code>this / super</code>	TYP0003 unsupported expression: ThisExpr
<code>try / catch / finally</code>	PRS0001 expected expression, got TRY
record update <code>r { x = 2 }</code>	PRS0001 (LBRACE)
record pattern <code>R { x }</code>	PRS0002 expected ARROW, got LBRACE
mutable record field <code>var x: Int</code>	PRS0002 expected IDENT, got VAR
annotation <code>@inline</code>	PRS0001 (AT)
<code>throws</code>	PRS0002 expected ASSIGN, got THROWS
<code>import a.b.{C, D}</code>	PRS0002 expected IDENT, got LBRACE
ADT field access <code>c.v</code>	TYP0003 unsupported member access

Not even reserved — `break`, `continue`, `return`, `export` and `use` are not keywords, so they are taken for ordinary identifiers and give TYP0001 (undefined name). To leave a loop early, weave the condition into the loop condition (§9.3).

Outside the language itself, these are not supported: JVM bytecode generation, calling real Java libraries, `Stream / Map / Set / Promise / Iterator`, a package manager, and an LSP server.

What the future is supposed to look like is in `documents/LANGUAGE_FUTURE_SPEC.md`. Why v0.1 was cut at this scope is in appendix E.

Where this appendix comes from

Section	Source
A.1 keywords	generated from <code>KEYWORDS</code> in <code>lune/tokens.py</code> (48 words)
A.1 operator table	§14 of <code>documents/SYNTAX_SPEC.md</code> ; the infix part cross-checked against <code>INFIX</code> in <code>lune/parser.py</code>
A.1 literals	each literal run through the REPL to confirm its type
A.2 layout	§3 of <code>documents/LEXER_PARSER_SPEC.md</code>
A.3 grammar	the skeleton extracted from the EBNF of §5–§10 of the same
A.4 types	§6 and §8 of <code>documents/LANGUAGE_SPEC.md</code>
A.5 evaluation	§17 of the same
A.6 unsupported	§21 of the same. Every entry was run through <code>--check</code> and the diagnostic recorded

Writing it turned up two places where the specification and the implementation disagreed. Both have been resolved.

`LXL0004` (tabs in indentation) was fixed **in the implementation** — the diagnostic existed, but the indentation width was measured by counting leading spaces only, which left it impossible to produce.

The type of a character literal was fixed **in the specification**. `LANGUAGE_SPEC.md` listed `char` as a basic type, but the implementation has neither an expression that produces one nor any function that handles characters. `Char` was removed from v0.1 and became a type of the future specification.

Appendix B: Standard Library Reference

All 41 names in the prelude. No `import` is needed: they are visible from the start, in every file and in the REPL.

To know what the implementation in front of you actually has, the surest thing is to ask the REPL.

```
lune> :env
```

Each entry gives a type signature, a one-line description and an example. Those that work on lists also say **whether they can be used on an infinite list**, which is the distinction that matters most in practice in Lune (chapter 8).

The “infinite” column — `○` means it comes back when applied to an endless list such as `naturalsFrom(1)`; `×` means it tries to reach the end and does not come back. These marks are not guesses: each function was actually applied to an infinite list.

A `[T]` in a type is a type variable (any type at all).

B.1 Option — “there might be a value”

Name	Type	Meaning
<code>Some</code>	<code>[T] T -> Option[T]</code>	build the “there is a value” form
<code>None</code>	<code>Option[T]</code>	the “there is no value” form
<code>isSome</code>	<code>[T] Option[T] -> Bool</code>	is it a <code>Some</code> ?
<code>isNone</code>	<code>[T] Option[T] -> Bool</code>	is it a <code>None</code> ?
<code>getOrElse</code>	<code>[T] Option[T] -> T -> T</code>	take the contents; the second argument if <code>None</code>
<code>optionMap</code>	<code>[T, U] Option[T] -> (T -> U) -> Option[U]</code>	apply a function to the contents only

```
lune> Some(3)
Some(3) : Option[Int]
lune> getOrElse(Some(3), 0)
3 : Int
lune> getOrElse(None, 0)
0 : Int
lune> optionMap(Some(3), fn n: Int -> n * 2)
Some(6) : Option[Int]
```

`option` is chapter 5; choosing between it and `null` is chapter 7.

B.2 Result — “success or failure”

Name	Type	Meaning
<code>Ok</code>	<code>[T, E] T -> Result[T, E]</code>	build the success form
<code>Err</code>	<code>[T, E] E -> Result[T, E]</code>	build the failure form; it can carry a reason
<code>isOk</code>	<code>[T, E] Result[T, E] -> Bool</code>	is it a success?
<code>isErr</code>	<code>[T, E] Result[T, E] -> Bool</code>	is it a failure?
<code>unwrapOr</code>	<code>[T, E] Result[T, E] -> T -> T</code>	the contents on success, the second argument on failure
<code>resultMap</code>	<code>[T, U, E] Result[T, E] -> (T -> U) -> Result[U, E]</code>	apply a function only on success

```
lune> Err("boom")
Err("boom") : Result[T, String]
lune> unwrapOr(Ok(3), 0)
3 : Int
lune> unwrapOr(Err("boom"), 0)
0 : Int
lune> resultMap(Ok(3), fn n: Int -> n * 2)
Ok(6) : Result[Int, E]
```

Unlike `option`, `Err` **can carry the reason for the failure**. Used in chapter 5 and in the ledger of chapter 13.

B.3 List — the basics

Name	Type	Meaning	Infinite
<code>Nil</code>	<code>List[T]</code>	the empty list	—
<code>Cons</code>	<code>[T] T -> List[T] -> List[T]</code>	prepend one element	○
<code>head</code>	<code>[T] List[T] -> Option[T]</code>	the first element; <code>None</code> when empty	○
<code>tail</code>	<code>[T] List[T] -> Option[List[T]]</code>	everything but the first; <code>None</code> when empty	○
<code>isEmpty</code>	<code>[T] List[T] -> Bool</code>	is it empty?	○
<code>length</code>	<code>Any -> Int</code>	the number of elements (works on strings too)	✖
<code>range</code>	<code>Int -> Int -> List[Int]</code>	<code>range(a, b)</code> is a up to but not including b	—
<code>map</code>	<code>[T, U] List[T] -> (T -> U) -> List[U]</code>	apply a function to each element	○
<code>filter</code>	<code>[T] List[T] -> (T -> Bool) -> List[T]</code>	keep the elements satisfying a condition	○
<code>fold</code>	<code>[T, U] List[T] -> U -> (U -> T -> U) -> U</code>	collapse it, from an initial value	✖
<code>take</code>	<code>[T] List[T] -> Int -> List[T]</code>	the first n	○
<code>drop</code>	<code>[T] List[T] -> Int -> List[T]</code>	discard the first n	○
<code>takeWhile</code>	<code>[T] List[T] -> (T -> Bool) -> List[T]</code>	take while the condition holds	○
<code>dropWhile</code>	<code>[T] List[T] -> (T -> Bool) -> List[T]</code>	discard while the condition holds	○
<code>zip</code>	<code>[T, U] List[T] -> List[U] -> List[Tuple[T, U]]</code>	pair two lists; stops with the shorter	○

Name	Type	Meaning	Infinite
zipWith	<code>[T, U, V] List[T] -> List[U] -> (T -> U -> V) -> List[V]</code>	pair them and apply a function	○

```
lune> let xs = [1, 2, 3]
ok
lune> head(xs)
Some(1) : Option[Int]
lune> tail(xs)
Some((2 3)) : Option[List[Int]]
lune> range(1, 5)
(1 2 3 4) : List[Int]
lune> map(xs, fn n: Int -> n * 2)
(2 4 6) : List[Int]
lune> filter(xs, fn n: Int -> n % 2 == 1)
(1 3) : List[Int]
lune> fold(xs, 0, fn a: Int n: Int -> a + n)
6 : Int
lune> zip(xs, ["a", "b"])
((1, "a") (2, "b")) : List[Tuple[Int, String]]
lune> zipWith(xs, xs, fn a: Int b: Int -> a * b)
(1 4 9) : List[Int]
```

Only **length** and **fold** cannot be used on an infinite list, because reaching the end is their job. Everything else is safe: **map** or **filter** over an infinite list computes only as much as is later taken out (chapters 4 and 8). **drop** forces only the cells it discards, so it is safe too — but what it returns is still infinite, so keep a **take** in front of any display.

range shows – in the infinite column (it takes no list), but **the list it returns is lazy**: the spine is built only as far as it is consumed, so `take(range(1, 1000000000), 5)` costs the same as `take(range(1, 11), 5)`. The width itself is free (§4.6).

Out-of-range and empty cases return plain values rather than raising.

```
lune> take([1, 2, 3], 10)
(1 2 3) : List[Int]
lune> drop([1, 2, 3], 10)
() : List[Int]
lune> range(5, 1)
() : List[Int]
lune> head(Nil)
None : Option[T]
```

B.4 List — building endless lists

Name	Type	Meaning
<code>naturalsFrom</code>	<code>Int -> List[Int]</code>	<code>n, n+1, n+2, ...</code>
<code>iterate</code>	<code>[T] (T -> T) -> T -> List[T]</code>	apply a function to a seed over and over
<code>repeat</code>	<code>[T] T -> List[T]</code>	the same value forever
<code>cycle</code>	<code>[T] List[T] -> List[T]</code>	the given list, repeated forever

```
lune> take(naturalsFrom(1), 5)
(1 2 3 4 5) : List[Int]
lune> take(iterate(fn n: Int -> n * 2, 1), 5)
(1 2 4 8 16) : List[Int]
lune> take(repeat("x"), 3)
("x" "x" "x") : List[String]
lune> take(cycle([1, 2]), 5)
(1 2 1 2 1) : List[Int]
```

These four are infinite in themselves, so always cut them with `take` or `takeWhile`. Chapter 8.

B.5 Display and output

Name	Type	Meaning
<code>show</code>	<code>Any -> String</code>	turn a value into a string in Lune's display form
<code>print</code>	<code>Any -> Unit</code>	print it (no newline)
<code>println</code>	<code>Any -> Unit</code>	print it with a newline

```
lune> show(42)
"42" : String
lune> show([1, 2])
"(1 2)" : String
lune> println("hi")
hi
() : Unit
```

`show` only **returns** a string; it puts nothing on the screen. `print` and `println` do that. `show("hi")` is `"\"hi\""` because turning a string into display form adds the

quotes.

B.6 Small tools

Name	Type	Meaning
<code>id</code>	<code>[T] T -> T</code>	return the argument unchanged
<code>const</code>	<code>[T, U] T -> U -> T</code>	return the first argument, ignoring the second
<code>not</code>	<code>Bool -> Bool</code>	negate a boolean

```
lune> id(7)
7 : Int
lune> const(1, 2)
1 : Int
lune> not(true)
false : Bool
```

The second argument of `const` stays lazy and is **never evaluated**. It is a one-line demonstration of chapter 4’s “what is not used is not computed”.

B.7 Built-ins for observation

Tools for watching lazy evaluation with your own eyes. They are **not a stable public API** (`STANDARD_LIBRARY_SPEC.md` §8.1 says so explicitly). They exist for teaching and for tests.

Name	Type	Meaning
<code>crash</code>	<code>() -> Nothing</code>	raise a runtime error when evaluated
<code>tick</code>	<code>() -> Int</code>	increment an internal counter and return it
<code>tickCount</code>	<code>() -> Int</code>	the current counter; does not increment it

`crash` is for demonstrating that nothing happens if it is not evaluated. `tick` counts **how many times something was evaluated**, which is the evidence that `let` is lazy.

```
lune> let t1 = tick()
ok
lune> let t2 = tick()
ok
lune> tickCount()
0 : Int
```

Two bindings of `tick()` and the counter is still `0`. `let` is lazy, so neither has been called. With `strict let` it would be `2`. Chapter 4.

B.8 seq and deepForce are not functions

These two are not prelude functions but **syntax of the language** (of the same family as `lazy` and `force`; they do not appear in `:env`).

```
seq a b          # evaluate a, then return b
deepForce x      # evaluate x all the way down
```

They are the tools for controlling the order and depth of evaluation, covered in §4.7.

Where this appendix comes from

The type signatures were taken from the REPL's `:env` — that is, from what the implementation actually has. The descriptions follow `documents/STANDARD_LIBRARY_SPEC.md`. The “infinite” column is the result of applying each function to `naturalsFrom(1)` and seeing whether it came back.

Appendix C: The Diagnostic Codes

The complete list of diagnostic codes issued by Lune's compiler and evaluator. Here the book confines itself to being a map: the detailed explanation of each code (what happened, the smallest reproduction, how to fix it) is not copied here but left to `lune explain CODE ( :explain CODE` in the REPL). The same content is readable in the [diagnostic catalogue](#) and in `documents/ERROR_INDEX.md` (the Japanese version is `ERROR_INDEX_JA.md`).

```
$ lune explain TYP0007          # read one code
$ lune explain --index         # list the explanations of every code
```

The first three letters of a code are its **family**, corresponding directly to a stage of the compiler. What the families mean and how to read them is §11.2.

You will never meet a code without an explanation.

`test_every_emitted_code_has_an_explanation` in `tests/test_explanations.py` checks that every code the compiler can issue has one, so adding a code without writing its explanation makes the test fail.

How to read the table

- **Chapters that mention it** — the chapters where the code appears in the text. Chapter 11 is about the machinery of diagnostics itself, so it touches nearly all of them.
- **Working example** — chapters where `books/examples-en/` contains a verified example that actually reproduces the code. Use it as a way in when you want to produce one yourself. A `–` means the book has no example aimed at that code (`LAY0002` and `REC0001`, for instance, take some contriving to produce deliberately).
- Codes issued as warnings say “(warning)” in the meaning column. A warning does not make `--check` fail.

The list

Code	Family	Meaning	Chapters that mention it	Working example
LAY0001	layout	inconsistent indentation	11	—
LAY0002	layout	unmatched closing bracket	11	—
LXL0001	lexing	unexpected character	2, 11	2
LXL0002	lexing	unterminated string or character literal	11	—
LXL0003	lexing	unterminated block comment	11	—
LXL0004	lexing	tabs are not allowed in indentation	11	—
MOD0001	module resolution	module not found or unreadable	10, 11	10
MOD0002	module resolution	cyclic module import	10, 11	10
MOD0003	module resolution	module declaration mismatch	10, 11	10
PRS0001	parsing	unexpected token	2, 6, 11	2
PRS0002	parsing	a particular token was required	10, 11	—
REC0001	record checking	duplicate field in a record declaration	11	—
REC0002	record checking	unknown record field	6, 11	6
REC0003	record checking	missing record field	6, 11	—

Code	Family	Meaning	Chapters that mention it	Working example
REC0004	record checking	field given twice in a construction	6, 11	—
REC0005	record checking	a field that was never declared	6, 11	—
REC0006	record checking	records are built with named fields	6, 11, 13	—
RUN0005	run time	recursive thunk evaluation	3, 4, 10, 11, 12	4
RUN0006	run time	runtime error	2, 4, 6, 9, 11, 12	4
TYP0001	type checking	undefined name	1, 4, 10, 11, 12	1, 4, 11
TYP0003	type checking	type mismatch	1, 2, 7, 8, 11, 12	2, 7
TYP0004	type checking	value is not callable	3, 11	—
TYP0005	type checking	wrong number of arguments	1, 3, 11	1
TYP0006	type checking	the subject of a <code>for</code> must be a List	9, 11	9
TYP0007	type checking	non-exhaustive match	5, 7, 11, 13	5, 7, 11
TYP0008	type checking	a refutable pattern in a binding	5, 9, 11	5
TYP0009	type checking	unreachable match case (warning)	5, 7, 11, 12	5
TYP0010	type checking	cannot infer a parameter's type (warning)	3, 11	—
TYP0011	type checking	a recursive function needs a result type	2, 3, 11	3

Code	Family	Meaning	Chapters that mention it	Working example
TYP0012	type checking	named arguments are not supported here	11, 13	—
TYP0013	type checking	assignment to an immutable binding	9, 11	9

About the gaps in the numbering

You may have noticed that the numbers are not consecutive: `TYP0002` and `RUN0001` – `RUN0004` are missing.

The `RUN` codes were, at the design stage, going to divide runtime errors by kind (`RUN0004` was reserved for “non-exhaustive match at run time”). The implementation abandoned that plan, keeping only the forcing of a recursive thunk as its own `RUN0005` and collecting the rest into the general `RUN0006`. An undefined variable, forcing a failed thunk, a type mismatch in the standard library and a non-exhaustive match at run time are all `RUN0006`. `TYP0002` is simply unused. See `documents/ERROR_DIAGNOSTICS_SPEC.md` for details.

The gaps will not be filled in. A number that once meant one thing coming to mean another later would harm readers more than a gap does. An error number is a key for looking things up, not a serial number.

Where this appendix comes from

The code, family and meaning columns of this table are built from `documents/ERROR_INDEX.md`. That index is itself not hand-written but generated from the compiler’s catalogue.

```
$ ./bin/lune explain --index > documents/ERROR_INDEX.md
```

So this table comes from the catalogue in `lune/explanations.py` — the same primary source as the implementation. When a code is added or changed, regenerate

the index and then update this table.

Appendix D: CLI and REPL Command Reference

Every feature of the `lune` command and the REPL. The story of how to use them is in chapter 11 (diagnostics) and chapter 12 (the tools); this is the place to look up “what was that argument again?”.

Following the book’s convention, `lune` means `./bin/lune` in the repository (or the `lune` installed by `pip install lune-lang`).

D.1 Startup and modes

```
$ lune [--lang en|ja] [--module-path DIR]... [MODE] [FILE]
```

The mode is one of the following, and **only one at a time**.

Form	What it does
<code>lune</code>	with no arguments, start the REPL
<code>lune --repl</code>	start the REPL explicitly
<code>lune --check FILE</code>	type-check and stop; do not evaluate
<code>lune --eval NAME FILE</code>	read the file, evaluate the top-level binding <code>NAME</code> and print it
<code>lune --eval NAME --trace FILE</code>	as above, plus a commentary on forcing to <code>stderr</code> (chapter 4)
<code>lune --tokens FILE</code>	print the token stream after layout processing (chapter 12)
<code>lune FILE</code>	with no mode, print the syntax tree (AST) as it is

The last line is the default, for developers. It is not meant to be read by people, so use `--check` or `--eval` in normal use.

`--eval` evaluates without type-checking. Running ill-typed code with `--eval` means the types of operators and conditions are checked at run time, giving `RUN0006` with a hint suggesting `lune --check`. The safe habit is to pass `--check` first.

D.2 Global options

Option	Effect
<code>--lang en ja</code>	the language of diagnostics; <code>--lang=ja</code> also works
<code>--module-path DIR</code>	add a module search root; may be repeated (chapter 10)
<code>--version / -V</code>	print the version and exit
<code>-h / --help</code>	print usage

The relationship for the language is: the environment variable `LUNE_LANG` is the default, and the `--lang` flag wins over it.

```
$ export LUNE_LANG=ja           # Japanese from here on
$ lune --lang en --check f.lune # English, just this once
```

Note — `--lang` does **not** appear in the list printed by `lune --help`, because it is handled before the other options. Do not forget it exists.

The version is worth quoting in a bug report.

```
$ lune --version
lune 0.1.0
```

The line the REPL prints at startup carries the same number (`Lune v0.1.0 REPL.`). This book covers the v0.1 series.

D.3 Subcommands

`lune explain`

```
$ lune explain CODE [--lang en|ja]
$ lune explain --index [--lang en|ja]
```

Prints the explanation of a diagnostic code. `--index` prints the explanations of every code, which is exactly what `documents/ERROR_INDEX.md` is (appendix C).

Give it a code it does not know and it tells you which codes it does.

lune fmt / lune fix

```
$ lune fmt [--write|--check] FILE...  
$ lune fix [--write|--check] FILE...
```

`fmt` formats; `fix` applies the mechanical fixes suggested by diagnostics (chapters 11 and 12). Their flags work the same way.

Flag	Behaviour
none	write the result to standard output; there must be exactly one file
<code>--write / -w</code>	rewrite the files in place; several files allowed
<code>--check</code>	change nothing; exit 1 if a change is needed. For CI

`--write` and `--check` cannot be combined.

`fmt` has one limitation: a file containing `###` block comments cannot be formatted (§12.3).

D.4 REPL commands

`:help` prints the complete list.

Command	What it does
<code>:help</code>	print this list
<code>:quit / :q</code>	quit
<code>:env</code>	list every visible name with its type
<code>:type NAME</code>	give the type of a name
<code>:thunks [NAME]</code>	show a binding's evaluation state without evaluating it (chapter 4)
<code>:trace [on off]</code>	toggle the running commentary on forcing (chapter 4)
<code>:lang [en ja]</code>	switch the diagnostic language
<code>:explain CODE [en ja]</code>	read the explanation of a diagnostic code (chapter 11)

With no argument, `:thunks` lists the state of every binding.

Multi-line input continues while a line ends in `=`, `:` or `->`, and finishes at a blank line. Started in a terminal, line editing and history (the up and down arrows) work, and the history is kept in `~/.lune_history`.

`import` works here too (chapter 10). The search starts from **the directory you started in**, and `--module-path` adds to it. The names of an imported module stay in the session, so they appear in `:env`.

```
$ lune --repl
lune> import math
ok
lune> add(1, 2)
3 : Int
```

D.5 Exit codes

For calling it from a script or from CI.

Code	Meaning
0	success. Warnings alone are still 0 — <code>--check</code> does not fail on a warning
1	a check or an evaluation failed, and a diagnostic was printed. Also “would change” from <code>fmt --check / fix --check</code>
2	misuse: an unknown flag, a missing argument, <code>--write</code> and <code>--check</code> together

The point is that “a diagnostic was reported” (1) and “the command was called wrongly” (2) are separate. When `--check` fails, you can tell whether the problem is in the code or in your script without reading the output.

Appendix E: The Design of Lune, and What Comes Next

The body of this book has been about how to write. This appendix is about why things were decided that way. A language design usually carries **inconveniences it chose to accept**; leaving those out in the open is this book's last piece of work.

E.1 Why lazy by default

“Lazy evaluation is a trap for beginners” is a fair criticism. When you cannot see when a computation runs, you cannot read the behaviour of a program, and the order of side effects defies intuition. It is no accident that the teaching languages — Racket, Pyret, Hedy — are all strict.

Lune chose lazy by default anyway, and **has decided not to reverse it**. The reason is not that laziness is convenient.

It is that the answer to the criticism was not to hide it but to **make it visible**.

	The hiding road	The road Lune took
the timing of evaluation	invisible, so not thought about	see the state with <code>:thunks</code> , the moment with <code>:trace</code>
a circular definition	it hangs (Haskell's <code><<loop>></code>)	<code>RUN0005</code> , reported immediately and with an explanation
when you want strictness	the language decides for you	you decide, with <code>strict</code> / <code>!</code> / <code>seq</code> / <code>force</code> / <code>deepForce</code>

Chapter 4 is that answer in full. “You do not have to take any of it on faith; all of it can be observed on the spot” was written there because **observability is not the price of the design, it is the design**.

There is a second aim in this. Teaching “evaluation strategy” in a strict language is hard, because the subject sits outside the program. In a lazy-by-default language, the evaluation strategy becomes **something observable inside the program**. The moment you call `tick()` twice and see the counter still at 0 (appendix B), laziness stops being a concept and becomes a phenomenon.

The design tension — “kind to beginners” and “able to teach evaluation strategy” do not usually go together. Lune is a language trying to resolve that tension with diagnostics and observability. Whether it succeeds is answered by how you felt at the end of chapter 4. The strategy in detail is measures D and E in `documents/STRATEGY.md`.

E.2 Why there is both `null` and `Option`

Chapter 5 taught `Option[T]` and chapter 7 taught `T?`. Two tools for “there might not be a value” looks like fat in a design.

It looks that way, and yet neither could be removed.

`Option` cannot be removed. It is not a language feature but **a type anyone can write** given ADTs and pattern matching. It is in fact defined in the prelude as nothing more than `type Option[T] = | Some(value: T) | None`, with no special treatment from the compiler. You cannot take `Option` away from a language that has ADTs — short of forbidding it, somebody will define it.

`T?` cannot be removed either. Lune’s long-term goals include running on the JVM and calling Java libraries directly (`documents/LANGUAGE_FUTURE_SPEC.md`). And every Java reference can be null. In the future specification’s table of Java type correspondences, `T?` is the type matching a **nullable reference** — in other words, **the type for talking to the outside world**. A language without null cannot talk to a world that has it.

So Lune chose **not to abolish `null` but to tame it with types** (chapter 7). Not “a language that removed null” but “a language that makes you write where null may appear, and will not let you use it before checking”.

The roles then divide naturally.

	<code>T?</code>	<code>Option[T]</code>
origin	a language feature (with <code>??</code> , <code>?.</code> and narrowing)	an ADT in the prelude
suits	the shape of data: record fields, parameters, results	list processing and higher-order pipelines
at the boundary	corresponds to a Java nullable reference	an internal matter for Lune

The practical rule of thumb is in §7.7. At the boundary, a `match` converts between them.

To be honest — the price of having two is exactly one thing: the reader has to learn which to use. That is a real cost. The table in §7.7 exists to keep that cost down to “look at a table”.

E.3 The limits v0.1 accepted

What was not built is as much a design as what was. The list of what does not work, and the diagnostics you get, is in A.6. Here is the **why**.

Frozen — the JVM and OO

`class` / `interface` / `extends`, calling Java libraries directly, JVM bytecode generation. They are in the future specification (Phases 2–4 of `LANGUAGE_FUTURE_SPEC.md`), but the decision to **freeze them for now** is explicit (`documents/STRATEGY.md` §5).

The reason is a choice of battlefield, not of capability. Java interoperation is a measure aimed at “get adopted in industry”, and that is a fight with poor odds. For teaching, meanwhile, the speed of a Python interpreter is enough. So Lune stepped off the road to becoming a production language and concentrated on being **an introductory functional language whose compiler teaches you in your own tongue**.

That `class` and `new` are syntax errors while still being reserved keywords — that half-way state is the mark of this freeze. The seats are kept for the future; nobody is sitting in them yet.

Not there yet — try / catch

There is no tool for throwing failure as an exception. This is a matter of **order** rather than of freezing. The tools that **return failure as a value** — `Result` in chapter 5 and `T?` in chapter 7 — were put in place first, and exceptions overlap with them. Chapter 9’s “express recoverable failure as a value; `raise` is for when stopping the program is correct” is the consequence of that design.

Small things missing

`break` / `continue` (weave it into the loop condition — §9.3), record update and record patterns, mutable record fields, field access on ADTs, `Stream` / `Map` / `Set` . There is no character type either (A.1).

Some of these are not philosophy but simply **not got round to yet**. A.6 lists what happens if you try them; `LANGUAGE_FUTURE_SPEC.md` lists what is wanted.

E.4 If you want to read the implementation

Lune's implementation is pure Python under `Lune/`, about 7,500 lines, with **no external dependencies**. It can be read in a day, which makes it a good subject for a first reading of a language implementation.

The sources are laid out in the order of the checks. The families of diagnostic codes from A.1 (`LXL` → `LAY` → `PRS` → `MOD` → `TYP` → `RUN`) are also the order of the files.

File	Lines	Role
<code>lexer.py</code>	294	text → tokens. <code>LXL</code>
<code>layout.py</code>	61	indentation → <code>INDENT / DEDENT . LAY</code>
<code>tokens.py</code>	214	token kinds and the keyword table
<code>parser.py</code>	763	tokens → AST (a Pratt parser). <code>PRS</code>
<code>nodes.py</code>	405	the AST definitions; read this for the shape of the whole language
<code>module_loader.py</code>	196	resolving <code>import . MOD</code>
<code>typechecker.py</code>	1597	type checking and exhaustiveness. <code>TYP , REC</code> . The largest file
<code>evaluator.py</code>	1346	lazy evaluation and thunks. <code>RUN</code>
<code>diagnostics.py</code>	124	the display format of diagnostics (assembling carets and hints)
<code>messages.py</code>	358	the English/Japanese message catalogue
<code>explanations.py / explanations_ja.py</code>	532 / 430	the teaching catalogue behind <code>lune explain</code>
<code>formatter.py / fixer.py</code>	490 / 66	<code>lune fmt / lune fix</code>
<code>repl.py</code>	403	the REPL. The Playground drives this same <code>ReplSession</code>
<code>cli.py</code>	279	the command-line entry point

A suggested reading order: `nodes.py` → `lexer.py` → `parser.py` → `evaluator.py`. Look at the AST first and everything else becomes a matter of “how it is built” and “how it is handled”. If lazy evaluation is what interests you, the shortest way in is `Thunk` and `force_value` in `evaluator.py`.

How to read the specifications

The specifications are in `documents/`. **`LANGUAGE_SPEC.md` is authoritative for v0.1**, with the detailed documents branching off it.

What you want to know	What to read
what is usable in v0.1	<code>LANGUAGE_SPEC.md</code> (authoritative)
surface syntax, EBNF	<code>SYNTAX_SPEC.md</code> / <code>LEXER_PARSER_SPEC.md</code>
the rules of lazy evaluation	<code>LAZY_EVALUATION_SPEC.md</code>
the scope of type checking	<code>TYPE_CHECKER_SPEC.md</code> / <code>MATCH_EXHAUSTIVENESS_SPEC.md</code>
the design of diagnostics	<code>ERROR_DIAGNOSTICS_SPEC.md</code> / <code>ERROR_INDEX.md</code>
where it is going	<code>LANGUAGE_FUTURE_SPEC.md</code> / <code>STRATEGY.md</code>

A caution when reading the specifications — they are called authoritative, but there are places where the v0.1 implementation has not caught up, and places where the implementation is the one that is right. Two were found while writing appendix A alone (the case where `LXL0004` could never fire, and the character type), and a third while translating chapter 8 (a “known bug” that had already been fixed). **When in doubt, run it through `./bin/lune`.** That is the primary source. Every appendix in this book was written that way.

E.5 What comes next

`documents/STRATEGY.md` describes where this is going. In summary: do not chase general adoption, concentrate on being an introductory functional language whose compiler teaches you in your own tongue, and keep the JVM/OO road frozen. Lazy by default is not a weakness to be hidden but the distinguishing feature to lead with.

Having read this book, there are a few things you can do.

- **Tell us where you got stuck.** Which diagnostic held you up, which explanation was not enough — those are directly the language’s improvements. If a diagnostic was unhelpful, we consider that a bug in the language.
- **Add diagnostics.** The runtime errors collected into the general `RUN0006` have room to be promoted to codes of their own (a work item of measure E in `STRATEGY.md`).
- **Write something.** Even v0.1 can write the programs of chapter 13. When you notice a tool that is missing, that is the next thing to build.

The club motto, once more. **Don't do it until you need to. Once done, never forget it.** It is about lazy evaluation and memoisation — and also about how to build a language.

Index

The main terms appearing in the chapters and appendices. The page numbers are those of the PDF edition (in HTML, use the search above).

Values and types

- Int (arbitrary precision) ... 19, 27, 141, 146
- Double ... 12, 17, 18, 19, 20, 21, 25, 27...
- String ... 8, 9, 10, 16, 19, 20, 21, 22
- Bool ... 5, 14, 19, 21, 22, 23, 27, 29...
- Unit ... 10, 19, 20, 27, 97, 98, 99, 100...
- Nothing ... 40, 45, 51, 132, 145, 146, 156
- tuple ... 12, 15, 22, 25, 28, 39, 55, 56...
- type annotation ... 24, 31, 32, 35, 145, 146, 147
- local type inference ... 61, 62, 63, 66, 67, 79, 81, 82...
- type variable ... 60, 64, 70, 151

Lazy evaluation

- lazy evaluation ... 6, 12, 15, 22, 40, 46, 49, 50...
- thunk ... 6, 9, 16, 33, 34, 40, 41, 42...
- memoisation ... 51, 148, 174
- strictness ... 47, 51, 52, 53, 54, 102, 148, 157
- force ... 41, 42, 43, 46, 47, 49, 50, 51...
- deepForce ... 47, 48, 49, 52, 135, 137, 139, 140...
- seq ... 47, 48, 51, 52, 135, 139, 157, 168
- infinite list ... 15, 54, 87, 88, 89, 90, 92, 93...
- stream ... 6, 125, 163

Functions

- partial application ... 16, 18, 31, 33, 34, 36, 39, 127...
- currying ... 146
- higher-order function ... 32, 80, 169
- lambda ... 11, 15, 25, 31, 33, 36, 37, 39...
- pipeline ... 15, 22, 34, 35, 36, 37, 73, 80...
- recursion ... 25, 30, 35, 36, 38, 49, 50, 91...

Data and branching

- algebraic data type ... 6, 15, 22, 55, 56, 57, 60, 63...

- pattern matching ... 56, 169
- exhaustiveness ... 58, 59, 63, 67, 76, 77, 85, 114...
- irrefutable pattern ... 59, 60, 66, 98, 100, 112, 113, 145...
- record ... 6, 15, 25, 56, 68, 69, 70, 71...
- Option ... 58, 60, 61, 62, 63, 65, 66, 67...
- Result ... 8, 61, 62, 63, 64, 65, 67, 100...

Null safety

- null safety ... 6, 15, 20, 75, 84
- narrowing ... 75, 76, 77, 78, 80, 82, 84, 136...
- null coalescing ... 22, 75, 77, 78, 79, 80, 82, 83...
- safe navigation ... 75, 77, 78, 80, 82, 83, 142, 147...

Operators

- floor division ... 20, 21, 27, 96, 134, 137, 138, 142
- compound assignment ... 96
- short-circuiting ... 22, 27, 44, 51, 53, 78, 82, 147

Imperative features

- var ... 95, 96, 97, 98, 100, 101, 102, 103...
- while ... 18, 54, 66, 84, 88, 89, 93, 95...
- for ... 97, 114, 169
- IO ... 95, 99, 100, 102, 103
- modules ... 104, 107

Tools and diagnostics

- diagnostic code ... 5, 13, 16, 50, 112, 115, 120, 121...
- explain ... 4, 5, 13, 14, 16, 18, 21, 22
- fmt ... 24, 105, 121, 123, 124, 125, 126, 127...
- fix ... 14, 16, 115, 116, 118, 119, 125, 126...
- REPL ... 5, 6, 7, 8, 9, 10, 14, 16...
- :thunks ... 6, 9, 16, 40, 41, 47, 48, 49...
- :trace ... 6, 40, 41, 42, 51, 52, 54, 121...
- Playground ... 6, 172

Appendix B (the standard library), appendix C (the diagnostic codes) and appendix D (the CLI and the REPL) are themselves usable as indexes.