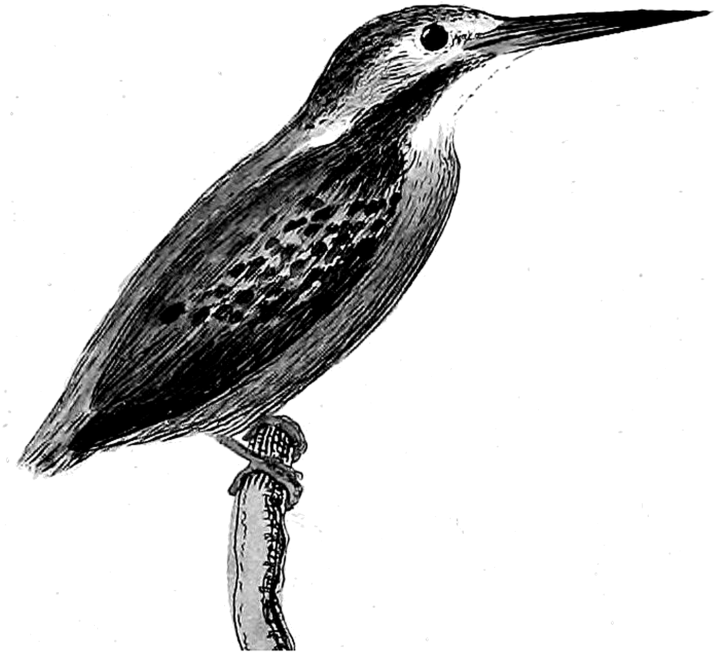


Functional Programming with Rill

Writing programs out of values instead of commands

BARIŞ AKIN

Version 0.2 · every example in this book was compiled and run

Cover: common kingfisher, from a copper plate by Heinrich von Kittlitz, 1832

Contents

Part I

Thinking Functionally

Chapter 1

What Functional Programming Is

There is a version of this chapter that defines functional programming in one sentence — *programming with pure functions over immutable data* — and then moves on. It is accurate and it teaches nobody anything, because every word in it is doing work you cannot see yet.

So we are going to take the long way. By the end of this chapter you will know what the definition means, which parts of it are essential and which are fashion, what the style costs, and why a language would give up assignment — the single most familiar operation in programming — to get it.

You will not write any Rill until Chapter 3. That is deliberate. Every idea in this chapter predates Rill by decades and outlives any particular language, and a reader who understands them will pick up the syntax in an afternoon.

1.1 Two ways to say the same thing

Here is a program that adds up the numbers from 1 to 10, skipping the odd ones. In Python, written the way most people write Python:

```
total = 0
for i in range(1, 11):
    if i % 2 == 0:
        total = total + i
print(total)                        # 30
```

And here is the same calculation, written as an expression:

```
print(sum(i for i in range(1, 11) if i % 2 == 0))    # 30
```

Both produce 30. The difference is not brevity — you could make the first one shorter and the second one longer. The difference is what each one *is*.

The first program is a **sequence of commands**. It says: make a box called `total` and put 0 in it. Make a box called `i`. Repeat the following, changing what is in `i` each time: look in `total`, add something to what you found, and put the result back in `total`. When you are done, look in `total` and print it.

The second program is a **description of a value**. It says: 30 is the sum of the even numbers between 1 and 10. There are no boxes. Nothing is put anywhere. There is no "when you are done", because there is no *doing* — there is a thing that the expression is equal to, and the machine's job is to work out what.

This is the whole idea. Everything else in this chapter is a consequence.

> **The word "functional" is a poor name**
>
> It suggests the subject is functions, and every language has functions. The subject is really *expressions*: building programs out of things that have values rather than out of things that have effects. "Value-oriented programming" would have been a better name, and it is too late.

1.2 The trouble with boxes

Assignment looks harmless. `total = total + i` is the first line of code most people ever write, and it does exactly what it appears to do.

The trouble is not what it does. The trouble is what it *prevents you from knowing*.

Consider this fragment, in any language you like:

```
def report(order):
    tax = compute_tax(order)
    total = order.subtotal + tax
    return format_money(total)
```

Read it and answer a question: **what is `order.subtotal` when line 3 runs?**

You cannot answer. `compute_tax` was handed the order, and for all you know it adjusted the subtotal on the way past. To find out you must open `compute_tax`, and then everything `compute_tax` calls, and so on until you reach the bottom or run out of patience. The three lines in front of you are not enough. In a large system they are never enough, and this is the practical reason that reading unfamiliar code is so much harder than writing it.

Now suppose the language did not permit `compute_tax` to change anything. Not by convention, not by code review — by construction. Then the question has an answer, and the answer is visible: `order.subtotal` is whatever it was on line 2, because there is nowhere else it could have come from.

That is the trade. You give up assignment, and in exchange **the meaning of an expression stops depending on the history of the program**. What a name refers to is decided where the name is introduced, and stays decided.

1.2.1 Aliasing, the sharp edge

The subtler version of the same problem is aliasing — two names for one piece of memory. Here is a bug that every working programmer has shipped at least once:

```
def add_tag(tags, tag):
    tags.append(tag)
    return tags

defaults = ["draft"]
a = add_tag(defaults, "urgent")
b = add_tag(defaults, "internal")

print(a)          # ['draft', 'urgent', 'internal']
print(b)          # ['draft', 'urgent', 'internal']
print(defaults)   # ['draft', 'urgent', 'internal']
```

Three names, one list. The function looked like it was returning a new value and was in fact editing a shared one. Nothing here is exotic; this is Tuesday.

With immutable data the bug is not fixed — it is *unspellable*. `add_tag` cannot append to `tags`, because there is no operation that changes a list. The only thing it can do is produce a new list with one more element, and then `a`, `b` and `defaults` are necessarily three different values.

> **"Immutable means copying everything, so it must be slow"**
>
> This is the objection everyone raises, and it is wrong for a specific reason: values that cannot change can be **shared** without risk. Adding an element to the front of an immutable list allocates one cell that points at the old list; the old list is not copied, because nothing can ever modify it and be noticed. A tree with one node replaced shares every untouched subtree. The technical name is *persistent data structures*, and the cost is usually a small constant factor, not the disaster the objection imagines.
>
> Chapter 15 has the measurements for Rill specifically, including the cases where the constant factor does bite.

1.3 Purity and referential transparency

A function is **pure** when two things hold:

1. Its result depends only on its arguments.

2. Calling it does nothing else — no writing to disk, no mutating a parameter, no incrementing a counter somewhere.

`math.sqrt` is pure. `random.random()` is not (it depends on hidden state). `print` is not (its whole purpose is the effect). `list.append` is not.

The property that purity buys is called **referential transparency**: an expression can be replaced by its value, anywhere it appears, without changing what the program means. If `f(3)` is 9 once, it is 9 always, so:

```
f(3) + f(3)      ≡      2 * f(3)      ≡      let x = f(3) in x + x
```

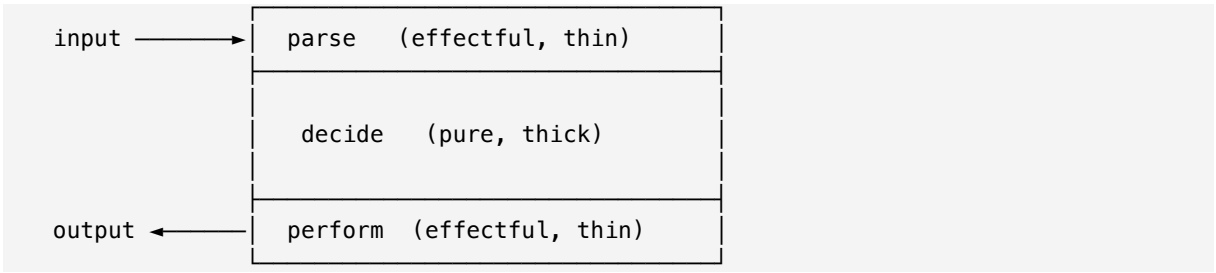
These are all the same program. That sounds like a triviality until you notice how much depends on it:

- **You can reason locally.** Understanding an expression requires reading the expression, not the program's history.
- **The compiler can rewrite freely.** Common subexpressions can be shared, loop-invariant work hoisted out, dead calls deleted. A compiler for an impure language must first prove that a call has no effects, and usually cannot.
- **Tests are cheap and stable.** A pure function needs no fixture, no database, no clock, no mocks. Give it inputs, compare outputs. There is no setup because there is no state to set up.
- **Caching is free.** If a result depends only on arguments, remembering it is always safe.
- **Concurrency stops being frightening.** Two threads that only read immutable data cannot race. There is nothing to lock, because a data race requires a write.

That last point deserves emphasis, because it is the reason the industry became interested in this style around 2005 and not before. When processors stopped getting faster and started getting more numerous, "shared mutable state" changed from a manageable nuisance into the central difficulty of the field. Functional programming had spent forty years accidentally preparing for exactly that.

1.3.1 Nobody is pure all the way down

A program with no effects computes something and then cannot tell you what. Every real program prints, writes files, answers sockets. The functional discipline is not *avoiding* effects, it is **pushing them to the edges**:



The middle does the thinking and can be tested by calling it. The edges do the talking and are kept small enough to eyeball. Languages differ enormously in how much ceremony they demand at the boundary — Haskell tracks it in the type system with `IO`, ML and Rill simply let you call `println` — but the architectural advice is identical, and it is good advice in Java too.

1.4 Functions as values

In most languages, functions and data live in separate worlds. You can put a number in a list; you cannot as easily put "adding" in a list.

In a functional language, a function is an ordinary value. It can be stored, passed, returned, and built at run time. A function that takes or returns a function is called **higher-order**, and three of them replace most of what loops are used for:

You want to	Loop version	Higher-order version
Do something to each element	`for x in xs: out.append(f(x))`	`map(xs, f)`
Keep some elements	`for x in xs: if p(x): out.append(x)`	`filter(xs, p)`
Combine into one value	`acc = 0; for x in xs: acc = acc + x`	`fold(xs, 0, add)`

The gain is not that `map` is shorter. It is that `map` **says what is happening in its name**. When you read a `for` loop you must read the whole body to discover whether it is a map, a filter, a fold, a search, or three of those tangled together. When you read `map(xs, f)` you already know the shape of the answer: same length, same order, each element transformed. You only need to read `f` to know the details.

This is the real argument for the style, and it is an argument about reading:

> **A loop tells you the mechanism and hides the intent. A combinator tells you the intent and hides the mechanism.**

Pipelines compose these into a description of a calculation that reads in the order it happens:

```
range(1, 100)
  |> filter(is_prime)
  |> map(square)
  |> sum
```

Four steps, each one a noun-phrase you could say out loud. The imperative version is a single loop with four things happening inside it, and separating them again is work the reader has to do.

1.4.1 Closures

A function value that captures variables from where it was written is a **closure**. This is what makes higher-order functions useful rather than merely possible:

```
def multiplier(k):
    return lambda x: x * k      # captures k

triple = multiplier(3)
triple(5)                       # 15
```

`triple` is a function that carries a 3 around with it. Closures are how you build behaviour at run time — a comparison function that knows which field to sort on, a callback that knows which connection it belongs to, a parser that knows which keyword it is looking for.

1.5 Recursion, and why loops go away

If nothing can be assigned, a `while` loop cannot exist: its counter would have to change. So how does anything repeat?

By calling itself. Here is a countdown, written as a loop and as recursion:

```
# loop                          # recursion
i = 10                          def go(i):
while i > 0:                        if i == 0: return
    print(i)                        print(i)
    i = i - 1                       go(i - 1)
```

The loop changes `i` in place. The recursion never changes anything: each call gets its own `i`, one smaller than the last. Same countdown, no assignment.

Newcomers reasonably object that recursion is slower and blows the stack. For a *general* recursion that is true. But the shape above is special:

```
    go(i - 1)          ← the last thing the function does
```

Nothing happens after the recursive call returns. There is therefore nothing in the current stack frame worth keeping, and a compiler can reuse it instead of pushing a new one. This is **tail-call elimination**, and it turns that recursion into a machine-code loop — same instructions, same speed, constant stack.

The rule to learn is simple: *if the recursive call is the last thing that happens, it is a loop*. The accumulator pattern is how you get there:

```
sum(list)                          # not a tail call: the + happens after
  = head + sum(tail)

sum(list, acc)                     # tail call: nothing happens after
  = sum(tail, acc + head)
```

The second version carries the running answer forward as an argument. That argument *is* the loop variable, and every functional programmer learns to see it that way.

Where tail calls are guaranteed and where they are not

Scheme, ML, OCaml, Haskell, Erlang, Elixir and Rill guarantee it — it is part of the language, not an optimization you hope for. Java, Python and JavaScript do not, which is why recursion carries a stigma among programmers who learned there. C compilers usually do it at `-O2` but promise nothing.

Chapter 7 shows how to check that Rill really has done it, rather than taking anyone's word.

1.6 Types that describe the problem

The second half of functional programming — and the half that changed most between 1975 and today — is the type system.

Most working programmers have met types as a tax: a thing you write down to keep the compiler quiet. In the ML tradition types are the opposite. They are the design notation, and the compiler is a machine that checks your design against your code.

1.6.1 Sum types: the missing half of the toolkit

Every language has **product types** — a struct, a record, a class, a tuple. "A point has an x *and* a y."

Far fewer have **sum types** — "a shape is a circle *or* a rectangle *or* a triangle." Where they are missing, people simulate them: a class hierarchy, a tagged struct, a dictionary with a `"kind"` key, an enum plus a union you must remember to read correctly.

```
type Shape
  Circle(radius)
  Rect(width, height)
  Triangle(base, height)
```

Three things follow, and they are worth more than the syntax suggests.

One: illegal states become unrepresentable. The classic example is a network request:

```
class Response {
    int status;
    String body;      // null unless status == 200
    String error;     // null unless status != 200
}
```

Four fields, and only two of the sixteen combinations are meaningful. Every reader must remember the rule; every writer must maintain it; and nothing enforces it.

```
type Response
  Ok(body)
  Failed(status, error)
```

Now the meaningless combinations cannot be constructed. Not "are discouraged" — cannot be typed. A whole category of bug stops being a thing you test for and starts being a thing you cannot write.

Two: `null` becomes unnecessary. Tony Hoare called null references his billion-dollar mistake, and the fix is a sum type:

```
type Option(a)
  None
  Some(a)
```

`Option(Int)` is either nothing or an integer, and the difference is in the type. A function returning `Option(User)` announces in its signature that the user may be absent, and the compiler will not let you skip the case. There is no `NullPointerException` because there is no null — absence is a value with a name.

The same shape handles failure without exceptions:

```
type Result(a, e)
  Ok(a)
  Err(e)
```

A function that can fail returns one of these. The caller cannot use the success value without saying what happens on failure, because the two live in different branches. Errors travel as data, along the same paths as everything else, and no control flow jumps out from under you.

Three: the compiler can check that you covered every case. This is **exhaustiveness checking**, and it is the feature that converts more imperative programmers than any other. Add a fourth shape to the type, compile, and the compiler lists every place that now has a hole. Not a runtime `default:` that logs and continues — a compile error, before the program runs.

Rill does this, and the message names what is missing:

```
$ rill run colour.rill
colour.rill: type error at line 6: match is not exhaustive; no arm matches `Blue`
```

That is a real message from the compiler this book is about, and it is the single most useful sentence it can say to you.

1.6.2 Pattern matching

Sum types would be tedious without a way to take them apart. Pattern matching is that way: it tests the shape and binds the pieces in one step.

```
fn area(s) = match s
  Circle(r)    -> 3.14159 * r * r
  Rect(w, h)   -> w * h
  Triangle(b, h) -> b * h / 2.0
```

Compare the object-oriented arrangement, where each shape carries its own `area` method. Both are reasonable, and the choice between them is a real design decision rather than a matter of taste:

	Add a new **shape**	Add a new **operation**
Methods on classes	easy — one new class	hard — edit every class
Match on a sum type	hard — edit every match (but the compiler lists them)	easy — one new function

This is the *expression problem*, and neither column wins. Sum types suit domains where the cases are known and the operations keep coming: compilers, protocols, state machines, anything with a fixed

vocabulary. Classes suit domains where new kinds of thing keep arriving. Knowing which you are in is most of the skill.

1.6.3 Inference: types without typing

The last piece is that you rarely write any of this down. Hindley–Milner inference, which ML introduced in 1973 and every language in this family has kept, works out the most general type of an expression from its use.

```
fn twice(f, x) = f(f(x))
```

The compiler reasons: `f` is applied to `x`, so `f` takes `x`'s type; the result is fed to `f` again, so `f` returns that same type. Therefore `twice : (a -> a, a) -> a`, for any `a`. Nobody wrote a type and the function is fully checked, at every type it is ever used at.

This is why "statically typed" and "verbose" are not synonyms, and why the comparison people usually make — Java's ceremony against Python's freedom — is a false choice. There is a third option, and it has been around for fifty years.

1.7 A short history, because it explains the landscape

1958 — Lisp. John McCarthy builds a language on expressions, recursion and lists. Programs are data; garbage collection is invented because the style demands it. Everything after this is a refinement of ideas Lisp had.

1973 — ML. Robin Milner needs a language for writing proof tactics and builds one with type inference, sum types, pattern matching and guaranteed tail calls. Nearly every idea in Section 1.6 dates from here.

1978 — Backus's Turing Award lecture. John Backus, who led the FORTRAN team, uses his acceptance speech to argue that assignment is the bottleneck of programming — that the "von Neumann style" makes programs hard to reason about and hard to run in parallel. The field mostly ignores him for twenty-five years.

1990 — Haskell. A committee unifies the lazy functional languages. By enforcing purity absolutely, Haskell discovers what a program looks like when effects are tracked in the type system, and produces a generation of ideas — monads, type classes, QuickCheck — that spread far beyond it.

1996–2005 — the industrial ML branch. OCaml, and later F#, keep ML's types and drop its academic reputation. Erlang, built at Ericsson for telephone switches, arrives at immutability and lightweight processes for entirely practical reasons: hardware fails and calls must not drop.

2005 onwards — absorption. Processors stop getting faster. Multi-core becomes the only way forward, shared mutable state becomes the central difficulty, and the industry goes looking for ideas. Between 2008 and 2020 the mainstream acquires lambdas and closures (Java 8, C++11, C# 3), map/filter/fold in every standard library, sum types and pattern matching (Rust, Swift, Kotlin, Scala, and eventually Java 17), immutability by default (Rust, and every Java style guide since), and `Option`/`Result` instead of null and exceptions (Rust, Swift, and a hundred libraries elsewhere).

The interesting thing about that list is what is *not* on it. The mainstream took the types, the values and the combinators. It did not take laziness, and it did not take effect tracking. Those turned out to be the parts of Haskell that most programmers were unwilling to pay for — which tells you something about which ideas here are load-bearing.

1.8 What it costs

An honest chapter has to include this section.

Some algorithms genuinely want mutation. In-place quicksort, union-find with path compression, a hash table with open addressing, a dynamic-programming table filled in place — these have functional equivalents, and the equivalents are often a logarithmic factor slower or a good deal more intricate. A pure array update is O(log n) with a persistent tree, not O(1). This is a real cost and it is unevenly distributed: it barely touches business logic and it lands hard on numerical kernels.

Performance is harder to predict. Allocation happens where an imperative programmer sees none. Lazy languages add a second layer of unpredictability — Haskell's classic beginner disaster is a space leak from thunks piling up in an accumulator. Strict functional languages, Rill among them, avoid that particular trap by evaluating when they are told to.

The learning curve is real, and it is in an unusual place. The syntax is small; you will have it in a day. What takes weeks is the *rearrangement*: learning to reach for a fold instead of a loop, to thread state through arguments instead of storing it, to make illegal states unrepresentable instead of validating at the door. Programmers coming from imperative languages routinely describe six weeks of feeling slow, then a step change.

The ecosystems are smaller. For any functional language you will find fewer libraries, fewer answers, fewer people who have hit your bug. This matters more than any language feature on a real project and it is the reason most teams sensibly stay where they are.

Some of it is genuinely awkward. Threading state through twenty call frames is worse than a global. Building a graph with cycles in an immutable language is a puzzle. Interacting with a fundamentally stateful world — a GPU, a socket, a file handle — requires a bridge, and the bridge is never as pretty as the core.

The people who get the most from this style are the ones who take the ideas and skip the fundamentalism: pure functions where purity is free, immutability by default and mutation where it pays for itself, sum types everywhere, effects at the edges. That is the position this book argues from, and it is the position Rill's design takes.

1.9 What to carry into Chapter 2

Six ideas. The rest of the book is these ideas made concrete:

1. **Expressions, not commands.** A program describes a value rather than a sequence of steps.
2. **Immutability.** Names do not change meaning; sharing is therefore safe; aliasing bugs become unspellable.
3. **Purity at the core, effects at the edges.** The thick middle can be read and tested locally.
4. **Functions are values.** Map, filter and fold say the intent that a loop hides.
5. **Recursion is repetition, and tail calls make it a loop** with no stack growth and no cost.
6. **Types describe the problem.** Sum types make illegal states unrepresentable; pattern matching takes them apart; the compiler proves you handled every case; inference means you barely write any of it down.

The next chapter asks the obvious follow-up question: given that all of this is fifty years old and several excellent languages already implement it, what is Rill for?

Chapter 2

Why Rill

Chapter 1 argued for a style. This chapter argues for a particular language, which is a harder thing to do honestly, because the style already has good implementations and most readers should probably use one of them.

So let us start with the case against.

2.1 The case against a new language

If you want a functional language today, you have excellent choices, and every one of them is more sensible than a language whose version number starts with 0.2.

Haskell is the purest expression of the ideas and has thirty years of research behind it. **OCaml** is fast, strict, mature, and runs serious industrial systems. **F#** and **Scala** give you the style with a vast ecosystem underneath. **Rust** brings sum types, exhaustive matching and `Option`/`Result` to systems programming with no runtime at all. **Elixir** and **Erlang** own fault-tolerant distributed systems. **Clojure** gives you Lisp on the JVM with the best story about immutable data structures anywhere.

Against that, any new language starts with no libraries, no community, no tooling beyond what its author wrote, and no track record. That is not a small disadvantage. It is usually decisive, and it should be.

So a new language has to be answering a question the existing ones do not.

2.2 The question Rill answers

Look at the functional languages by *what they produce*:

Language	Output	Runtime underneath
Haskell	native binary	GC, green threads, several MB
OCaml	native binary	GC, ~1 MB minimum
F#, Scala, Clojure	bytecode	.NET or JVM
Elixir, Erlang	bytecode	BEAM VM
Rust	native binary	none — but not a functional language

Every functional language on that list ships a garbage collector. Most ship a substantial runtime; several ship an entire virtual machine. If you want functional programming *and* you care about binary size, startup time, predictable latency, or memory footprint, the list is short and the answer has historically been "use Rust and give up the functional parts."

Rill's question is:

> **Can you have ML's types and Go's concurrency in a binary the size of a C program, with no garbage collector and no pauses?**

That is the whole design brief. Everything specific about the language follows from it.

2.3 The four decisions

2.3.1 Strict, not lazy

Rill evaluates arguments when it is told to, like ML, OCaml, Scheme and every mainstream language — not on demand, like Haskell.

Laziness buys real things: infinite structures, and control constructs written as ordinary functions. It costs two, and they are the ones that matter here. Every unevaluated expression becomes a heap-allocated *thunk*, so allocation appears where you did not write any; and the time at which work happens stops being visible in the source, which is why Haskell's characteristic beginner bug is a space leak in an accumulator that looks perfectly innocent.

For a language whose brief is predictable memory, laziness is the wrong trade. When a Rill program computes a value, it computes it there.

2.3.2 Reference counting decided at compile time, not a garbage collector

This is the decision that makes the rest possible, and it is worth being precise about what it means.

A tracing garbage collector runs periodically, walks the live objects, and frees the rest. It buys you cycle collection and very cheap allocation; it costs you a runtime, unpredictable pauses, and a heap several times larger than the live set.

Rill instead does what Perceus-style systems do: the **compiler** works out where each value's last use is and inserts the retain and release operations itself, following ownership rules. Nothing runs periodically. A value is freed at the instruction after it dies.

Three consequences you will see in this book:

- **Deterministic.** No pauses, ever. The worst case is the cost of a free.
- **Small.** Peak memory is close to the live set rather than a multiple of it. The binary-trees benchmark in §2.4 shows Rill using less memory than either GC'd language while running faster.
- **Cyclic data leaks.** This is the honest cost. Reference counting cannot collect a cycle, and Rill has no cycle collector. In practice immutable data is overwhelmingly acyclic — you cannot easily build a cycle out of values that never change — but if you construct one through buffers or FFI, it will not be freed. Run with `RILL_DEBUG_ALLOC=1` and the runtime tells you how many allocations were still live at exit.

2.3.3 Go's concurrency, not Haskell's or Erlang's

Rill takes `spawn`, typed channels, and `select` more or less directly from Go: many lightweight threads, communicating by passing values, over a scheduler that parks a strand when it blocks on a channel or a socket.

The reason is that this model composes with the memory decision. Because strands run on one OS thread by default, reference counting can be *non-atomic* — an ordinary increment rather than a locked one — and that is worth a great deal on every value operation in the language. Programs that want real parallelism opt in at build time (`--parallel`) and at run time (`RILL_THREADS=8`), at which point the counts become atomic and you pay for what you asked for.

100,000 strands cost about 53 MB and start in microseconds. Chapter 16 shows how that is done — the trick is that a parked strand's stack is copied out to a heap block sized to what it actually used, rather than reserving a stack per thread.

2.3.4 Monomorphized generics and static trait dispatch

A generic function in Rill is compiled once per concrete type it is used at. Nothing is boxed, no type information is carried at run time, and a trait method call is resolved to a direct call during compilation. A missing implementation is a compile error, not a runtime failure.

The cost is compile time and code size for heavily generic code; the benefit is that `map` over a list of integers compiles to the same machine code you would have written by hand.

2.4 What that produces

These are the repository's own benchmarks (`python3 benchmarks/run.py`, Apple M4, outputs verified identical across all three languages):

benchmark	rill	go	ocaml	rill RSS	go RSS	ocaml RSS
fib(35)	**17.2 ms**	22.0 ms	23.7 ms	**1.3 M**	3.6 M	2.1 M
binary trees (alloc churn)	**33.0 ms**	55.9 ms	60.0 ms	**13.3 M**	16.6 M	18.3 M
100k-thread channel ring	**13.5 ms**	96.2 ms	61.7 ms	**53.4 M**	269.4 M	103.0 M
100M-iteration loop	**127.6 ms**	271.6 ms	295.3 ms	**1.3 M**	3.5 M	2.1 M

And the size figure, which you can reproduce in ten seconds:

```
$ rill build hello.rill -o hello && ls -l hello
33 KB
```

Against roughly 2.1 MB for the equivalent Go program and about 1 MB for OCaml.

Three cautions about reading any table like this one. It is four microbenchmarks, not a workload. It is one machine and one operating system. And the honest summary of the first row is not "Rill is faster than Go at arithmetic" — it is "both of these are LLVM-quality code generation and the difference is small." The rows that carry real information are the third, where the concurrency implementation differs by 7×, and the second, where a compiler inserting frees beats two mature garbage collectors on both time *and* memory.

2.5 What Rill deliberately does not have

A language is defined as much by its omissions, and these are chosen rather than pending:

- **Mutation.** No assignment, no mutable fields. State is threaded through arguments or passed over channels. The exception is `Buf`, a counted block of C-width elements, which exists so that C libraries and pixel buffers are workable; it is a deliberate hole with a fence around it (Chapter 15).
- **Exceptions.** Errors are `Result` values. Nothing unwinds the stack from under you.
- **Null.** Absence is `Option`.
- **Implicit numeric conversion.** `1 + 2.0` is a type error, and the compiler says so.
- **Inheritance and subtyping.** Traits and composition instead.
- **Laziness.** Discussed above.
- **A garbage collector.** Discussed above.
- **Macros, effect systems, higher-kinded types.** Not on principle, but because the language is 0.2 and each of them is a large design in itself.

2.6 Who this book is for

You should read on if you want to understand what a functional language looks like when every feature has to justify itself against a native-code, no-runtime budget — or if you want a working tour of the ideas in Chapter 1 with a compiler you can run each example through.

You should not adopt Rill for production work. It is a young language with one implementation, a small standard library, and no ecosystem. This book will tell you plainly when a thing is missing rather than working around it quietly.

What Rill does have, and what makes it a good vehicle for a book like this one, is that the whole language fits in your head. There are no dark corners, no legacy syntax, and no feature that exists because a committee needed it. Every construct in Chapters 3 through 8 is one you will use in Chapters 9 through 14, and there are no others.

2.7 Setting up

Everything in this book is run with the compiler in this repository:

```
cargo build --release            # builds ./target/release/rill
```

Two commands cover the whole book:

```
rill run   program.rill          # compile to a temporary binary and run it
rill build program.rill -o prog  # compile to ./prog
```

Two more are worth knowing early:

```
rill fmt   program.rill          # canonical formatting — there is exactly one
rill repl                        # try an expression without writing a file
```

Every program in this book lives under `book/code/`, is compiled by the same compiler as part of preparing the text, and prints the output shown. Where a listing shows an error message, that message was produced by running the broken program, not written by hand.

Chapter 3 starts with the smallest program that does anything.

Part II

The Language

Chapter 3

First Programs

This chapter covers the parts of Rill you need before anything interesting can happen: how a program is shaped, how values are named, and how the compiler talks to you when you get it wrong. It is short, and almost everything in it will be familiar from other languages. The unfamiliar parts are flagged.

3.1 The smallest program

```
# book/code/ch03/hello.rill
fn main() = println("Hello, Rill")
```

```
$ rill run hello.rill
Hello, Rill
```

Four things are worth noticing in one line of code.

`fn name(args) = expression` is the whole function syntax. There is no `return`, because the body *is* the value. There are no braces, because Rill uses layout.

Every program needs `main`. It takes nothing and returns nothing; the program exits when it returns.

`#` starts a comment, running to the end of the line.

`rill run` compiles and runs. It is not an interpreter — a native binary is produced and executed. `rill build hello.rill -o hello` keeps the binary, which is about 33 KB.

3.2 Expressions

```
# book/code/ch03/arithmetic.rill
fn main() =
  println(2 + 3 * 4)
  println(7 / 2)
  println(7 % 2)
  println(7.0 / 2.0)
  println(to_float(7) / 2.0)
  println("a" + "b" + "c")
  println(3 < 4 && !(1 == 2))
```

```
$ rill run arithmetic.rill
14
3
1
3.5
3.5
abc
true
```

The operators are the usual ones: `+ - * / %` for arithmetic, `== !=` for equality, `< <= > >=` for ordering, `&& || !` for logic, `+` for joining strings. Precedence is conventional — `2 + 3 * 4` is 14.

Two things differ from what you may expect.

`/` on integers truncates (`7 / 2` is 3) and `%` is integers only. Float division is written on floats.

There are no implicit conversions. This is a type error, and it is the first error most newcomers meet:

```
# book/code/ch03/mixing.rill
fn main() = println(1 + 2.0)
```

```
$ rill run mixing.rill
mixing.rill: type error at line 1: operands of a binary operator: expected Int, found Float
```

`1` is an `Int`, `2.0` is a `Float`, and Rill will not quietly widen one to the other. `to_float` and `to_int` cross between them and you write them down.

This is a small nuisance about twice a week and it removes an entire class of silent bug — the kind where a currency total is a hair off because something became a float three call frames ago. The rule is: *numeric types never mix by accident.*

> **Float literals need a decimal point**
>
> `2.0` is a `Float`; `2` is an `Int`. There is no context in which `2` becomes a float on its own. When a float prints as a whole number Rill shows it without the point — `float_to_str(72.0)` is `"72"` — which the next listing demonstrates.

3.3 Bindings and blocks

A function body can be an indented block. Each line is evaluated in order, and the block's value is its last line:

```
# book/code/ch03/bindings.rill
fn celsius(f) = (f - 32.0) * 5.0 / 9.0

fn report(f) =
  c = celsius(f)
  warm = c > 20.0
  float_to_str(f) + "F is " + float_to_str(c) + "C" + (if warm then " (warm)" else " (cold)")

fn main() =
  println(report(72.0))
  println(report(50.0))
```

```
$ rill run bindings.rill
72F is 22.2222C (warm)
50F is 10C (cold)
```

`c = celsius(f)` is a **binding**, not an assignment. It introduces a name for a value, visible for the rest of the block. What it cannot do is change one:

```
c = celsius(f)
c = c + 1.0        # not "c becomes bigger" — a second binding of the same name
```

There is no assignment operator in Rill. This is the point of Chapter 1 made concrete: `c` means one thing from where it is written to the end of the block, and no call in between can change what it means.

`if` is an expression. Notice where it appears in the listing — in the middle of a string concatenation. `if c then a else b` produces a value, so it goes wherever a value goes. There is no statement form and no `elif`; chains are written by putting another `if` in the `else`.

Both branches must have the same type, which is why an `else` is never optional.

3.4 Layout

Blocks are indentation, two spaces per level, spaces only. There are no braces and no semicolons.

```
fn report(f) =
  c = celsius(f)          # this block is two spaces in
  warm = c > 20.0
  ...
```

Two rules cover the cases where this could be ambiguous:

A construct may continue on the next line only where the grammar allows it. `then` and `else` may begin a line, so a long conditional can be broken up.

Inside parentheses, indentation is ignored. Anything written between brackets stays on one line as far as the layout rules are concerned.

If you are unsure how something should be laid out, do not guess — run `rill fmt` on the file. Rill has exactly one canonical formatting, the formatter produces it, and every listing in this book has been through it.

```
rill fmt program.rill           # rewrite in place
rill fmt --check program.rill   # exit non-zero if it is not canonical
```

3.5 Printing and converting

`println(x)` / `print(x)`	print any value, with or without a newline
`show(x) -> Str`	render any value as a string
`int_to_str` `float_to_str` `bool_to_str` `str_to_int`	conversions to and from `Str`
`to_float(n)` `to_int(x)`	between `Int` and `Float`; `to_int` truncates toward zero

`println` takes any value type, including data types you define — Chapter 5 shows what that prints and how to change it.

3.6 The REPL

For trying one expression, there is no need for a file:

```
$ rill repl
> 2 + 3 * 4
14
> fn double(x) = x * 2
> double(21)
42
```

Definitions persist for the session; bindings do not. Under the hood each entry is compiled to a native binary and run, which takes a few hundred milliseconds — slower than an interpreter, and it means the REPL and the compiler can never disagree about what the language means.

3.7 Exercises

1. Write a program that prints the area and circumference of a circle of radius 2.5. Do it without writing `3.14159` twice.
2. `rill run` the following and read the error before fixing it: `fn main() = println(str_len(42))`
3. Write `fahrenheit(c)` as the inverse of `celsius(f)` from §3.3, and print `fahrenheit(celsius(72.0))`. Explain the last digit of what you get.
4. Try writing a block whose lines are indented three spaces, then run `rill fmt` on it.

Chapter 4 introduces the part that makes the style work: functions that take and return other functions.

Chapter 4

Functions Are Values

In Chapter 1 this was an abstract claim. Here it becomes the thing you actually type. Functions in Rill are values like integers are values: they can be passed to functions, returned from functions, stored in data, and made up on the spot.

4.1 Passing a function by name

```
# book/code/ch04/values.rill
fn double(x) = x * 2

fn apply_twice(f, x) = f(f(x))

fn multiplier(k) = \x -> x * k

fn main() =
  println(apply_twice(double, 5))
  triple = multiplier(3)
  println(triple(7))
  println(apply_twice(triple, 2))
  println(apply_twice(\s -> s + "!", "hi"))
```

```
$ rill run values.rill
20
21
18
hi!!
```

Line by line:

`apply_twice(double, 5)` passes `double` without calling it. `double` is `double(x)` called; `double` on its own is the function itself. This is the same distinction as Python's `f` versus `f()`.

`\x -> x * k` is a lambda: backslash, parameters, arrow, body. Multiple parameters are separated by commas (`\a, b -> a + b`).

`multiplier(3)` returns a function that has captured 3. That is a closure. `triple` is a value carrying a 3 around inside it, and it outlives the call to `multiplier` that created it.

The last line shows `apply_twice` working on strings. The same `apply_twice` — nothing was overloaded and nothing was declared generic. §4.4 explains why.

4.2 Pipelines

Nested calls read inside-out, which is the wrong direction for a sequence of steps. The `|>` operator turns them around:

```
x |> f(a)      means      f(x, a)
```

The value on the left becomes the *first* argument of the call on the right.

```
# book/code/ch04/pipeline.rill
fn square(x) = x * x

fn even(x) = x % 2 == 0
```

```
fn main() =
  println(range(1, 10) |> filter(even) |> map(square) |> sum)
  println(sum(map(filter(range(1, 10), even), square)))
  println(range(1, 5) |> map(\x -> x * 10))
```

```
$ rill run pipeline.rill
220
220
Cons(10, Cons(20, Cons(30, Cons(40, Cons(50, Nil)))))
```

The first two lines are the same computation and produce the same answer. The first reads left to right in the order the work happens: take one to ten, keep the even ones, square them, add them up. The second reads outside-in, and you have to find the innermost call before you can start.

This is not a small stylistic matter. Pipelines are how functional code stays readable as it grows, and `|>` is why `map(l, f)` takes the list first — every list function in the prelude is arranged so that the data comes first and the pipeline works.

The third line shows what a list looks like when printed raw: `Cons(10, Cons(20, ...))`. Chapter 6 explains why, and Chapter 8 shows how to make it print as `[10 20 30 40 50]` instead.

Compared to other languages

`|>` comes from F# and OCaml; Elixir has it too. Where a language lacks it, the same job is done by method chaining — `xs.filter(even).map(square)` in Rust, Scala or JavaScript. The advantage of an operator over methods is that any function works in a pipeline, including one you wrote five minutes ago, without it having to be a method on the type.

4.3 Where closures come from

A closure is a function plus the variables it captured. `multiplier` above is the toy version; here is the shape you actually use it in.

Suppose you want to keep the elements of a list above some threshold. The threshold is not known when you write the code — it arrives at run time:

```
fn above(xs, limit) = filter(xs, \x -> x > limit)
```

The lambda mentions `limit`, which is not one of its parameters. It comes from the enclosing function, and it is captured by value when the lambda is made. Every call to `above` produces a different predicate, each carrying its own `limit`.

That is the whole mechanism, and it is behind most uses of higher-order functions: a comparison that knows which field to sort on, a callback that knows which connection it belongs to, a validator that knows the rules it was configured with.

4.4 Generics happen without asking

None of the functions in §4.1 were declared generic, and `apply_twice` worked on integers and strings. This is Hindley–Milner type inference, and it is the feature that lets a statically typed language feel like a dynamically typed one.

```
# book/code/ch04/generic.rill
fn id(x) = x

fn twice(f, x) = f(f(x))

fn biggest(a, b) = if a > b then a else b

fn main() =
  println(id(42))
```

```
  println(id("forty-two"))
  println(twice(\n -> n + 1, 0))
  println(twice(\s -> s + "-", "x"))
  println(biggest(3, 9))
  println(biggest("apple", "pear"))
  println(biggest(1.5, 0.5))
```

```
$ rill run generic.rill
42
forty-two
2
x--
9
pear
1.5
```

The compiler works out the most general type each function could have:

- `id(x) = x` — the result is the argument, so `id : a -> a` for any `a`.
- `twice(f, x) = f(f(x))` — `f` is applied to `x` and then to its own result, so `f`'s input and output are the same type: `twice : (a -> a, a) -> a`.
- `biggest(a, b) = if a > b then a else b` — this one is more interesting. The body uses `>`, so whatever `a` is must be **ordered**. The inferred type is "for any ordered `a`, `(a, a) -> a`", and that requirement travels with the function.

Try `biggest` at a type with no ordering and the compiler names both the function and the type. This is the same mechanism as Rust's trait bounds or Haskell's type classes, with the difference that you did not write the bound — it was inferred from the body.

4.4.1 Monomorphization

Generic in Rill does not mean boxed. The compiler emits **one specialized copy per concrete type**: `id` at `Int` and `id` at `Str` are two separate machine- code functions, each with no indirection, no type tag, and nothing to unbox.

The trade is compile time and binary size against run time. Chapter 15 returns to it; the practical effect is that `map(xs, \x -> x * 2)` over a list of integers compiles to the loop you would have written by hand.

4.4.2 Two limits worth knowing now

A function used before its definition is pinned to that use. If line 3 calls `helper(1)` and `helper` is defined on line 40, `helper` is fixed at `Int` even if it would otherwise have been generic. Define first, or accept the narrower type.

There is no polymorphic recursion. A generic function cannot call itself at a different type than it was called at. This rules out a small number of exotic data structures and is not something you are likely to trip over by accident.

4.5 The three combinators

Almost every loop you would write in another language is one of these:

	Type, informally	What it produces
`map(list, f)`	`(List(a), a -> b) -> List(b)`	same length, each element transformed
`filter(list, p)`	`(List(a), a -> Bool) -> List(a)`	the elements satisfying `p`, in order
`fold(list, init, f)`	`(List(a), b, (b, a) -> b) -> b`	one value, accumulated left to right

`map` and `filter` are the easy two. `fold` is the one that takes a moment, and it is worth spending that moment because the other two are special cases of it.

Fold walks the list carrying an accumulator. At each element it calls `f(accumulator, element)` and the result becomes the new accumulator:

```
fold(Cons(1, Cons(2, Cons(3, Nil))), 0, \a, x -> a + x)

  a = 0,  x = 1   →   1
  a = 1,  x = 2   →   3
  a = 3,  x = 3   →   6
                      6
```

That is `sum`, and in the prelude that is literally how `sum` is defined:

```
fn sum(l) = fold(l, 0, \a, x -> a + x)
```

Change the operator and the seed and you get a different aggregate: `max` of a list, the length, a string joined together, a count of matches. Once you see that a fold is "a loop with one variable in it", you stop writing loops.

4.6 Exercises

1. Write `compose(f, g)` returning a function that applies `g` then `f`. Use it to build "double then increment" and "increment then double", and show they differ.
2. Write `count_if(xs, p)` two ways — once with `filter` and `len`, once with a single `fold`.
3. `product` is to $*$ what `sum` is to `+`. Write it as a fold. What seed?
4. Predict the type of `fn pair_up(f, a, b) = f(a, b)` before checking. What does the compiler infer for `fn odd_one(f, x) = f(x) + 1`?

Chapter 5 introduces the other half of the language: describing your own data, and taking it apart with pattern matching.

Chapter 5

Data and Pattern Matching

This is the chapter that changes how you design programs. Everything so far has been a variation on things other languages do; sum types and exhaustive pattern matching are the part where a reader from Java or Python starts writing different code, not just different syntax.

5.1 Declaring a type

```
type Shape
  Circle(r: Float)
  Rect(w: Float, h: Float)
  Triangle(base: Float, height: Float)
```

Read it as: *a `Shape` is a `Circle` with a radius, **or** a `Rect` with a width and a height, **or** a `Triangle` with a base and a height.*

The "or" is the important word. A struct is an *and* — a point has an x and a y. This is the other one, and most languages before 2015 did not have it.

- Constructor names start with a capital and are **global** — `Circle` is a function you call, not `Shape.Circle`.
- Field names are optional documentation. `Rect(Float, Float)` means the same thing.
- A constructor with no fields (`Red`, `Nil`, `None`) is a value rather than a function, and never allocates.
- A type and one of its constructors may share a name, which is handy for single-case wrappers: `type Money  Money(Int)`.

5.2 Taking it apart

A value built with `Circle(1.0)` has to be examined before you can use it, and that is what `match` is for:

```
# book/code/ch05/shapes.rill
type Shape
  Circle(r: Float)
  Rect(w: Float, h: Float)
  Triangle(base: Float, height: Float)

fn area(s) = match s
  Circle(r) -> 3.14159 * r * r
  Rect(w, h) -> w * h
  Triangle(b, h) -> b * h / 2.0

fn describe(s) = match s
  Circle(_) -> "round"
  _ -> "straight-edged"

fn main() =
  shapes = Cons(Circle(1.0), Cons(Rect(2.0, 3.0), Cons(Triangle(4.0, 5.0), Nil)))
  println(shapes)
  println(map(shapes, area))
  println(map(shapes, describe))
```

```
$ rill run shapes.rill
Cons(Circle(1), Cons(Rect(2, 3), Cons(Triangle(4, 5), Nil)))
```

```
Cons(3.14159, Cons(6, Cons(10, Nil)))
Cons(round, Cons(straight-edged, Cons(straight-edged, Nil)))
```

`match` does two things at once, and this is what makes it more than a switch: it **tests** which constructor a value was built with, and it **binds** the fields to names in the same breath. `Circle(r) -> ...` means "if this is a circle, call its radius `r` in the branch that follows".

`_` is a wildcard: it matches anything and binds nothing. `describe` uses it twice — once inside a constructor to say "a circle, don't care what radius", once on its own as a catch-all.

Notice also that every data value already prints: `Cons(Circle(1), Cons(Rect(2, 3), Nil))`. You do not write a `toString`. (The floats show as `1` and `2` because Rill prints whole floats without a decimal point. Chapter 8 replaces this derived printing with your own.)

5.3 The compiler checks you covered every case

Leave out an arm:

```
# book/code/ch05/missing.rill
type Colour
  Red
  Green
  Blue

fn name(c) = match c
  Red -> "red"
  Green -> "green"

fn main() = println(name(Red))
```

```
$ rill run missing.rill
missing.rill: type error at line 6: match is not exhaustive; no arm matches `Blue`
```

The program never runs. The compiler enumerated the cases, found one with no arm, and **named it**.

This is worth pausing on, because it changes what refactoring feels like. Add a fourth colour to the type and recompile: every `match` that now has a hole is listed, with a line number and the missing case. Not a runtime `default:` that logs a warning in production six weeks later — a compile error, immediately, everywhere.

Programmers who work this way come to rely on it. The move "add a case to the type, then fix what the compiler complains about" is a genuinely different way to change a large program, and it is only available if the checker is exact.

The other half of the check is redundancy: an arm that earlier arms already cover is rejected too, so a `_` in the middle of a match is an error rather than silent dead code.

5.4 Option: absence with a name

Rill has no null. A value that might not be there has a type that says so, and the prelude defines it:

```
type Option(a)
  None
  Some(a)
```

```
# book/code/ch05/option.rill
fn find_even(l) = match l
  Nil -> None
  Cons(x, rest) -> if x % 2 == 0 then Some(x) else find_even(rest)

fn describe(o) = match o
  None -> "none found"
  Some(n) -> "found " + int_to_str(n)
```

```
fn main() =
  println(describe(find_even(Cons(1, Cons(3, Cons(8, Cons(5, Nil)))))))
  println(describe(find_even(Cons(1, Cons(3, Nil)))))
  println(unwrap_or(find_even(Cons(1, Nil)), 0 - 1))
  println(opt_map(Some(21), \x -> x * 2))
```

```
$ rill run option.rill
found 8
none found
-1
Some(42)
```

`find_even` returns `Option(Int)`, and its signature therefore *tells the caller* that there may be no answer. There is no way to use the integer without first saying what happens when there isn't one, because the integer only exists inside the `Some` branch.

Compare the three ways other languages handle this:

Approach	The failure mode
Return `null` / `None`	forgetting to check compiles fine and crashes later
Throw an exception	the signature doesn't say; control jumps somewhere
Return `Option`	forgetting to check does not compile

Two prelude helpers cover the common shapes: `unwrap_or(o, d)` supplies a default, and `opt_map(o, f)` applies a function inside the `Some` and leaves `None` alone.

> **The last line prints `Some(42)`, not 42**
>
> `opt_map` gives back an `Option`, because the value might still be absent. Getting the plain integer out means saying what `None` becomes — that is the whole discipline in one line.

5.5 Result: failure as a value

Errors work the same way. Rill has no exceptions; a function that can fail says so in its type:

```
type Result(a, e)
  Ok(a)
  Err(e)
```

```
# book/code/ch05/result.rill
type Money
  Money(Int)

fn parse_amount(s) =
  n = str_to_int(s)
  if str_len(s) == 0 then Err("empty") else if n < 0 then Err("negative: " + s) else Ok(Money(n))

fn show_result(r) = match r
  Ok(Money(c)) -> "ok: " + int_to_str(c)
  Err(msg) -> "error: " + msg

fn main() =
  println(show_result(parse_amount("1250")))
  println(show_result(parse_amount("")))
  println(show_result(parse_amount("-5")))
```

```
$ rill run result.rill
ok: 1250
```

```
error: empty
error: negative: -5
```

Two things to take from this listing.

`Ok(Money(c))` is a nested pattern. It unwraps the `Result` and the `Money` in one step and binds the integer inside. Patterns nest as deeply as the data.

Nothing unwinds the stack. The error travels back as an ordinary return value along the ordinary path. There is no invisible second exit from every function, which means the reading rule from Chapter 1 still holds: what you see is what happens.

The cost is that errors must be handled or explicitly passed along at each step, and in a deep call chain that is more typing than an exception. Languages in this family answer with syntax — Rust's `?`, Haskell's do-notation. Rill 0.2 does not have that sugar yet; you write the `match`, or use `res_map` when you only want to transform the success case.

5.6 Patterns nest

```
# book/code/ch05/nested.rill
fn classify(l) = match l
  Nil -> "empty"
  Cons(Some(0), _) -> "starts with zero"
  Cons(Some(n), Cons(Some(m), _)) -> "two: " + int_to_str(n + m)
  Cons(Some(n), _) -> "one: " + int_to_str(n)
  Cons(None, _) -> "starts with nothing"

fn main() =
  println(classify(Nil))
  println(classify(Cons(Some(0), Nil)))
  println(classify(Cons(Some(3), Cons(Some(4), Nil))))
  println(classify(Cons(Some(9), Nil)))
  println(classify(Cons(None, Nil)))
```

```
$ rill run nested.rill
empty
starts with zero
two: 7
one: 9
starts with nothing
```

A pattern may be:

Pattern	Matches
`_`	anything, binds nothing
`name`	anything, binds it to `name`
`0`, `42`	that integer exactly
`Ctor(p1, p2)`	that constructor, with sub-patterns for its fields

Arms are tried **in order**, so put the specific ones first. `Cons(Some(0), _)` has to come before `Cons(Some(n), _)`, because the second would swallow it — and if you get that backwards, the redundancy check says so rather than letting the arm sit there unreachable.

This listing is also a small argument for the whole approach. Five cases, each one a shape drawn on the page, and no combination of `if`, `is`, `!= null` and field access to read past. The structure of the data is visible in the structure of the code.

5.7 Designing with sum types

The technique worth learning from this chapter is **making illegal states unrepresentable**. It is a design move, not a language feature.

Here is a connection, modelled the way most codebases model it:

```
type Connection
  Connection(host: Str, port: Int, socket: Int, error: Str, connected: Bool)
```

Five fields, and the invariants live in your head: if `connected` then `socket` is valid and `error` is empty; if not, `socket` is -1 and `error` might be set. Nothing enforces any of it, every function must check, and one wrong branch gives you a read from socket -1.

Now the same thing as a sum:

```
type Connection
  Idle(host: Str, port: Int)
  Live(host: Str, port: Int, socket: Int)
  Failed(host: Str, error: Str)
```

There is no state with a socket and an error. There is no "connected but socket is -1". A function that needs a live socket takes the `Live` case and cannot be handed anything else. The invariants moved out of your head and into the type, where the compiler enforces them for free.

The question to ask of any record you write is: **how many combinations of these fields are meaningful, and how many are possible?** When the second number is much larger than the first, there is a sum type waiting.

5.8 Exercises

1. Model a traffic light as a type and write `next(light)`. Then add a flashing-amber state and see what the compiler asks you to fix.
2. Write `safe_div(a, b)` returning `Option(Int)`, and use it to write `average(list)` returning `Option(Int)` that is `None` for an empty list.
3. Take the `Connection` example in §5.7 and write `describe(conn)` for the five-field version and the three-case version. Which one can be wrong?
4. Write `depth(shape_list)`-style nested matching practice: a function over `List(Option(Shape))` that returns the total area, treating `None` as zero.

Chapter 6 looks at the list type that has been quietly used in every listing so far, and at the prelude built on top of it.

Chapter 6

Lists and the Prelude

Every listing so far has used `Cons` and `Nil` without explanation. This chapter explains them, which turns out to explain a good deal about the language: the list is not a built-in type, it is an ordinary sum type declared in the prelude, and every function over it is ordinary Rill you could have written.

6.1 The list is not special

Here is the whole definition, from `crates/rill_syntax/src/prelude.rill`:

```
type List(a)
  Nil
  Cons(head: a, tail: List(a))
```

A list of `a` is either empty, or a first element followed by a list of `a`.

That is a **recursive** sum type: `Cons`'s second field is another `List(a)`. And it is **generic**: `a` is a type parameter, so `List(Int)` and `List(Str)` are different types made from the same declaration.

```
# book/code/ch06/lists.rill
fn main() =
  xs = Cons(1, Cons(2, Cons(3, Nil)))
  println(xs)
  ...
```

```
$ rill run lists.rill
Cons(1, Cons(2, Cons(3, Nil)))
3
6
Cons(3, Cons(2, Cons(1, Nil)))
Cons(1, Cons(2, Cons(3, Cons(4, Nil))))
Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Nil)))))
Cons(8, Cons(9, Cons(10, Nil)))
Some(2)
None
true
false
```

`Cons(1, Cons(2, Cons(3, Nil)))` is verbose, and Rill 0.2 has no `[1, 2, 3]` literal. In practice most lists come from `range`, from `split`, or from a function rather than from being typed out — but this is a genuine ergonomic gap and worth naming rather than glossing over.

Why a linked list and not an array?

Because of immutability. Adding an element to the front of a linked list allocates one cell that points at the existing list, and the existing list is untouched — so both the old and the new list are valid, and they share every element. An immutable array would have to be copied.

The cost is that indexing is O(n) and there is no cache locality. Lists are the right structure for *sequences you walk*, which is most of them; when you need indexing or packed memory, Chapter 15's `Buf` is the answer.

6.2 Reading the prelude

The value of the list being ordinary is that the prelude is readable. Here is `map`:

```
fn map(l, f) = match l
  Nil -> Nil
  Cons(x, rest) -> Cons(f(x), map(rest, f))
```

Two cases, matching the two cases of the type. An empty list maps to an empty list; a list with a head maps to *`f` of the head, followed by the map of the rest*. That is the entire definition and it is the shape of nearly every list function you will ever write.

```
fn filter(l, p) = match l
  Nil -> Nil
  Cons(x, rest) -> if p(x) then Cons(x, filter(rest, p)) else filter(rest, p)

fn fold(l, acc, f) = match l
  Nil -> acc
  Cons(x, rest) -> fold(rest, f(acc, x), f)

fn len(l) = match l
  Nil -> 0
  Cons(_, rest) -> 1 + len(rest)
```

Learn to write these three from memory. Once the shape is automatic — *base case for `Nil`, recursive case for `Cons`* — a large class of problems stops needing any thought.

Note the difference between `fold` and the other two: in `fold` the recursive call is the **last thing that happens**, while in `map` a `Cons` is built around the result and in `len` a `1 +` is applied to it. That difference is the subject of Chapter 7, and it decides which of these run in constant stack.

6.3 What the prelude gives you

Every program has these in scope. Any definition of your own with the same name replaces the prelude's.

Types: `Option(a)`, `Result(a, e)`, `List(a)`.

Lists

`map(l, f)` `filter(l, p)` `fold(l, acc, f)`	the three combinators
`len(l)` `sum(l)`	aggregate
`range(a, b)`	the list `a` through `b` inclusive
`reverse(l)` `append(a, b)`	rearrange
`take(l, n)` `drop(l, n)` `nth(l, n)`	slice; `nth` returns `Option`
`any(l, p)` `all(l, p)`	quantify

Option and Result: `unwrap_or(o, d)`, `opt_map(o, f)`, `res_map(r, f)`.

Numbers: `max(a, b)`, `min(a, b)`, `abs(n)`.

Strings: `join(l, sep)`, `repeat(s, n)`, `split(s, sep)`, `lines(s)`, `trim(s)`, `contains(s, p)`, `starts_with(s, p)`.

IO: `args()`, `read_file(path) -> Result(Str, Str)`.

That is the whole standard library. It is small, and that is a fair criticism of a young language — there is no map, no set, no sort, no formatted printing. Chapters 10 and 11 write a sort and a map, which is one way to turn the complaint into a lesson.

6.4 Strings

```
# book/code/ch06/strings.rill
fn main() =
  s = "  the quick brown fox  "
  println("[" + trim(s) + "]")
  println(split(trim(s), " "))
  println(join(split(trim(s), " "), "-"))
  println(len(split(trim(s), " ")))
  println(contains(s, "brown"))
  println(starts_with(trim(s), "the"))
  println(repeat("ab", 3))
  println(str_len("fox"))
  println(str_sub("quick", 1, 3))
  println(str_find("quick", "ick"))
```

```
$ rill run strings.rill
[the quick brown fox]
Cons(the, Cons(quick, Cons(brown, Cons(fox, Nil))))
the-quick-brown-fox
4
true
true
ababab
3
uic
2
```

Strings are immutable and reference counted, and + joins them. The low-level operations are byte-indexed:

`str_len(s)`	length in bytes
`str_get(s, i)`	the byte at `i`, as an `Int`
`str_sub(s, start, len)`	a substring
`str_find(s, needle)`	index, or -1
`chr(byte)`	a one-byte string

"Byte-indexed" is the honest limitation: Rill 0.2 has no Unicode-aware string handling. ASCII text is fine; anything else needs care, and `str_len` counts bytes rather than characters.

Notice that `split` returns a `List(Str)`, so the list functions apply to it directly — `len(split(s, " "))` is a word count, and `join` puts it back together. Text processing in Rill is list processing.

6.5 Your own recursive types

Nothing about `List` is privileged, so your own structures work identically:

```
# book/code/ch06/mylist.rill
type Stack(a)
  Empty
  Push(top: a, rest: Stack(a))

fn depth(s) = match s
  Empty -> 0
  Push(_, rest) -> 1 + depth(rest)

fn peek(s) = match s
  Empty -> None
  Push(x, _) -> Some(x)
```

```
fn main() =
  s = Push(3, Push(2, Push(1, Empty)))
  println(s)
  println(depth(s))
  println(peek(s))
  println(peek(Empty))
```

```
$ rill run mylist.rill
Push(3, Push(2, Push(1, Empty)))
3
Some(3)
None
```

`peek` returning `Option` rather than crashing on an empty stack is the Chapter 5 discipline applied without being asked. Notice also that `Push` and `Cons` are the same structure with different names — a stack *is* a list, and the reason to declare your own is to give the operations names that mean something in your problem, and to stop list functions from applying where they would not make sense.

Chapter 11 builds a binary tree the same way, which is where recursive types start to earn their keep.

6.6 Exercises

1. Write `last(l)` returning `Option(a)`, and `init(l)` returning everything but the last element.
2. Write `zip(a, b)` pairing two lists into a `List` of a two-field type you declare. What should it do when the lists differ in length?
3. Write `words(s)` that splits on spaces and drops empty strings, so that `words("  a  b ")` has two elements.
4. `flatten(list_of_lists)` — write it with `fold` and `append`. Then work out why that is $O(n^2)$ and what it would take to fix.
5. Read `take` and `drop` in the prelude listing in §6.2 and predict what `take(range(1, 5), 100)` does before running it.

Chapter 7 answers the question this chapter kept deferring: if `len` calls itself once per element, what happens on a list of ten million?

Chapter 7

Loops Are Recursion

Rill has no `for` and no `while`. This is the feature that most reliably makes newcomers uneasy, and the uneasiness is about performance: *surely a million iterations of a recursive function will exhaust the stack?*

This chapter answers that concretely, with measurements, and it ends up teaching something more interesting than the answer.

7.1 The transformation

A loop has three parts: variables that change, a condition that ends it, and a body. A recursive function has all three, in different places:

```
# a loop, imperatively                # the same loop in Rill
i = 10                                fn go(i, acc) =
acc = 0                                 if i == 0 then acc
while i != 0:                           else go(i - 1, acc + i)
    acc = acc + i
    i = i - 1
```

- The **loop variables** become parameters.
- The **exit condition** becomes the `if`.
- **Advancing** becomes the arguments of the recursive call.

Nothing is assigned. Each call has its own `i` and its own `acc`, one step further along than the last.

```
# book/code/ch07/loop.rill
fn count_up(i, n) =
  if i > n then 0 else count_on(i, n)

fn count_on(i, n) =
  println(i)
  count_up(i + 1, n)

fn sum_to(n) = go(n, 0)

fn go(i, acc) = if i == 0 then acc else go(i - 1, acc + i)

fn main() =
  count_up(1, 3)
  println(sum_to(10))
  println(sum_to(10000000))
```

```
$ rill run loop.rill
1
2
3
55
50000005000000
```

Ten million iterations, and the program returns. Nothing overflowed.

Why `count_up` is split into two functions

`count_on` does two things — prints, then loops — so it needs a block. The `if` that decides whether to continue lives in `count_up`. Splitting a loop body from its condition this way is a common shape in Rill, and it is the price of not having a statement form for `if`.

7.2 Why it does not overflow

Look at where the recursive call sits in `go`:

```
fn go(i, acc) = if i == 0 then acc else go(i - 1, acc + i)
                                         ^^^^^^^^^^^^^^^^^^
                                         the last thing that happens
```

When `go` calls itself, there is nothing left to do afterwards. The result of the inner call *is* the result of the outer one. So the outer frame holds nothing worth keeping — no pending addition, no saved local — and the compiler reuses it instead of pushing a new one.

That is a **tail call**, and eliminating it turns the recursion into a jump. The machine code for `go` is a loop: compare, subtract, add, jump back.

In Rill this is a guarantee, not an optimization. A call in tail position compiles to a jump, so a tail-recursive function runs in constant stack for any number of iterations. The same guarantee exists in Scheme, ML, OCaml, Haskell and Erlang. It does *not* exist in Java, Python or JavaScript, which is why recursion has a bad reputation among programmers who learned there.

Here is the check, at twenty million iterations:

```
$ /usr/bin/time -l ./tail20
200000010000000
             1310720  maximum resident set size
```

1.3 MB. Twenty million iterations of a function calling itself, and the process uses about as much memory as a hello-world.

7.3 Making a loop out of a non-tail function

Not every recursion is in tail position. Compare:

```
fn sum_up(n) = if n == 0 then 0 else n + sum_up(n - 1)
                                      ^^^^^^^^^^^^^^^^
                                      an addition is still pending
```

Here the recursive call is *not* the last thing: after it returns, `n` must still be added. So the frame has to survive, and this is a genuine stack consumer.

The fix is the **accumulator**: carry the running answer forward as an argument so nothing is left pending.

```
fn sum_up(n) = go(n, 0)
fn go(n, acc) = if n == 0 then acc else go(n - 1, acc + n)
```

This is the single most useful transformation to have automatic. Whenever you write a recursion and find an operation waiting on the result, ask whether it can be pushed into an extra parameter. Usually it can. `reverse` in the prelude is exactly this trick:

```
fn rev_onto(l, acc) = match l
  Nil -> acc
  Cons(x, rest) -> rev_onto(rest, Cons(x, acc))

fn reverse(l) = rev_onto(l, Nil)
```

The accumulator is a list being built backwards, which is why reversing is the natural thing to fall out of it.

7.4 An honest measurement, and a surprise

Having explained that `sum_up` is not tail recursive and therefore consumes stack, here is what actually happens when you run it at twenty million:

```
$ /usr/bin/time -l ./deep20
200000010000000
             1294336  maximum resident set size
```

It works. 1.29 MB — no more than the tail-recursive version.

The explanation is that Rill compiles through LLVM at `-O3`, and LLVM performs its own transformation: when the pending operation is **associative**, as `+` is, it can rewrite accumulator-style recursion into a loop by itself. The compiler did to `sum_up` exactly what §7.3 told you to do by hand.

Three lessons, and the third is the one worth keeping:

One. The language guarantee and the optimizer are different things. Tail position is guaranteed by Rill. Everything else is LLVM being clever, and LLVM being clever is not a specification.

Two. Do not measure this with a toy. LLVM at `-O3` is aggressive enough to fuse whole pipelines: `len(range(1, 5000000))` never allocates the list at all, because the optimizer inlines both functions and discovers it is counting. Any microbenchmark you write to "prove" something about allocation may be measuring a program that was deleted.

Three. Write the accumulator version anyway. It costs one parameter, it holds when the pending operation is not associative, it holds when the recursion allocates, and it holds when you compile with `--debug`. Relying on an optimizer to rescue a shape you could have written correctly is a bet you will lose eventually, at scale, in production.

7.5 Loops that never end

A server loop is a tail-recursive function with no exit condition:

```
fn worker(jobs, results) =
  n = recv(jobs)
  send(results, n * 2)
  worker(jobs, results)
```

That call is in tail position, so this runs forever in constant stack. There is no `while true`, and there does not need to be — an infinite loop is just a recursion that never reaches its base case.

The same shape appears in the frame loop of a game, the accept loop of a server, and the read loop of a parser. Chapter 16 uses it throughout.

7.6 When not to write the loop at all

Most everyday iteration should not be a hand-written recursion. The prelude's combinators are themselves tail recursive where it matters, and they say what they mean:

```
range(1, 10) |> filter(\x -> x % 2 == 0) |> map(\x -> x * x) |> sum
```

Reach for explicit recursion when:

- the loop carries **several** pieces of state that do not fit a fold,
- it walks something that is **not a list** — a tree, a grid, a parser's input,
- it must **stop early**, which `filter` and `map` cannot express,
- or the shape genuinely is a loop and naming it makes the code clearer.

Otherwise use the combinator. The reason is Chapter 1's: `map` tells the reader the shape of the answer before they read the body, and a recursion does not.

7.7 Exercises

1. Write `factorial(n)` twice — once naively, once with an accumulator — and check both at `n = 20`. What happens at `n = 30`, and why is that a different problem than stack depth?
2. Write `count_down(n)` that prints `n` down to 1 and then `"done"`. Keep it tail recursive.
3. Write `fib(n)` with two accumulators, so that it is linear rather than exponential. Compare `fib(40)` against the naive two-call version.
4. Rewrite `len` from the prelude to be tail recursive. Does `map` admit the same treatment? What would you have to give up?
5. Take the pipeline in §7.6 and write it as one hand-rolled tail recursion. Which version would you rather find in unfamiliar code?

Chapter 8 covers the last piece of the language proper: traits, which let your own types plug into `println`, `<` and the rest.

Chapter 8

Traits

A trait is an interface: a set of operations that a type can provide. Rill uses them for two jobs — letting your own types plug into the language's own operators, and writing functions that work across unrelated types.

This is the last chapter of the language tour. After it you have seen everything, and the rest of the book is about using it.

8.1 Plugging into the language

Two traits are built in, and implementing them changes how the language treats your type.

```
# book/code/ch08/traits.rill
type Money
  Money(Int)

impl Show(Money)
  fn show(m) = match m
    Money(c) -> "$" + int_to_str(c / 100) + "." + pad(c % 100)

fn pad(n) = if n < 10 then "0" + int_to_str(n) else int_to_str(n)

impl Ord(Money)
  fn compare(a, b) = match a
    Money(x) -> match b
      Money(y) -> x - y

fn main() =
  println(Money(1250))
  println(Money(305))
  println(max(Money(1250), Money(305)))
  println(Money(500) < Money(750))
  prices = Cons(Money(1250), Cons(Money(305), Cons(Money(999), Nil)))
  println(prices)
```

```
$ rill run traits.rill
$12.50
$3.05
$12.50
true
Cons($12.50, Cons($3.05, Cons($9.99, Nil)))
```

`Show` replaces derived printing. Without it, `println(Money(1250))` prints `Money(1250)`; with it, `$12.50`. And notice the last line — the list printed its elements through your `show`, without being told. Derived printing calls `show` on each field, so an impl anywhere composes everywhere.

`Ord` enables `<`, `<=`, `>`, `>=`. You write one function, `compare(a, b)`, returning a negative number, zero, or a positive number, and the four operators follow from it. So does `max`, and so will the sort in Chapter 10 — anything written against "an ordered type" now accepts `Money`.

That is the leverage: two small impls, and a type you invented behaves like a built-in one everywhere in the language and the prelude.

What you get without any impl

Every data type already prints structurally, and == already compares structurally, field by field. You only write `Show` when the derived form is not what you want to read, and `Ord` when there is an ordering worth having.

8.2 Traits of your own

```
# book/code/ch08/own_trait.rill
type Dog
  Dog(name: Str)

type Robot
  Robot(id: Int)

trait Speak(a)
  noise(a) -> Str
  loudness(a) -> Int

impl Speak(Dog)
  fn noise(d) = match d
    Dog(n) -> n + " says woof"
  fn loudness(d) = 7

impl Speak(Robot)
  fn noise(r) = match r
    Robot(i) -> "unit " + int_to_str(i) + " says beep"
  fn loudness(r) = 3

fn announce(x) = println(noise(x) + " (" + int_to_str(loudness(x)) + "/10)")

fn main() =
  announce(Dog("Rex"))
  announce(Robot(7))
```

```
$ rill run own_trait.rill
Rex says woof (7/10)
unit 7 says beep (3/10)
```

`trait Speak(a)` declares the interface: any type `a` that implements it provides `noise` and `loudness`. Each `impl` supplies them for one type.

`announce` is the payoff. It was never told which types exist. Its body calls `noise` and `loudness`, so its inferred type is *"for any `a` that can Speak, `a -> Unit`"*, and it works for `Dog`, for `Robot`, and for any type someone adds later. You did not write the constraint down; it came from the body, the same way `biggest` picked up its ordering requirement in Chapter 4.

Method names are global, like constructor names. There is no `dog.noise()` — `noise` is a function, and which implementation runs is decided by the type of its argument.

8.3 What "resolved at compile time" means

Dispatch in Rill is **static**. During monomorphization the compiler knows the concrete type at every call site, looks up the implementation, and emits a direct call.

The consequences are worth spelling out:

- **No vtables, no dictionaries, no type tags.** A trait call costs exactly what a normal call costs, and is usually inlined away entirely.
- **A missing implementation is a compile error**, naming the trait, the method and the type. It cannot be a runtime failure, because there is no runtime lookup to fail.

- **You cannot make a heterogeneous list** of "things that Speak". There is no boxed trait object, no `dyn Trait`, no `interface{}`. A `List(a)` has one element type.

That last point is a real limitation, and it is the trade for the first two. In Rust you would reach for `Box<dyn Speak>`; in Java, an interface reference. In Rill, when you need a collection of mixed things, you declare a sum type listing them:

```
type Speaker
  ADog(Dog)
  ARobot(Robot)
```

Which is more work up front, and closes the set — and which the compiler will then check you have handled exhaustively everywhere. Chapter 5's table of the expression problem is the right way to think about the choice.

8.4 Implementations for generic types

An `impl` may target a generic type by naming its parameters, in which case it covers every instantiation at once:

```
# book/code/ch08/generic_impl.rill
type Box(a)
  Box(a)

impl Show(Box(a))
  fn show(b) = match b
    Box(x) -> "[" + show(x) + "]"

impl Show(List(a))
  fn show(l) = "{" + join(map(l, \x -> show(x)), " ") + "}"

fn main() =
  println(Box(42))
  println(Box("hi"))
  println(Cons(1, Cons(2, Cons(3, Nil))))
  println(Box(Cons(1, Cons(2, Nil))))
  println(Cons(Box(1), Cons(Box(2), Nil)))
```

```
$ rill run generic_impl.rill
[42]
[hi]
{1 2 3}
[{1 2}]
{[1] [2]}
```

This is the listing to study. Two impls were written; five different types are printed, including two that nest one inside the other.

The mechanism is that `show(x)` **inside** the `List` impl is itself a trait call, resolved when the element type becomes known. So `List(Box(Int))` uses the list impl on the outside and the box impl on the inside, composing without any declaration that they should. In Haskell this needs a `where` clause (`instance Show a => Show [a]`); in Rill the requirement is inferred from the body and checked at monomorphization.

The rule the compiler enforces is that a generic impl must genuinely be generic. An `impl Show(List(a))` whose body only works when `a` is `Int` is rejected with a "not general enough" error, rather than silently working for one case.

Note also that the third line replaced the built-in list printing everywhere in the program — `{1 2 3}` instead of `Cons(1, Cons(2, Cons(3, Nil)))`. If the `Cons`-chains in earlier chapters have been bothering you, six lines fix them.

8.5 The whole language, in one page

That is all of it. For reference, everything Chapters 3–8 introduced:

```
import "other"                 # and: import "other" as o

type Shape                     # sum type, optionally generic
  Circle(r: Float)
  Rect(w: Float, h: Float)

trait Speak(a)                 # interface
  noise(a) -> Str

impl Speak(Shape)              # implementation
  fn noise(s) = "..."

fn area(s) = match s           # function, pattern match
  Circle(r) -> 3.14159 * r * r
  Rect(w, h) -> w * h

fn main() =
  x = 3 + 4                    # binding (immutable)
  f = \a, b -> a * b           # lambda
  if x > 5 then ... else ...   # expression
  xs |> map(area) |> sum       # pipeline
  spawn worker(ch)             # strand (Chapter 16)
```

Nine constructs. There is no `for`, no `while`, no `return`, no `null`, no exception, no class, no inheritance, no assignment, no macro, no attribute system, and no module system beyond `import`. Everything in Chapters 9 to 14 is built out of the list above and nothing else.

8.6 Exercises

1. Give `Shape` from Chapter 5 a `Show` impl that prints `circle r=1.0`, and an `Ord` impl that orders by area. Then sort a list of shapes with `max` and a fold.
2. Write a `Describe` trait with a default-ish helper: a plain function `label(x) = "<" + describe(x) + ">"` that works for every implementing type.
3. Write `impl Show(Option(a))` printing `Some 3` and `-`. Does it apply to `Option(Option(Int))`? Predict, then check.
4. Try implementing a trait for a type that does not satisfy it — call `compare` on a type with no `Ord` impl — and read the error.

Part II ends here. Chapter 9 starts Part III, where the language stops being the subject and algorithms are.

Part III

Algorithms

Chapter 9

Small Algorithms, Compared

From here on the language is assumed and the subject is programs. Each chapter takes a family of algorithms, writes them in Rill, and puts them beside the same algorithm in a language you probably know — not to score points, but because the differences are where the ideas of Chapter 1 become concrete.

This chapter does the small ones: the problems that fit on half a page and turn up in every interview and every teaching language.

9.1 FizzBuzz

```
# book/code/ch09/fizzbuzz.rill
fn word(n) =
  by3 = n % 3 == 0
  by5 = n % 5 == 0
  if by3 && by5 then "FizzBuzz" else if by3 then "Fizz" else if by5 then "Buzz" else int_to_str(n)

fn main() = println(join(map(range(1, 20), word), " "))
```

```
$ rill run fizzbuzz.rill
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17 Fizz 19 Buzz
```

The imperative version everyone writes:

```
for i in range(1, 21):
    if i % 15 == 0:   print("FizzBuzz")
    elif i % 3 == 0:  print("Fizz")
    elif i % 5 == 0:  print("Buzz")
    else:             print(i)
```

The two programs are the same length and equally clear, and anyone claiming a big win for either is selling something. But there is one real structural difference, and it is the pattern the rest of the book leans on:

The Rill version separates deciding from printing. `word` is a pure function from a number to a string. You can call it, test it, put its results in a list, join them with a comma instead of a space, or send them over a socket. The Python version has the printing welded into the decision, and getting a testable `word` out of it means rewriting it.

That is the "pure core, effectful edge" shape from Chapter 1, and at this scale it looks like pedantry. At the scale of a real program it is the difference between a codebase you can test and one you cannot.

9.2 Greatest common divisor

Euclid's algorithm is a tail recursion in its natural form, which makes it the one classic algorithm that is *shorter* in this style:

```
# book/code/ch09/gcd.rill
fn gcd(a, b) = if b == 0 then a else gcd(b, a % b)

fn lcm(a, b) = a / gcd(a, b) * b

fn main() =
  println(gcd(1071, 462))
```

```
  println(gcd(17, 5))
  println(lcm(4, 6))
  println(fold(range(1, 10), 1, lcm))
```

```
$ rill run gcd.rill
21
1
12
2520
```

Beside the imperative version:

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

These are the same algorithm, and the Rill one is what the mathematical definition says: *gcd(a, b) is a when b is zero, and gcd(b, a mod b) otherwise.* The Python version has to introduce a simultaneous assignment to avoid needing a temporary, which is a fact about Python rather than about Euclid.

The last line is worth noticing on its own:

```
fold(range(1, 10), 1, lcm)
```

The smallest number divisible by everything from 1 to 10, as a one-liner, because `lcm` is exactly the kind of two-argument function `fold` wants. Once you have the combinators, "combine a list with this operation" stops being a loop you write and becomes a phrase you say.

9.3 Primes, two ways

```
# book/code/ch09/primes.rill
fn divides(d, n) = n % d == 0

fn no_factor_from(n, d) =
  d * d > n || (if divides(d, n) then false else no_factor_from(n, d + 1))

fn is_prime(n) = n >= 2 && no_factor_from(n, 2)

fn sieve(l) = match l
  Nil -> Nil
  Cons(p, rest) -> Cons(p, sieve(filter(rest, \x -> x % p != 0)))

fn main() =
  println(filter(range(1, 50), is_prime))
  println(sieve(range(2, 50)))
  println(len(filter(range(2, 10000), is_prime)))
```

```
$ rill run primes.rill
Cons(2, Cons(3, Cons(5, Cons(7, ... Cons(47, Nil)))))
Cons(2, Cons(3, Cons(5, Cons(7, ... Cons(47, Nil)))))
1229
```

Two different algorithms, same answer.

Trial division (`is_prime`) tests each candidate on its own. The loop is `no_factor_from`, tail recursive, with `d` as the loop variable and `d * d > n` as the exit — the usual "stop at the square root" without ever computing a square root.

> **That line was written differently**

> It was first written `if d * d > n then true else if divides(d, n) then false else ...`, and `rill fmt` rewrote it to the `||` above. The two are the same program: `||` in Rill is parser sugar that desugars to exactly that `if`, and the formatter recovers the operator when it sees the shape. It is a small reminder of what canonical formatting is for — there is one spelling, and the tool knows it even when you do not.

The sieve (`sieve`) is the one worth studying. Read it as a sentence: *the sieve of a list is its first element, followed by the sieve of everything in the rest that the first element does not divide.* That is four words of recursion and one `filter`, and it is the whole of Eratosthenes' idea.

```
# the imperative sieve
def sieve(n):
    mark = [True] * (n + 1)
    for p in range(2, int(n ** 0.5) + 1):
        if mark[p]:
            for m in range(p * p, n + 1, p):
                mark[m] = False
    return [i for i in range(2, n + 1) if mark[i]]
```

Here the comparison is genuinely uneven, and in both directions.

The Rill version is clearer: no array, no bounds, no nested loop, no off-by-one. It is a definition rather than a procedure.

The Python version is much faster. The array sieve crosses off multiples in place, which is O(n log log n) over contiguous memory. The functional sieve builds a new list at every step and filters it, which is closer to $O(n^2/\log n)$ with allocation on top. This is precisely the cost Chapter 1 warned about: *some algorithms genuinely want mutation*, and a sieve is the textbook one.

The honest conclusion is that you should write the first version for clarity, and if the sieve turns out to be your bottleneck, reach for a `Buf` (Chapter 15) and write the array version — which Rill can also express.

9.4 Collatz: a loop with a maximum

```
# book/code/ch09/collatz.rill
fn step(n) = if n % 2 == 0 then n / 2 else 3 * n + 1

fn length_from(n, acc) = if n == 1 then acc else length_from(step(n), acc + 1)

fn chain(n) = length_from(n, 0)

fn best(l, bn, bl) = match l
  Nil -> Cons(bn, Cons(bl, Nil))
  Cons(n, rest) -> if chain(n) > bl then best(rest, n, chain(n)) else best(rest, bn, bl)

fn main() =
  println(chain(27))
  println(best(range(1, 10000), 1, 0))
```

```
$ rill run collatz.rill
111
Cons(6171, Cons(261, Nil))
```

Two loops of different shapes, which is why this problem is here.

`length_from` is a plain accumulator loop: one number in, one number out, exit when it reaches 1.

`best` is an **argmax** — a fold that carries two pieces of state, the best input seen and its score. This is the pattern that a plain `fold` handles awkwardly (you would need a pair type, which Rill 0.2 has no literal for) and that explicit recursion handles naturally. It is the first case in this book where writing the recursion yourself is the better choice.

There is a bug-shaped inefficiency in `best` worth spotting: `chain(n)` is called twice on the winning branch, and `chain` is not cheap. Fixing it means computing it once and passing it along.

The obvious fix — binding `c` inside the arm — does not compile, and it is worth showing the error because it is the roughest edge in the language:

```
collatz.rill: syntax error at line 6, column 5: expected an expression, found indent
```

A `match` arm's body cannot be an indented block. So the binding moves into a helper function instead:

```
# book/code/ch09/collatz_fixed.rill
fn best(l, bn, bl) = match l
  Nil -> Cons(bn, Cons(bl, Nil))
  Cons(n, rest) -> best_on(rest, n, chain(n), bn, bl)

fn best_on(rest, n, c, bn, bl) = if c > bl then best(rest, n, c) else best(rest, bn, bl)
```

```
$ rill run collatz_fixed.rill
Cons(6171, Cons(261, Nil))
```

Same answer, and `chain` is now called once per candidate instead of twice. Chapter 12 hits this limitation again in a parser, where it costs three extra functions.

Note in passing that hoisting the call out like this is *always* safe here. In an impure language you could not be sure — the second call might return something different, or do something. That is referential transparency being useful rather than being a definition.

9.5 Digits and palindromes

```
# book/code/ch09/digits.rill
fn digits_of(n) = if n < 10 then Cons(n, Nil) else append(digits_of(n / 10), Cons(n % 10, Nil))

fn is_palindrome(n) =
  d = digits_of(n)
  d == reverse(d)

fn main() =
  println(digits_of(9021))
  println(sum(digits_of(9021)))
  println(is_palindrome(12321))
  println(is_palindrome(12320))
  println(filter(range(100, 140), is_palindrome))
```

```
$ rill run digits.rill
Cons(9, Cons(0, Cons(2, Cons(1, Nil))))
12
true
false
Cons(101, Cons(111, Cons(121, Cons(131, Nil))))
```

`d == reverse(d)` deserves a moment. That is **structural equality on a data type**, working for free: no `equals` method, no `__eq__`, no comparator. Rill compares two lists element by element because that is what equality on a data value means, and the same line would work for a tree or any other type you declare.

In Java that expression is `d.equals(reversed)` and only if someone wrote `equals` correctly. In C it is a loop. In Python it works, because Python's lists do the same thing — which is a good reminder that structural equality is not a functional-programming idea, just a good one.

`digits_of` also shows the cost of building a list in the wrong direction. `append(digits_of(n / 10), ...)` walks the whole accumulated list on every digit, so it is $O(d^2)$ in the number of digits. For a 4-digit number that is nothing; the exercise at the end asks you to fix it, and the fix is the accumulator pattern from Chapter 7.

9.6 What these five have in common

Looking back at the chapter, three habits appear in every listing:

A loop is a function with the loop variables as parameters. `no_factor_from`, `length_from`, `best` — each one is a `while` turned inside out, and each one is tail recursive so it costs nothing.

The recursive definition of a problem is often the program. The sieve and Euclid's algorithm are the clearest examples: the code and the mathematical statement are the same sentence.

Deciding is separated from doing. `word`, `is_prime`, `chain` and `is_palindrome` are pure functions that return values. Nothing prints except `main`.

And one warning, which the sieve made concrete: **an elegant functional algorithm is sometimes asymptotically worse than the array version**. It is not always, and the cases where it is are predictable — they involve in-place updates of a large structure. Knowing which you are in is part of the job.

9.7 Exercises

1. Rewrite `digits_of` with an accumulator so it is linear. Check it against the original for a few hundred numbers.
2. Write `is_perfect(n)` — a number equal to the sum of its proper divisors — and find all of them below 10,000.
3. Write the Collatz `best` with a `fold` by declaring a two-field type to carry the pair. Which version do you prefer, and why?
4. Write `to_binary(n)` returning a `Str`. Then `from_binary(s)`, and check that they round-trip.
5. Time `len(filter(range(2, 100000), is_prime))` against the functional sieve at the same limit. Predict the ratio before you measure it.

Chapter 10 takes the first algorithm where the functional and imperative versions really do diverge: sorting.

Chapter 10

Sorting

Sorting is the standard place to compare programming styles, because everyone knows the algorithms and the differences are stark. It is also the standard place for functional programming to be oversold — the famous two-line quicksort is *not* quicksort — so this chapter is careful about what is being claimed.

Rill has no sort in its prelude. By the end of this chapter you will have written three.

10.1 Insertion sort

Start with the one that is genuinely nicer functionally:

```
# book/code/ch10/sorts.rill
fn insert(x, l) = match l
  Nil -> Cons(x, Nil)
  Cons(y, rest) -> if x <= y then Cons(x, Cons(y, rest)) else Cons(y, insert(x, rest))

fn insertion_sort(l) = fold(l, Nil, \acc, x -> insert(x, acc))
```

```
$ rill run sorts.rill
Cons(1, Cons(2, Cons(3, Cons(5, Cons(7, Cons(8, Cons(9, Nil)))))))
```

`insert` puts one element into an already-sorted list: if it belongs before the head, it goes at the front; otherwise it goes somewhere in the tail. Two lines, and both are the definition read aloud.

`insertion_sort` is then a **fold** — start with the empty list and insert each element. The entire sort is one line, because the recursive work lives in `insert` where it belongs.

Beside the imperative version:

```
def insertion_sort(a):
    for i in range(1, len(a)):
        key = a[i]
        j = i - 1
        while j >= 0 and a[j] > key:
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key
```

Every index in that function is an opportunity to be wrong by one, and the shifting loop has to run backwards for a reason that takes a paragraph to explain. The functional version has no indices at all. This is the clearest win in the chapter, and it is not a coincidence: insertion sort is naturally expressed as "build a sorted result", and the array version is only shifting things because it insists on using no extra memory.

Both are $O(n^2)$. Neither should sort a large list.

10.2 The quicksort that is not quicksort

This is the program that appears in every functional-language tutorial:

```
fn quicksort(l) = match l
  Nil -> Nil
  Cons(p, rest) -> append(quicksort(filter(rest, \x -> x < p)), Cons(p, quicksort(filter(rest, \x -> x >= p))))
```

```
Cons(1, Cons(2, Cons(3, Cons(5, Cons(7, Cons(8, Cons(9, Nil)))))))
```

It is beautiful, it is correct, and it is important to be honest about it: **this is not quicksort.**

Real quicksort is defined by in-place partitioning. Its entire reason for existing is that it sorts with O(log n) extra memory and excellent cache behaviour, which is why it beats merge sort in practice despite the same average complexity. The program above:

- allocates a new list at every level, so it uses O(n log n) extra memory, not O(log n);
- walks `rest` **twice** at every level, once per `filter`;
- calls `append`, which walks the left result again;
- and has no cache locality at all, because a linked list is scattered.

What it is, precisely, is *a description of the divide-and-conquer idea behind quicksort*, and it is excellent for that. Just do not put it in a benchmark and claim a language win. The definitive discussion is a paper by Ben Lynn titled "Quicksort is not quicksort", and everyone who has ever written this listing in a tutorial should read it.

It also inherits real quicksort's worst case: an already-sorted input picks the smallest element as pivot every time and degrades to $O(n^2)$. The duplicate test in the listing (`quicksort of 2 2 1 2`) checks the other classic bug, which is that `<` and `>=` must partition *all* the remaining elements between them.

10.3 Merge sort, which is the one to use

Merge sort is the algorithm that suits immutable linked lists, for a reason worth stating: it never needs to look at an element more than once per level, and merging two sorted lists is a natural list operation.

```
fn merge(a, b) = match a
  Nil -> b
  Cons(x, xs) -> match b
    Nil -> a
    Cons(y, ys) -> if x <= y then Cons(x, merge(xs, b)) else Cons(y, merge(a, ys))

fn halve(l, n) = Cons(take(l, n / 2), Cons(drop(l, n / 2), Nil))

fn merge_sort(l) =
  n = len(l)
  if n <= 1 then l else merge_on(halve(l, n))

fn merge_on(parts) = match parts
  Cons(a, Cons(b, _)) -> merge(merge_sort(a), merge_sort(b))
  _ -> Nil
```

```
Cons(1, Cons(2, Cons(3, Cons(5, Cons(7, Cons(8, Cons(9, Nil)))))))
Nil
Cons(1, Nil)
```

`merge` is the heart of it, and it is the same nested-match shape as everything else in this book: two lists, four cases, take whichever head is smaller.

`halve` returns two lists in a list, because Rill 0.2 has no tuples. That is an ergonomic wart and worth naming as one — in OCaml this would be `(take l (n/2), drop l (n/2))` and the caller would destructure it with `let (a, b) = ...`. Here it costs a helper function, `merge_on`, whose only job is to take the two-element list apart. The alternative is declaring a `type Pair(a, b)  Pair(a, b)`, which is what a real program should do.

Complexity. O(n log n) comparisons, and unlike the quicksort listing it is guaranteed rather than average — merge sort has no bad input. It allocates O(n log n) cells in total, but only O(n) are live at once because the earlier levels die as soon as they are merged, and Chapter 15 explains why "die" means "freed immediately" here.

Stability. `if x <= y` takes from the left list on a tie, which makes this sort stable. Change it to `<` and it stops being stable. That one character is the entire difference, and it is worth knowing which one you wrote.

Beside the imperative merge sort — which needs an auxiliary array, index arithmetic for three pointers, and a copy-back step — the functional version is the clearer program by some distance. This is the fair comparison to make, and it is the one that tutorials should be making instead of the quicksort one.

10.4 Sorting by something other than <

Real sorting is rarely on the values themselves. Here is a comparator-driven version, with a `Show` impl so the output is readable:

```
# book/code/ch10/sortby.rill
type Person
  Person(name: Str, age: Int)

impl Show(Person)
  fn show(p) = match p
    Person(n, a) -> n + "(" + int_to_str(a) + ")"

impl Show(List(a))
  fn show(l) = "[" + join(map(l, \x -> show(x)), " ") + "]"

fn sort_by(l, before) = match l
  Nil -> Nil
  Cons(p, rest) -> append(sort_by(filter(rest, \x -> before(x, p)), before), Cons(p, sort_by(filter(rest, \x ->

fn age_of(p) = match p
  Person(_, a) -> a

fn name_of(p) = match p
  Person(n, _) -> n

fn main() =
  people = Cons(Person("Ada", 36), Cons(Person("Bob", 24), Cons(Person("Cyd", 31), Nil)))
  println(sort_by(people, \a, b -> age_of(a) < age_of(b)))
  println(sort_by(people, \a, b -> name_of(a) > name_of(b)))
```

```
$ rill run sortby.rill
[Bob(24) Cyd(31) Ada(36)]
[Cyd(31) Bob(24) Ada(36)]
```

`sort_by` takes the ordering as a **function**, and the two calls in `main` pass two different lambdas — one ascending by age, one descending by name. This is Chapter 4's higher-order functions doing the job they exist for. No interface to implement, no comparator class, no `Comparable`; a two-argument function returning a `Bool`.

The two `Show` impls are Chapter 8 earning its place: six lines, and the output went from `Cons(Person(Ada, 36), ...)` to `[Bob(24) Cyd(31) Ada(36)]`.

> **Or use `Ord` instead**
>
> If a type has one natural ordering, implement `Ord` for it (Chapter 8) and every sort written against `<` accepts it directly. Use a comparator argument when the ordering is a property of the *call*, not of the type — which is why the same list is sorted two ways above.

10.5 The comparison, honestly

	Rill (linked lists, immutable)	C/Rust/Java (arrays, in place)
Insertion sort	clearly nicer — no indices	index arithmetic, shifting loop
Merge sort	clearly nicer — `merge` is four cases	auxiliary array, three cursors, copy-back

	Rill (linked lists, immutable)	C/Rust/Java (arrays, in place)
Quicksort	not the real algorithm	this is where the array wins outright
Extra memory	O(n) live, O(n log n) allocated	O(log n) for quicksort
Cache behaviour	poor — pointers everywhere	excellent — contiguous
Sorting by a key	pass a lambda	pass a comparator, or an interface
Getting it wrong	usually a type error	usually an off-by-one, at run time

The summary is that functional sorting is **easier to write correctly and harder to make fast**. For a few thousand elements, which is what most programs actually sort, the difference does not matter and the clearer program wins. For a million elements in a hot loop, you want contiguous memory and in-place partitioning, and in Rill that means a `Buf` and Chapter 15.

The mistake to avoid is deciding this in the abstract. Write `merge_sort`, measure it, and only reach for the array if the measurement asks you to.

10.6 Exercises

1. `merge_sort` calls `len`, `take` and `drop`, walking the list three times just to split it. Write a `split_alternating` that produces both halves in one pass by dealing elements left, right, left, right.
2. Make the quicksort listing partition in a single pass instead of two `filter`s. You will need to return two lists — declare a `Pair` type.
3. Write `is_sorted(l)` and use it to check all three sorts on a few hundred random-ish inputs (generate them with a linear congruential generator — Chapter 13 has one).
4. Add a third sort: `bottom_up_merge_sort`, which starts with a list of one-element lists and repeatedly merges neighbouring pairs. No recursion on halves, and it is the version real libraries use for linked lists.
5. Which of the three sorts in §10.1–10.3 are stable? Prove it by sorting a list of `Person` by age where two people share an age.

Chapter 11 moves from sequences to trees, where recursive types stop being a teaching device and start being the obviously right tool.

Chapter 11

Trees

A tree is where recursive types stop being a teaching example. Lists are recursive but they are also just sequences, and every language has those. A tree is where the shape of the data and the shape of the code line up so exactly that writing the second is mostly a matter of reading the first.

This chapter also answers the question Chapter 6 left open — what do you do when you need a map and the standard library has none.

11.1 The type

```
type Tree(a)
  Leaf
  Node(left: Tree(a), value: a, right: Tree(a))
```

A tree is empty, or a node with a left subtree, a value, and a right subtree.

Compare that with the equivalent in a language without sum types:

```
class Node<T> {
    Node<T> left;      // may be null
    T value;
    Node<T> right;     // may be null
}
```

The Java version has no way to say "empty tree" except `null`, so every method starts by checking for it and every field access is a potential `NullPointerException`. `Leaf` is a value, costs no allocation (Chapter 15 explains why nullary constructors are singletons), and the compiler makes you handle it.

11.2 A search tree, whole

```
# book/code/ch11/tree.rill
fn insert(t, x) = match t
  Leaf -> Node(Leaf, x, Leaf)
  Node(l, v, r) -> if x < v then Node(insert(l, x), v, r) else if x > v then Node(l, v, insert(r, x)) else t

fn member(t, x) = match t
  Leaf -> false
  Node(l, v, r) -> if x == v then true else if x < v then member(l, x) else member(r, x)

fn to_list(t) = match t
  Leaf -> Nil
  Node(l, v, r) -> append(to_list(l), Cons(v, to_list(r)))

fn size(t) = match t
  Leaf -> 0
  Node(l, _, r) -> 1 + size(l) + size(r)

fn height(t) = match t
  Leaf -> 0
  Node(l, _, r) -> 1 + max(height(l), height(r))

fn from_list(l) = fold(l, Leaf, insert)
```

```
fn main() =
  t = from_list(Cons(5, Cons(3, Cons(8, Cons(1, Cons(9, Cons(3, Nil))))))
  println(to_list(t))
  println(size(t))
  println(height(t))
  println(member(t, 8))
  println(member(t, 4))
  println(to_list(from_list(range(1, 7))))
  println(height(from_list(range(1, 7))))
```

```
$ rill run tree.rill
Cons(1, Cons(3, Cons(5, Cons(8, Cons(9, Nil)))))
5
3
true
false
Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Cons(7, Nil)))))))
7
```

Six functions, and every one of them has the same two-case shape: what to do for `Leaf`, what to do for `Node`. That is not a style — it is the type dictating the code. When the data has two cases the function has two cases, and the compiler checks you wrote both.

Three things in the output are worth reading closely.

`to_list` gives `1 3 5 8 9` — sorted. An in-order walk of a search tree produces its contents in order, which means `to_list(from_list(xs))` is a sort. It is O(n log n) if the tree is balanced, and it does not survive the next observation.

`size` is 5, but six numbers went in. The duplicate `3` hit the `else t` branch of `insert` and was dropped. This tree is a *set*, and that was a design decision made in one word.

`height(from_list(range(1, 7)))` is 7. Inserting 1, 2, 3, 4, 5, 6, 7 in order gives a tree where every node has an empty left subtree — a linked list wearing a tree costume. Every operation is O(n), and this is the classic weakness of an unbalanced search tree, in every language.

Fixing it means rebalancing, and the standard answer in this style is a **red-black tree**, whose functional insertion is a famous half-page of pattern matching by Chris Okasaki. It is beyond this chapter, but it is worth knowing that the functional version is dramatically shorter than the imperative one — rotations become patterns rather than pointer surgery. Okasaki's *Purely Functional Data Structures* is the book, and it is the natural sequel to this chapter.

11.3 Where the immutability pays

```
# book/code/ch11/sharing.rill
fn main() =
  a = fold(Cons(5, Cons(3, Cons(8, Nil))), Leaf, insert)
  b = insert(a, 7)
  println(to_list(a))
  println(to_list(b))
```

```
$ rill run sharing.rill
Cons(3, Cons(5, Cons(8, Nil)))
Cons(3, Cons(5, Cons(7, Cons(8, Nil))))
```

`b` has the 7; `a` does not. Both are valid, usable trees, and `insert` did not modify anything.

Now the part that answers Chapter 1's objection about copying. Inserting 7 into this tree did **not** copy it. Look at what `insert` builds:

```
Node(l, v, insert(r, x))
      ^  ^
      └──┴── the untouched left subtree and value, reused as-is
```

Only the nodes **on the path from the root to the new leaf** are rebuilt. Everything hanging off that path is shared between `a` and `b` — the same memory, pointed at by both. For a balanced tree of a million nodes, inserting one element allocates about twenty nodes, not a million.

This is what "persistent data structure" means, and it is only safe because nothing can change. If `a` could be modified, `b` sharing its subtrees would be a catastrophic aliasing bug — the exact bug in Chapter 1's `add_tag` example. Immutability is what converts sharing from a hazard into the default.

The practical consequences are large: you can keep the old version for undo, hand a snapshot to another strand with no lock and no copy, or keep every historical version of a structure at the cost of the deltas.

11.4 A map, since the prelude has none

Rill 0.2 has no dictionary type. Here is one, as a search tree of key-value entries:

```
# book/code/ch11/map.rill
type Entry(k, v)
  Entry(key: k, value: v)

type Dict(k, v)
  Empty
  Branch(left: Dict(k, v), item: Entry(k, v), right: Dict(k, v))

fn put(d, k, v) = match d
  Empty -> Branch(Empty, Entry(k, v), Empty)
  Branch(l, e, r) -> match e
    Entry(ek, _) -> if k < ek then Branch(put(l, k, v), e, r) else if k > ek then Branch(l, e, put(r, k, v)) els

fn get(d, k) = match d
  Empty -> None
  Branch(l, e, r) -> match e
    Entry(ek, ev) -> if k == ek then Some(ev) else if k < ek then get(l, k) else get(r, k)

fn count_words(d, l) = match l
  Nil -> d
  Cons(w, rest) -> count_words(put(d, w, unwrap_or(get(d, w), 0) + 1), rest)

fn main() =
  words = split("the cat sat on the mat the end", " ")
  d = count_words(Empty, words)
  println(get(d, "the"))
  println(get(d, "cat"))
  println(get(d, "dog"))
```

```
$ rill run map.rill
Some(3)
Some(1)
None
```

Four things this listing demonstrates at once.

Two type parameters. `Dict(k, v)` is generic in both, and `Dict(Str, Int)` comes out of the use with no annotation anywhere.

`get` returns `Option`. A missing key is `None`, not an exception and not a zero that might have been a real value. `get(d, "dog")` is `None` and the caller must say what that means.

`unwrap_or(get(d, w), 0) + 1` is the word-count idiom: look up, default to zero, add one, put back. In Python that is `counts[w] = counts.get(w, 0) + 1` — the same sentence, and the difference is that `put` returns a new dictionary rather than modifying one.

`count_words` threads the dictionary through the recursion. This is the functional answer to "a loop that updates a variable": the variable becomes a parameter, and each step passes the updated version to the next. It is tail recursive, so it is a loop.

11.4.1 What this costs

A tree map is O(log n) for lookup where a hash map is O(1), and the constant factor is worse because it chases pointers. It is also unbalanced here, so inserting sorted keys degrades it to a list — inserting words in alphabetical order would be O(n) per operation.

For counting the words in a sentence, none of that matters. For a hot lookup table with a million entries, it matters a great deal, and the answer is a hash table over a `Buf` (Chapter 15) or a balanced tree. Knowing that you are making this trade is the point; making it silently is the problem.

11.5 Trees are everywhere once you look

The shape in this chapter — a recursive sum type, and functions that match its cases — is not really about search trees. It is the shape of:

- **an expression** (`Add(Mul(Num(2), Num(3)), Num(4))`), which Chapter 12 parses and evaluates;
- **a document** (a heading contains paragraphs contain spans);
- **a file system** (a directory contains files and directories);
- **a decision** (a game tree, a behaviour tree);
- **JSON**, which is a five-case sum type and nothing else.

In every one of them the same two rules apply: the type has a case per shape, and each function has an arm per case. Once that reflex is trained, a surprising number of "hard" programs turn out to be a type declaration and four short functions.

11.6 Exercises

1. Write `remove(t, x)`. The hard case is a node with two children — the usual answer is to replace it with the smallest value in its right subtree.
2. Write `fold_tree(t, init, f)` that folds the values in order, and redefine `size`, `sum` and `to_list` in terms of it.
3. `to_list` uses `append`, which makes it O(n^2) on a degenerate tree. Rewrite it with an accumulator so it is linear.
4. Add `keys(d)` and `values(d)` to the dictionary, and `merge(a, b)` where `b`'s entries win.
5. Write `depth_of(t, x)` returning `Option(Int)` — how far down a value is, or `None` if it is not there.
6. Model JSON as a type: null, boolean, number, string, array of values, object of key-value pairs. Write `render(json) -> Str`. (`examples/json/json.rill` in this repository is a full parser and printer, if you want to compare.)

Chapter 12 takes the tree that a program's text turns into, and builds it from characters.

Chapter 12

Parsing

Parsing is the problem functional languages were invented for. ML was built to write proof tactics; Haskell's early users wrote compilers; the reason sum types, pattern matching and recursion feel so natural here is that someone designed them for exactly this shape of work.

This chapter builds a complete expression calculator: a type for the syntax tree, an evaluator, and a recursive-descent parser with error reporting. It is the longest program in the book and it is about 60 lines.

12.1 The tree first

Before parsing anything, decide what you are parsing *into*:

```
# book/code/ch12/calc.rill
type Expr
  Num(Int)
  Add(Expr, Expr)
  Sub(Expr, Expr)
  Mul(Expr, Expr)

impl Show(Expr)
  fn show(e) = match e
    Num(n) -> int_to_str(n)
    Add(a, b) -> "(" + show(a) + " + " + show(b) + ")"
    Sub(a, b) -> "(" + show(a) + " - " + show(b) + ")"
    Mul(a, b) -> "(" + show(a) + " * " + show(b) + ")"

fn eval(e) = match e
  Num(n) -> n
  Add(a, b) -> eval(a) + eval(b)
  Sub(a, b) -> eval(a) - eval(b)
  Mul(a, b) -> eval(a) * eval(b)

fn main() =
  e = Add(Mul(Num(2), Num(3)), Sub(Num(10), Num(4)))
  println(e)
  println(eval(e))
```

```
$ rill run calc.rill
((2 * 3) + (10 - 4))
12
```

An abstract syntax tree is a recursive sum type, and that is all it is. Four cases, and `Add`, `Sub` and `Mul` hold two `Exprs` each — the recursion in the type is what makes arbitrarily deep expressions representable.

`eval` is then four lines that could not be much else. *The value of a number is the number; the value of an addition is the sum of the values.* Each arm of the match is one line of the language's semantics, which is why interpreters in this style read like specifications.

`show` is the same walk producing text instead of a number, and it prints the tree fully bracketed — useful, because it shows you what the parser decided about precedence.

This is the visitor pattern, without the pattern

In a language with classes, an AST is a class hierarchy and every operation over it is a Visitor: an interface with a method per node type, an `accept` method on each node, and double dispatch to make it work. That is perhaps eighty lines of ceremony to express what `match` expresses in four.

The trade is Chapter 5's table. Adding a new *operation* here is one new function; adding a new *node type* means the compiler lists every match that needs an arm. For a language's AST — where the node types are fixed by the grammar and the operations keep coming — that is exactly the right way round.

12.2 The parser

The grammar is the usual three-level one, which encodes precedence in its shape:

```
expr := term (("+" | "-") term)*
term := atom ("*" atom)*
atom := number | "(" expr ")"
```

`*` binds tighter than `+` because `term` is *inside* `expr`. Parentheses come back to the top by having `atom` call `expr` again.

The key design decision in a functional parser is what a parse *returns*. It cannot mutate a position counter, so it must hand one back:

```
type Parsed
  Parsed(value: Expr, next: Int)
  Bad(why: Str, at: Int)
```

A parse either produced a value and consumed up to position `next`, or failed with a reason at a position. That is `Result` with the success case carrying two things — and writing it as its own type instead of reusing `Result` keeps the field names.

Here is the whole parser:

```
# book/code/ch12/parser.rill
fn digit(c) = c >= 48 && c <= 57

fn skip_space(s, i) = if i < str_len(s) && str_get(s, i) == 32 then skip_space(s, i + 1) else i

fn read_int(s, i, acc) =
  if i < str_len(s) && digit(str_get(s, i)) then read_int(s, i + 1, acc * 10 + str_get(s, i) - 48) else Parsed(N

# atom := number | "(" expr ")"
fn atom(s, i0) =
  i = skip_space(s, i0)
  if i >= str_len(s) then Bad("expected a number", i) else if digit(str_get(s, i)) then read_int(s, i, 0) else i

fn bracketed(s, i) = match expr(s, i)
  Bad(w, at) -> Bad(w, at)
  Parsed(e, j) -> closing(s, e, skip_space(s, j))

fn closing(s, e, j) =
  if j < str_len(s) && str_get(s, j) == 41 then Parsed(e, j + 1) else Bad("expected )", j)

# term := atom ("*" atom)*
fn term(s, i) = match atom(s, i)
  Bad(w, at) -> Bad(w, at)
  Parsed(a, j) -> term_more(s, a, j)

fn term_more(s, a, j0) =
  j = skip_space(s, j0)
  if j < str_len(s) && str_get(s, j) == 42 then term_mul(s, a, j + 1) else Parsed(a, j)

fn term_mul(s, a, j) = match atom(s, j)
```

```
  Bad(w, at) -> Bad(w, at)
  Parsed(b, k) -> term_more(s, Mul(a, b), k)

# expr := term (("+" | "-") term)*
fn expr(s, i) = match term(s, i)
  Bad(w, at) -> Bad(w, at)
  Parsed(a, j) -> expr_more(s, a, j)

fn expr_more(s, a, j0) =
  j = skip_space(s, j0)
  plus = j < str_len(s) && str_get(s, j) == 43
  minus = j < str_len(s) && str_get(s, j) == 45
  if plus || minus then expr_op(s, a, j + 1, plus) else Parsed(a, j)

fn expr_op(s, a, j, plus) = match term(s, j)
  Bad(w, at) -> Bad(w, at)
  Parsed(b, k) -> expr_more(s, if plus then Add(a, b) else Sub(a, b), k)
```

```
$ rill run parser.rill
2 + 3 * 4  ->  (2 + (3 * 4)) = 14
(2 + 3) * 4  ->  ((2 + 3) * 4) = 20
10 - 4 - 3  ->  ((10 - 4) - 3) = 3
2 * (3 + (4 - 1))  ->  (2 * (3 + (4 - 1))) = 12
2 +  ->  error: expected a number at 3
(1 + 2  ->  error: expected ) at 6
```

The output confirms the two things a parser must get right. `2 + 3 * 4` parsed as `(2 + (3 * 4))`, so precedence works. `10 - 4 - 3` parsed as `((10 - 4) - 3)` and evaluated to 3, so subtraction is left-associative — if it had built `(10 - (4 - 3))` you would see 9.

12.2.1 Reading it

Each grammar rule is a function. `expr`, `term`, `atom` — the code and the grammar are the same three lines. This is recursive descent, and it is the reason this style of parser is the one people write by hand.

The `("+" term)*` repetition is a tail-recursive helper. `expr_more` is the loop: it looks for an operator, and if it finds one it parses another term, builds a bigger tree, and calls itself. `a` — the tree built so far — is the accumulator. This is Chapter 7's pattern doing real work, and it is also why the result comes out left-associative for free.

Errors propagate by pattern matching. Every call site has a `Bad` arm that passes the failure up unchanged. That is more typing than an exception, and it is the visible cost of Chapter 5's decision. In exchange there is no hidden control flow: every path out of every function is on the page.

The position is threaded, never mutated. `i`, `j`, `k` are three different positions, each one returned by the previous step. A mutable parser has one `self.pos` that everything shares, which is convenient right up to the moment you want to backtrack — at which point you need to save and restore it, and forgetting to is the classic parser bug. Here backtracking is free: the old position is still in scope because nothing overwrote it.

12.2.2 The awkward part

`bracketed` had to be split into two functions, and `term_more`/`term_mul` into two more. The reason is a real limitation of Rill 0.2: **a `match` arm's body cannot be an indented block on the following lines**. Where you want

```
Parsed(e, j0) ->
  j = skip_space(s, j0)
  ...
```

you must instead pass the intermediate value to a helper. The compiler is clear about it —

```
parser.rill: syntax error at line 41, column 5: expected an expression, found indent
```

— but it does push a parser into more small functions than the grammar strictly needs. In OCaml or Haskell this would be a `let ... in` inside the branch. It is the single roughest edge encountered in writing this book.

12.3 How other languages do this

```python
# Python, mutable position
class Parser:
    def __init__(self, s):
        self.s, self.i = s, 0

    def expr(self):
        a = self.term()
        while self.peek() in "+-":
            op = self.take()
            b = self.term()
            a = Add(a, b) if op == "+" else Sub(a, b)
        return a
```

Shorter, and most people find it easier to write the first time. Three differences are worth naming:

State. `self.i` is shared mutable state. Every method may move it, and whether a failed parse leaves it where it started is a matter of discipline. The Rill version cannot get this wrong, because a function that fails returns `Bad` and the caller still holds the old position.

Errors. Python raises; the type says nothing. Rill returns `Bad`; every caller is forced by the compiler to decide what to do about it.

Exhaustiveness. Add a node type to the AST in Python and `eval` silently falls through to whatever the last `elif` was. In Rill it does not compile.

The honest summary: for a hundred-line calculator the Python version is faster to write. For a language front end that will be maintained for years and grow node types, the property that *adding a case makes the compiler list every place that must change* is worth much more than the lines it costs.

Real parser combinator libraries — Haskell's Parsec, Rust's nom — take this chapter's `Parsed` type and make it composable, so that `many`, `optional` and `choice` become functions and the grammar becomes an expression. That needs a type for "a parser" as a first-class value, which Rill can express; it is the natural next step from here and the last exercise asks for its first piece.

12.4 Exercises

1. Add division and unary minus. Division by zero should produce a `Bad` rather than crashing — which means `eval` must return a `Result` too, and that change ripples exactly as far as the compiler tells you.
2. Add floating-point numbers, so `2.5 * 4` works. Where does `read_int` become two functions?
3. Add variables and a `let`: `let x = 3 in x * x`. `eval` now needs an environment — thread a `Dict` from Chapter 11 through it.
4. Report the error with a caret under the offending column, using the position in `Bad`.
5. Write `simplify(e)` that rewrites `Add(Num(0), x)` to `x`, `Mul(Num(1), x)` to `x`, and `Mul(Num(0), _)` to `Num(0)`, recursively. This is a compiler optimization pass, and it is fifteen lines of pattern matching.
6. Make `Parsed` generic — `Parsed(a)` for any result type — so the same machinery can parse something that is not an `Expr`.

Chapter 13 leaves trees for graphs, where the data no longer has a shape the type system can enforce and the algorithms get more interesting.

Chapter 13

Graphs and Grids

Trees are the friendly case: the type enforces the shape, and every node has one parent. A graph has neither property. It has cycles, so a naive recursion runs forever; it has no root, so there is no obvious place to start; and it has no shape a sum type can capture.

This is where the functional style stops being obviously better and starts requiring judgement. This chapter shows both answers — the list-based one that is short and slow, and the array-based one that is longer and fast — and is explicit about when to reach for which.

13.1 A graph as a list of edges

The simplest representation needs no new machinery at all:

```
# book/code/ch13/graph.rill
type Edge
  Edge(from: Int, to: Int)

fn edge_from(e) = match e
  Edge(a, _) -> a

fn edge_to(e) = match e
  Edge(_, b) -> b

fn neighbours(edges, n) = map(filter(edges, \e -> edge_from(e) == n), edge_to)
```

`neighbours` is a `filter` followed by a `map`: keep the edges leaving `n`, then take where each one goes. Two combinators, one line, and no data structure beyond a list.

13.1.1 Breadth-first search

```
fn seen(l, n) = any(l, \v -> v == n)

fn bfs(edges, frontier, visited) = match frontier
  Nil -> reverse(visited)
  Cons(n, rest) -> bfs_on(edges, n, rest, visited)

fn bfs_on(edges, n, rest, visited) =
  fresh = filter(neighbours(edges, n), \m -> !seen(visited, m) && !seen(rest, m))
  bfs(edges, append(rest, fresh), append(reverse(fresh), visited))

fn reachable(edges, start) = bfs(edges, Cons(start, Nil), Cons(start, Nil))

fn main() =
  g = Cons(Edge(1, 2), Cons(Edge(1, 3), Cons(Edge(2, 4), Cons(Edge(3, 4), Cons(Edge(4, 5), Cons(Edge(6, 7), Nil)
  println(neighbours(g, 1))
  println(reachable(g, 1))
  println(reachable(g, 6))
  println(reachable(g, 5))
```

```
$ rill run graph.rill
[2 3]
[1 2 3 4 5]
[6 7]
```

```
[5]
```

The classic BFS has a mutable queue and a mutable visited-set. Here both are parameters:

- **`frontier`** is the queue. Taking from the front is `Cons(n, rest)`; adding to the back is `append(rest, fresh)`.
- **`visited`** is the set. Membership is `any`, and marking is `Cons`.

The termination argument is the same as the imperative one and easier to see: a node enters `frontier` only if it is in neither `visited` nor `frontier`, so each node is enqueued at most once and the frontier must eventually empty. The cycle 1→2→4 and 1→3→4 does not loop, because 4 is already in `visited` the second time it is offered.

`reachable(g, 6)` gives `[6 7]`, correctly finding the disconnected component, and `reachable(g, 5)` gives `[5]` because these edges are directed.

13.1.2 Why this is slow

Every operation in the listing is linear:

Operation	This version	What it should be
membership test (`seen`)	O(V)	O(1) with a hash set
enqueue (`append`)	O(V)	O(1) with a real queue
`neighbours`	O(E) — scans every edge	O(deg) with an adjacency list

Multiply those together and BFS over the whole graph is roughly $O(V^2 \cdot E)$ rather than O(V+E). For a graph with twenty nodes nobody will notice. For a road network it is unusable.

Two of the three are fixable within the functional style: an adjacency `Dict` from Chapter 11 makes `neighbours` a lookup, and the same `Dict` makes `seen` O(log V). The queue is fixable with the standard two-list trick — a front list and a reversed back list, giving amortised O(1) at both ends, which is one of the classic results in Okasaki's book.

The third fix, and the honest one for a grid, is to stop using lists.

13.2 The same problem on a grid, with buffers

A maze is a graph whose structure is implicit: cells are numbered by position, and the neighbours of `(x, y)` are the four cells around it. Nothing needs to be stored, and the natural representation is an array.

Rill's `Buf` (Chapter 15 covers it properly) is a counted block of elements of one C width — a real array, bounds-checked, reference counted, and **mutable**. It is the deliberate hole in the language's immutability, and this is exactly the kind of code it exists for.

```
# book/code/ch13/flood.rill
fn clear(d: Buf(Int), i) =
  buf_set(d, i, 0 - 1)
  if i + 1 >= buf_len(d) then 0 else clear(d, i + 1)

fn flood(g: Buf(U8), d: Buf(Int), q: Buf(Int), from) =
  clear(d, 0)
  buf_set(d, from, 0)
  buf_set(q, 0, from)
  walk(g, d, q, 0, 1)

fn walk(g: Buf(U8), d: Buf(Int), q: Buf(Int), head, tail) =
  cell = buf_get(q, head)
  after = side(g, d, q, cell % gw(), cell / gw(), buf_get(d, cell), 0, tail)
  if head + 1 >= after then 0 else walk(g, d, q, head + 1, after)
```

```
fn side(g: Buf(U8), d: Buf(Int), q: Buf(Int), x, y, far, k, tail) =
  nx = x + dx_of(k)
  ny = y + dy_of(k)
  next = if fresh(g, d, nx, ny) then push(d, q, ny * gw() + nx, far + 1, tail) else tail
  if k >= 3 then next else side(g, d, q, x, y, far, k + 1, next)

fn fresh(g: Buf(U8), d: Buf(Int), x, y) =
  open_at(g, x, y) && buf_get(d, y * gw() + x) < 0

fn push(d: Buf(Int), q: Buf(Int), cell, far, tail) =
  buf_set(d, cell, far)
  buf_set(q, tail, cell)
  tail + 1
```

```
$ rill run flood.rill
 ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
 ##  0  1  2  3  4 ## 18 19 20 21 22 23 24 ##
 ##  1 ## ## ##  5 ## 17 ## ## ## ## ## 25 ##
 ##  2 ##  8  7  6 ## 16 ## 18 19 20 ## 26 ##
 ##  3 ##  9 ## ## ## 15 ## 17 ## 21 ## 25 ##
 ##  4 ## 10 11 12 13 14 ## 16 ## 22 23 24 ##
 ##  5 ## ## ## ## ## ## ## 15 ## ## ## 25 ##
 ##  6  7  8  9 10 11 12 13 14 ## ## ## 26 ##
 ## ## ## ## ## ## ## ## ## ## ## ## ## ## ##
```

Each number is that cell's distance from the corner. This is a **flood fill**, and it is BFS with three arrays instead of three lists:

- **`d`** holds the distance to every cell, or -1 for "not reached". It is the visited-set and the answer at once.
- **`q`** is the queue: a flat array of cell indices with `head` and `tail` as cursors. Every cell is pushed at most once, so `q` never needs more than one slot per cell and no reallocation is possible.
- **`g`** is the maze, one byte per cell.

The loops are still tail recursion — `walk` over the queue, `side` over the four directions — so the control flow is the same shape as everything else in this book. What changed is the data.

Complexity: O(V + E), which for a grid is O(cells). Each cell is visited once and its four neighbours are examined once. There is no membership scan, because `d[cell] < 0` *is* the membership test, in one indexed read.

13.2.1 Reading the picture

The output repays a look. Follow the numbers from `0` at the top left: they run down the left wall, turn right, wind through the middle, and come back up the right-hand side to 26. The distances are corridor distances, not straight lines — cell `(13,1)` shows 24 despite being twelve cells away as the crow flies, because the only route to it goes all the way around.

That is what makes a flood fill useful for something other than distances: read the numbers around you and step to the smallest, and you are following the shortest path home, round every corner and out of every dead end, without ever storing a path. `examples/gl/maze.rill` in this repository does exactly this to make a creature chase the player through a maze, and `examples/gl/floodmap.rill` runs the same flood with no window attached so its output can be checked — which is how the numbers above were verified against an independent breadth-first search.

13.3 Choosing between them

	Lists (§13.1)	Buffers (§13.2)
Lines of code	~15	~40
Complexity	$O(V^2 \cdot E)$ as written	O(V + E)

	Lists (§13.1)	Buffers (§13.2)
Memory	allocates per step	three fixed arrays
Immutable	yes	no — `buf_set` mutates
Reads like the definition	yes	not really
Works for arbitrary graphs	yes	grids and dense integer-keyed graphs

The rule that falls out of this chapter, and it generalises past graphs:

> **Use values until the shape of the problem is an array. Then use an array.**

The list version is the one to write first, because it is short enough to be obviously correct and it is where you discover whether the algorithm is right at all. The buffer version is what you write when the profile says the graph is big and this is the hot loop — and by then you have a correct version to check the fast one against.

That last point is not rhetorical. When the flood fill in the maze demo was first written it silently produced nothing, and the way it was fixed was to run the same flood outside the game and compare its distances against a reference BFS, cell by cell. Having two implementations of the same thing is not duplication; it is the only cheap way to know that the fast one is right.

13.4 What is missing, and what to reach for

Rill 0.2 has no priority queue, so **Dijkstra's algorithm** and **A*** are not one-liners. On a grid with unit costs you do not need them — BFS gives the shortest path, which is why the flood fill above is enough for a maze. With weights you need a heap, and a functional leftist heap is about thirty lines and is the other famous chapter of Okasaki.

Two more standard graph algorithms are worth knowing fit comfortably here:

Depth-first search is the same code with the frontier used as a stack instead of a queue — `append(fresh, rest)` instead of `append(rest, fresh)`. One expression, and you have DFS.

Topological sort is DFS with the node appended to the answer *after* its children, which is a two-line change and is the natural way to schedule anything with dependencies.

Connected components is `reachable` in a loop over the nodes not yet seen, which the exercises ask for.

13.5 Exercises

1. Turn `bfs` into DFS by changing one expression, and check that the traversal order changes but the reachable set does not.
2. Make the graph undirected by adding both directions in `neighbours`, then compute the connected components of a graph with three of them.
3. Replace `seen` and `neighbours` with the `Dict` from Chapter 11 and measure the difference on a graph of a thousand nodes.
4. In `flood.rill`, write `path_from(d, x, y)` that returns the list of cells from `(x, y)` back to the start by always stepping to a neighbour whose distance is one less.
5. Add a second flood from a different start, and produce the set of cells that are strictly closer to the first one — a Voronoi partition of the maze.
6. Prove the queue in §13.2 can never overflow. What is the exact bound, and which line of `push` guarantees it?

That is the last algorithm chapter. What follows is the machinery underneath — how memory is managed with no collector, and how the concurrency works.

Chapter 14

Dynamic Programming

Dynamic programming is the family of algorithms that solve a problem by filling in a table of smaller answers. It is also the family where the functional style runs hardest into its own constraints, because "fill in a table" is a sentence about mutation.

This chapter is therefore the most useful one in Part III for calibrating your expectations. It has one problem that is *better* functionally, and one that is not, and the difference between them is worth understanding precisely.

14.1 The one that is better: Fibonacci

The textbook motivation for memoisation:

```
# book/code/ch14/fib.rill
fn slow(n) = if n < 2 then n else slow(n - 1) + slow(n - 2)

fn fast(n) = go(n, 0, 1)

fn go(n, a, b) = if n == 0 then a else go(n - 1, b, a + b)

fn main() =
  println(slow(27))
  println(fast(27))
  println(fast(90))
```

```
$ rill run fib.rill
196418
196418
2880067194370816120
```

`slow` is the exponential version — it recomputes `fib(25)` thousands of times, and it is the standard illustration of why memoisation exists.

`fast` does not memoise. It does something better: it observes that the recurrence only ever looks back two steps, and carries those two values forward as accumulators. It is O(n), constant space, tail recursive, and `fib(90)` returns instantly.

This is the pattern to look for before reaching for a table. A great many "dynamic programming" problems are really *linear recurrences with a small window*, and a window that fits in parameters does not need a table at all. Functional programming pushes you towards noticing that, because a table is awkward and parameters are not.

Where memoisation goes when you do want it

A memo table is a `Dict` (Chapter 11) threaded through the recursion: each call takes the table and returns the answer *and* the updated table. It works, it is O(log n) per lookup, and it is noticeably more code than the imperative `@lru_cache`. Languages with mutable-but-controlled state (a `ref` in OCaml, `unsafePerformIO` in Haskell, `Buf` here) all cheat at this point, and so does the next section.

14.2 The one that is not: edit distance

Levenshtein distance — the minimum number of insertions, deletions and substitutions to turn one string into another — is genuinely a table problem. The recurrence looks two ways at once:

```
d[i][j] = min( d[i-1][j-1] + (a[i] != b[j])   substitute
             , d[i-1][j]   + 1                delete
             , d[i][j-1]   + 1 )              insert
```

Every cell depends on three neighbours. There is no window that fits in a parameter, and building it out of immutable lists means either O(n) indexing or rebuilding a row per cell.

So this is where `Buf` earns its place in the language:

```
# book/code/ch14/edit.rill
fn cost(a, b) = if a == b then 0 else 1

fn min3(a, b, c) = min(a, min(b, c))

fn first_row(row: Buf(Int), j, n) =
  buf_set(row, j, j)
  if j >= n then 0 else first_row(row, j + 1, n)

fn next_row(a, b, prev: Buf(Int), cur: Buf(Int), i, j, n) =
  # substitution from the diagonal, deletion from above, insertion from the left
  d = min3(buf_get(prev, j - 1) + cost(str_get(a, i - 1), str_get(b, j - 1)), buf_get(prev, j) + 1, buf_get(cur,
  buf_set(cur, j, d)
  if j >= n then 0 else next_row(a, b, prev, cur, i, j + 1, n)

fn rows(a, b, prev: Buf(Int), cur: Buf(Int), i, m, n) =
  buf_set(cur, 0, i)
  next_row(a, b, prev, cur, i, 1, n)
  copy_row(cur, prev, 0, n)
  if i >= m then buf_get(prev, n) else rows(a, b, prev, cur, i + 1, m, n)

fn copy_row(from: Buf(Int), to: Buf(Int), j, n) =
  buf_set(to, j, buf_get(from, j))
  if j >= n then 0 else copy_row(from, to, j + 1, n)

fn distance(a, b) =
  m = str_len(a)
  n = str_len(b)
  prev = buf_int(n + 1)
  cur = buf_int(n + 1)
  first_row(prev, 0, n)
  if m == 0 then n else rows(a, b, prev, cur, 1, m, n)

fn main() =
  println(distance("kitten", "sitting"))
  println(distance("flaw", "lawn"))
  println(distance("", "abc"))
  println(distance("same", "same"))
  println(distance("intention", "execution"))
```

```
$ rill run edit.rill
3
2
3
0
5
```

Those are the right answers — `kitten` → `sitting` is 3, `intention` → `execution` is 5, both textbook values, and they were checked against an independent implementation while writing this chapter.

14.2.1 Reading it

Only two rows exist at a time. The full table is m×n, but each row depends only on the one above it, so `prev` and `cur` are enough. That is the standard space optimisation, and it is the same in every language.

The loops are still tail recursion. `next_row` walks a row, `rows` walks the rows, `copy_row` copies one to the other — three loops, each a function with its index as a parameter. The control flow is Chapter 7; only the data is different.

`buf_set` mutates. This is the one place the language lets you, and it is what makes each cell O(1). The buffer is reference counted like a string, so there is no `free` to forget, and every index is bounds-checked — which is how the off-by-one in the first draft of this listing announced itself:

```
rill: buffer index 8 out of range, length 8
```

That message is worth the check it costs. The bug was reading the diagonal from `prev[j]` instead of `prev[j-1]`; in C it would have read whatever was next in memory and produced a plausible wrong number.

14.2.2 What it would look like without `Buf`

You can write this with lists. `nth` gives you indexing, so the recurrence translates directly — and every `nth` is O(n), turning an O(mn) algorithm into O(mn·n). For two ten-character words nobody notices. For diffing two files it is the difference between instant and never.

The alternative that stays immutable is to build each row as a list, left to right, with the partially built row available as an accumulator. That works, is O(mn), and is genuinely elegant — `cur[j-1]` is the head of the accumulator, so "insertion from the left" is free. It allocates a cons cell per cell of the table, where the buffer version allocates two rows for the whole run.

All three are legitimate. The order to try them in is: the elegant list version first, and the buffer only when a measurement says the allocation matters.

14.3 The general shape

Across the chapter, three ways to carry the state of a table:

Method	Lookup	Allocation	When
Accumulator parameters	free	none	the recurrence has a small fixed window
A `Dict` threaded through	O(log n)	per entry	sparse tables, or keys that are not integers
A `Buf`	O(1)	one array	dense integer-indexed tables, in a hot loop

And the decision procedure that goes with it:

1. **Can the recurrence be rewritten with a window?** Fibonacci can; edit distance cannot. If it can, you are done and you never build a table.
2. **Is the table sparse, or keyed by something other than a small integer?** Then a `Dict` is the right structure and the log factor is the price of staying immutable.
3. **Is it dense, integer-keyed and hot?** Then it is an array, and Rill lets you say so.

The thing to resist is deciding this by ideology in either direction. A functional programmer who insists on lists for edit distance has written a quadratic slowdown into their program for no benefit; an imperative programmer who reaches for an array before checking whether a window exists has written twenty lines where three would do.

14.4 Exercises

1. Write the 0/1 knapsack in a `Buf`, then work out whether it has a window that would let you avoid the table. (It does not, and understanding why is the point.)
2. Write the list-based edit distance described in §14.2 and check it against the buffer version on a few hundred random pairs.
3. Extend `distance` to return the *alignment* rather than just the number. This needs the whole table rather than two rows — what does that change?
4. Longest common subsequence is the same table shape with a different recurrence. Write it, and then write `diff(a, b)` on top of it.
5. Write `coins(amount, denominations)` — the fewest coins making an amount — first with a `Dict` memo, then with a `Buf`. Compare the code and the time.

Part III ends here. Chapter 15 goes underneath the language: what happens to memory when there is no collector.

Part IV

Underneath

Chapter 15

Memory Without a Collector

Every functional language before this one shipped a garbage collector, because the style makes so much garbage: every `map` builds a list, every update copies a path, every intermediate value is allocated and dropped. Something has to clean up, and for fifty years that something has been a collector.

Rill does not have one. This chapter explains what it has instead, what that buys, and the one thing it cannot do.

15.1 The compiler inserts the frees

A tracing collector runs periodically, walks from the roots to find what is still reachable, and frees the rest. It is a *run-time* activity that needs to know about every pointer in the program.

Rill does the work at *compile time*. The compiler follows ownership rules — this is the Perceus approach, from the Koka language — and works out where each value's last use is. Then it inserts the release right there.

Concretely, for a function like

```
fn area_of(s) =
  a = area(s)
  a * 2.0
```

the compiler knows `s` is not used after `area(s)`, so it emits the release immediately after that call rather than at the end of the function. The rules are mechanical: every expression yields an owned reference, arguments are consumed by the callee, a binding retains, and anything still owned at the end of a function is released.

Nothing runs periodically. There is no heap walk, no pause, no read barrier, no write barrier, and no collector thread. A value is freed at the instruction after it dies.

15.1.1 Checking it

The runtime can count for you:

```
# book/code/ch15/alloc.rill
type Point
  Point(x: Int, y: Int)

fn make(n, acc) = if n == 0 then acc else make(n - 1, Cons(Point(n, n * 2), acc))

fn main() =
  pts = make(100000, Nil)
  println(len(pts))
  println(nth(pts, 0))
```

```
$ RILL_DEBUG_ALLOC=1 rill run alloc.rill
100000
Some(Point(1, 2))
live allocations: 0
```

Two hundred thousand objects — a hundred thousand list cells and a hundred thousand points — allocated, used, and every one of them freed by the time the program exits. Nobody wrote a `free`.

`RILL_DEBUG_ALLOC=1` is the tool to reach for when you suspect a leak. Zero is the number you want, and it is the number every example in this book produces.

15.2 What this buys

Determinism. There is no pause, because there is nothing that runs. The worst case for any single operation is the cost of freeing a chain of objects that all died at once, which is bounded by what you allocated. For anything with a latency budget — a game frame, an audio callback, a request tail — this is the difference between a guarantee and a hope.

Small heaps. A tracing collector needs headroom: it must let garbage accumulate before a collection is worth doing, so peak memory is typically two to five times the live set. Reference counting frees on the spot, so peak memory is close to the live set. This is why the binary-trees benchmark in Chapter 2 shows Rill using *less* memory than either Go or OCaml while also running faster — it is not a better collector, it is no collector.

No runtime to ship. The 33 KB hello-world in Chapter 2 is 33 KB partly because there is no collector in it.

Cheap nullary constructors. `Nil`, `None`, `Leaf` and every other constructor with no fields is a static singleton — one immortal object in the binary, shared by everything:

```
$ RILL_DEBUG_ALLOC=1 rill run nullary.rill
2
live allocations: 0
```

They allocate nothing at all, which matters more than it sounds: a program full of `Option` and lists creates `None` and `Nil` constantly, and in this language those are free.

15.3 What it costs

Cycles leak. This is the honest, permanent cost. Reference counting frees an object when its count reaches zero, and two objects pointing at each other keep each other's count at one forever. Rill has no cycle collector.

In practice this bites far less than it sounds, and the reason is Chapter 1's: **you cannot easily build a cycle out of immutable values.** To make `a` point at `b` and `b` point at `a`, one of them must be modified after the other was created — and nothing can be modified. Every structure in Chapters 6 through 14 is acyclic by construction.

Where you *can* build one is through a `Buf` or through C. If you store an index into a buffer that refers back, that is fine — an index is not a reference. If you stash a counted value inside a raw `Ptr` slot, the count is lost and you are on your own. Both are avoidable, and `RILL_DEBUG_ALLOC=1` will tell you if you did not avoid them.

Counting has a cost. Every value passed, bound and returned adjusts a count. The compiler elides many of them — a value that is created, used once and dropped may never be counted at all — but not all. This is the ordinary trade against a tracing collector, which makes allocation nearly free and pays in pauses instead.

That cost is also why Rill's concurrency defaults to a single OS thread (Chapter 16). One thread means the counts can be plain increments rather than atomic ones, which is worth a great deal on an operation this frequent.

15.4 `Buf`: the deliberate hole

Everything above assumes immutable values. But a texture, a pixel row, a dynamic-programming table and an array a C library wants to fill are all mutable blocks of one machine type, and pretending otherwise produces bad programs.

So Rill has one mutable structure, fenced off and named:

```
# book/code/ch15/buffers.rill
fn fill(v: Buf(F32), i) =
  buf_set(v, i, to_float(i) * 1.5)
  if i + 1 >= buf_len(v) then 0 else fill(v, i + 1)

fn main() =
  v = buf_f32(5)
  fill(v, 0)
  println(buf_get(v, 3))
  println(buf_len(v))
  w = buf_u8(2)
  buf_set(w, 0, 200)
  println(buf_get(w, 0))
  s = buf_i8(2)
  buf_set(s, 0, 200)
  println(buf_get(s, 0))
  println(buf_get(v, 99))
```

```
$ rill run buffers.rill
4.5
5
200
-56
rill: buffer index 99 out of range, length 5
```

A `Buf(w)` is a counted block of elements of one C width. There is a constructor per width — `buf_i8` `buf_u8` `buf_i16` `buf_u16` `buf_i32` `buf_u32` `buf_int` `buf_f32` `buf_float` `buf_ptr` — and the argument is a length in *elements*, not bytes.

Four properties, and each one is a decision:

The width lives in the type. `Buf(U8)` and `Buf(I8)` are different types, and reading the same byte through them gives 200 and -56 — the fourth and fifth lines of the output. This is not a cast; it is what the type means.

Indices are bounds-checked. Reading past the end ends the program with the index and the length rather than returning whatever was next in memory. The last line of the output is that check, and Chapter 14 showed it catching a real off-by-one in a table recurrence.

It is reference counted like a string. It is released when the last reference goes. There is no `free`, and no way to forget one. This is the difference between `Buf` and the raw `alloc_bytes` pointer used for C interop, where you do own the free.

It is mutable. `buf_set` changes the block in place. That is the hole in the language's immutability, and it is deliberate — every use of it in this book is a case where the alternative was asymptotically worse.

The width is never inferred, because it is not a type variable. A function taking a buffer says which one: `fn fill(v: Buf(F32), i) = ...`.

15.4.1 When to reach for it

The rule from Chapter 13, restated:

> **Use values until the shape of the problem is an array. Then use an array.**

The cases in this book were: a flood fill over a grid (Chapter 13), a dynamic-programming table (Chapter 14), and generating a texture. All three are dense, integer-indexed, and hot. Everything else in fourteen chapters used ordinary immutable values, and that ratio is about right for real programs too.

15.5 How it compares

	Rill	Go, Java, OCaml, Haskell	Rust	C
Who frees	the compiler, via counts	a collector, at run time	the compiler, via ownership	you
Pauses	none	yes	none	none
Peak heap	≈ live set	2–5× live set	≈ live set	≈ live set
Cycles	leak	collected	leak (`Rc`)	your problem
Annotation burden	none	none	lifetimes	none
Use after free	impossible	impossible	impossible	possible

The interesting column is Rust's. Both languages free deterministically with no collector, and the difference is what they ask of you. Rust makes ownership part of the type system, so you write lifetimes and get zero counting overhead. Rill keeps ownership entirely inside the compiler, so you write nothing and pay for the counts that could not be elided.

That is a real trade rather than a win: Rust is faster on the operations Rill counts, and Rill is a language you can teach in eight chapters.

15.6 Exercises

1. Run every program in `book/code/` under `RILL_DEBUG_ALLOC=1`. Which ones report a non-zero count, and why?
2. Build a cycle on purpose using two `Buf(Ptr)` values and confirm it leaks. Then convince yourself you could not have done it with ordinary values.
3. Write `sum_buf(v: Buf(Int))` and compare its speed against summing a list of the same length. Predict the ratio first.
4. Write a growable array — a `Buf` plus a used-count, doubling when full. What does "doubling" mean when a `Buf` cannot be resized?
5. Read the `Buf` section of `LANGUAGE.md` and work out why the width cannot be inferred, given how the rest of the type system works.

Chapter 16 covers the other half of the runtime: how a hundred thousand concurrent things fit in fifty megabytes.

Chapter 16

Strands and Channels

Rill's concurrency is Go's: many lightweight threads passing values over typed channels, rather than shared memory and locks.

One of those lightweight threads is a **strand** — a strand of a rope, of which a program is many, twisted together. Go calls the same thing a goroutine, Erlang a process, Java a virtual thread, Rust a task. The concept is common; the word here is Rill's own.

> **A note for C++ networking people:** Boost.Asio uses `strand` for something else — a serialised execution context that guarantees handlers do not run concurrently. The two ideas are unrelated.

16.1 Three primitives

```
# book/code/ch16/strands.rill
fn worker(jobs, results) =
  n = recv(jobs)
  send(results, n * n)
  worker(jobs, results)

fn feed(jobs, i, n) =
  send(jobs, i)
  if i >= n then 0 else feed(jobs, i + 1, n)

fn collect(results, left, acc) =
  v = recv(results)
  if left <= 1 then acc + v else collect(results, left - 1, acc + v)

fn main() =
  jobs = channel()
  results = channel()
  spawn worker(jobs, results)
  spawn worker(jobs, results)
  spawn worker(jobs, results)
  spawn feed(jobs, 1, 10)
  println(collect(results, 10, 0))
```

```
$ rill run strands.rill
385
```

385 is $1^2 + 2^2 + \ldots + 10^2$, computed by three workers pulling from one queue.

That is the whole API:

`spawn expr`	run `expr` on a new strand
`channel()`	a `Chan(a)`; `channel(n)` buffers n values
`send(ch, v)` / `recv(ch)`	rendezvous — a sender blocks until a receiver arrives, and vice versa
`select`	wait on several channels; first ready wins

Three things about the listing are worth drawing out.

`worker` is an infinite tail recursion. It receives, sends, and calls itself. That is Chapter 7's "loop that never ends", running in constant stack forever. There is no `while true` because there does not need to be.

Three workers share one channel with no lock. A rendezvous channel *is* the synchronisation: whichever worker reaches `recv` first gets the value, and the others wait. There is no mutex in this program, and there is nothing to forget to unlock.

Nothing is shared but the channels. Each strand has its own stack and its own values. This is why Chapter 1's point about immutability and concurrency matters — a data race needs a write, and there is nothing to write.

> **`spawn` takes the whole expression**
>
> `spawn handle(accept(s))` runs *both* calls on the new strand, including the accept. If the caller was supposed to do the accepting, bind it first: `c = accept(s)` then `spawn handle(c)`. This is the most common mistake with `spawn`, and it turns an accept loop into a strand-creation loop.

16.2 Pipelines

Strands compose into stages, each doing one job and passing the result on:

```
# book/code/ch16/pipeline.rill
fn source(out, i, n) =
  send(out, i)
  if i >= n then send(out, 0 - 1) else source(out, i + 1, n)

fn square(inp, out) =
  v = recv(inp)
  if v < 0 then send(out, 0 - 1) else pass_on(inp, out, v)

fn pass_on(inp, out, v) =
  send(out, v * v)
  square(inp, out)

fn drain(inp, acc) =
  v = recv(inp)
  if v < 0 then acc else drain(inp, acc + v)

fn main() =
  a = channel()
  b = channel()
  spawn source(a, 1, 10)
  spawn square(a, b)
  println(drain(b, 0))
```

```
$ rill run pipeline.rill
385
```

Same answer, different structure: a generator, a transformer and a consumer, each an independent strand, connected by two channels.

This is `map` over a stream instead of over a list — and the difference is that the stages run concurrently and nothing is ever fully materialised. A negative number is the end-of-stream marker; a language with sum types over channels would use `Option` and be tidier, and there is no reason Rill could not.

16.3 `select`

`select` waits on several channels at once and takes whichever is ready first:

```
# book/code/ch16/selecting.rill
fn poll(ch) =
```

```
  v = select
    x = recv(ch) -> x
    default -> 0 - 1
  v

fn both(a, b, left, acc) =
  got = select
    x = recv(a) -> x
    y = recv(b) -> y
  if left <= 1 then acc + got else both(a, b, left - 1, acc + got)
```

```
$ rill run selecting.rill
309
-1
```

309 is (1+2+3) + (100+101+102) — six values taken from two channels in whatever order they arrived. The `-1` is the second call: `poll` on a channel with nothing in it takes the `default` arm rather than blocking.

- Arms are tried **in order**, so an earlier arm wins a tie.
- A **`default` arm** turns `select` into a non-blocking poll.
- A **send arm** (`send(ch, v) -> e`) is ready when the send could proceed.
- Like `match`, `select` needs an indented block, so it goes in a body, a statement or a binding.

16.4 A hundred thousand strands

```
# book/code/ch16/many.rill
fn relay(inp, out) = send(out, recv(inp) + 1)

fn chain(first, n) = if n == 0 then first else chain(link(first), n - 1)

fn link(inp) =
  out = channel()
  spawn relay(inp, out)
  out

fn main() =
  head = channel()
  tail = chain(head, 100000)
  spawn send(head, 0)
  println(recv(tail))
```

```
$ /usr/bin/time -l ./many
100000
        0.31 real
            56049664  maximum resident set size
```

A hundred thousand strands, each holding a channel and waiting to pass a number along, and the whole thing costs **56 MB and a third of a second** — including creating all of them, passing a value down the entire chain, and tearing it down.

That is about 560 bytes per strand. An OS thread would be a megabyte of stack each, so this program would need 100 GB.

16.4.1 How

A strand needs a stack, and a stack is the expensive part. The trick is not to give each one a stack of its own.

All running strands share **one** guarded 8 MB execution stack at a fixed address. When a strand parks — blocks on a channel or a socket — the scheduler copies out only the bytes it actually used, into a heap

block sized to fit, and keeps that on the strand's record. When it resumes, the bytes are copied back to the *same address*, so every interior pointer in those frames is still valid.

A parked strand therefore costs roughly what it was actually using — a few hundred bytes — rather than a reserved stack. There are no per-strand guard pages to install, no `mmap` per spawn, and no page faults for stack that was never touched.

The cost is the copy on every park and resume, which is why this design suits strands that block often and briefly. The Chapter 2 benchmark table is the measurement: the 100k-strand ring is 13.5 ms against Go's 96 ms and OCaml's 62 ms, and 53 MB against 269 MB and 103 MB.

16.5 Blocking on the world

A strand that waits on a socket does not hold up anything. Its descriptor goes to the operating system's poller and the worker moves on to another strand; when the socket is ready the scheduler wakes it.

```
fn serve(server) =
  c = accept_on(server)
  spawn handle(c)
  serve(server)
```

One strand per connection, ten thousand connections, and the cost while they wait is the parked stacks. This is why `examples/http/http.rill` — an HTTP/1.1 server in about a hundred lines of Rill — outruns Go's `net/http` on the same one-route service.

The limitation is C. A foreign call that blocks freezes the OS thread running it, because the scheduler cannot preempt code it did not compile. Chapter 17 covers the boundary.

16.6 Deadlock, and the exit rule

Two behaviours to know, both inherited from Go and both visible immediately.

A program where every strand is blocked is reported rather than hung:

```
$ rill run dead.rill
rill: deadlock — all strands are blocked
```

That is the whole program `ch = channel()` then `recv(ch)`. Nothing will ever send, the scheduler notices that no strand can make progress, and it says so instead of hanging forever.

The program exits when `main` returns. Strands still running are abandoned, exactly as in Go. If the answer matters, `main` must wait for it — which is what the `collect` and `drain` functions in this chapter are doing. Under `RILL_DEBUG_ALLOC=1` an abandoned strand shows up as live allocations at exit; that is not a leak in the reference counter, it is a program that stopped while holding things.

16.7 Several cores, on purpose

By default every strand runs on **one OS thread**. That is not a limitation that was left in; it is what allows reference counting to be non-atomic (Chapter 15), and that is worth a great deal on an operation as frequent as counting.

Programs that want real parallelism opt in twice:

```
rill build main.rill -o main --parallel   # counts become MT-safe
RILL_THREADS=8 ./main                     # eight workers
```

`--parallel` makes each count check a flag and use an atomic update when more than one worker is live. Without it the runtime refuses `RILL_THREADS` and says so — you cannot get the parallelism by accident, and you cannot get data races by forgetting the flag.

Workers steal strands that have not started yet from one another. A strand that has already run stays on its worker, because its frames live at that worker's stack address.

Parallelism helps when strands compute independently. A chain of rendezvous — one strand handing a value to the next, like §16.4 — has nothing to overlap, and runs fastest on a single worker. Measure before you reach for the flag.

16.8 Compared

	Rill strands	Go goroutines	Erlang processes	OS threads
Cost each	~560 B parked	~2–4 KB	~2 KB	~1 MB stack
Communication	typed channels	typed channels	mailboxes	shared memory
Shared mutable state	none	yes, with locks	none	yes, with locks
Parallel by default	no (opt in)	yes	yes	yes
Failure isolation	none	none	supervision trees	none

Rill is closest to Go, by design. It differs in defaulting to one thread — a consequence of the memory model — and it lacks Erlang's supervision, which is the feature that makes that language special and is a much larger design than a scheduler.

16.9 Exercises

1. Turn §16.2's pipeline into three stages and make the end marker an `Option` over the channel instead of a negative number.
2. Write a worker pool that shuts down cleanly: a poison value on the jobs channel, and each worker acknowledging before it stops.
3. Use `select` with a `default` to write `try_send` that gives up rather than blocking when a buffered channel is full.
4. Build the ring from §16.4 with 1,000 strands and pass a token round it ten times. Where does the time go — creating strands, or passing the token?
5. Take a program that spawns strands doing independent arithmetic, build it with and without `--parallel`, and measure with `RILL_THREADS` from 1 to 8. Then do the same with §16.4's chain and explain the difference.

That is the runtime. `LANGUAGE.md` covers what remains — calling C, sockets, and the toolchain — and `examples/` has the programs that use them.