An Introduction to Rank-Polymorphic Programming in Remora

Updated by Eric McCarthy

Original authors:
Olin Shivers, Justin Slepak, and Panagiotis Manolios (2019)

Version 2.0 — June 10, 2026

Abstract

Remora is a higher-order, rank-polymorphic array-processing language, in the same general class of languages as APL and J, intended for writing programs to be executed on parallel hardware.

This is an example-driven introduction to the language as accepted by its current Haskell implementation. Where the original tutorial began with a dynamically typed dialect, the current language is explicitly typed throughout: every program carries the type, shape, and polymorphism annotations that feed Remora's static, dependent type system, in which the shapes of computed arrays are known at compile time. We develop the rank-polymorphic computational model — the implicit lifting of operations across the cells of high-dimensional arrays — one worked example at a time.

Every example in this edition has been machine-verified against the implementation: a doctest harness runs each example through the interpreter and checks its result, so the examples cannot silently drift out of step with the language.

A reader interested in the rank-polymorphic array-processing model that underlies this whole class of languages should find the tutorial informative, above and beyond the specific details of Remora.

For fuller background, the reader may wish to consult the 2019 tutorial alongside this edition — in particular its broader discussion of parallelism and the map/reduce execution model, and its treatment of topics tied to language features that the current implementation does not yet provide.

Contents

Background

This is a rewrite of An Introduction to Rank-polymorphic Programming in Remora (Draft), by Olin Shivers, Justin Slepak, and Panagiotis Manolios (December 31, 2019). That tutorial taught the language in two stages: first **Dynamic Remora**, an untyped dialect that survives in Slepak's earlier Racket implementation (`#lang remora/dynamic`, github.com/jrslepak/Remora) but is no longer maintained, and then **Explicitly Typed Remora**, whose syntax has since evolved. This edition teaches the language accepted by the current Haskell implementation (`remora-lang/remora`), in which all type, shape, and polymorphism information is written explicitly — there is no type inference and no dynamic dialect.

Every example in this document has been machine-verified against the implementation at commit `ebca13b` (2026-06-10). Examples are shown as a fenced code block whose last line is a comment giving the verified result:

```
(+ [1 2] [3 4])
; ⇒ [4 6]
```

A small script (Appendix A) extracts each block, strips the `; ⇒` line, runs the rest through `remora interpret`, and compares. If the implementation drifts, the doctest fails, and this document gets fixed — the defect this edition most wants to avoid is the one its predecessor suffers from: examples that stop being true.

Running Remora

Please see the Remora repository on GitHub to install Remora. The implementation builds with nix; from a checkout:

```
$ nix shell .#remora-wrapped             # Puts 'remora' (plus z3, futhark) on PATH,
                                         # in a subshell. Including '.#remora-wrapped'
                                         # is faster because it doesn't build docs.
$ remora interpret -e "(+ [1 2] [3 4])"
[4 6]
$ remora interpret -f program.remora     # run a file
$ remora parse -e "[0 3]"                # parse, print desugared form
(frame [2] (array [] 0) (array [] 3))
$ remora parse -e "(+ 1 [2 3])" -s       # fully elaborated s-expression
$ remora repl                            # interactive REPL (>> prompt)
```

Notes: `remora repl` works interactively but hangs on piped input (the CLI source labels it a work in progress) — use `interpret -e` for scripting. The `futhark` subcommand compiles to Futhark instead of interpreting; this tutorial uses only the interpreter. On macOS, use `nix shell .#remora-wrapped` rather than `nix develop` (the dev shell references CUDA packages that do not evaluate on Darwin).

Comments run from `;` to end of line. Keywords with Greek letters (`λ`, `Π`, …) all have ASCII spellings (`fn`, `Pi`, …); both are used below and are interchangeable.

---

1. Everything is an array

In Remora, **all values are arrays**. Every expression evaluates to an array. An array is a collection of atoms arranged in a hyper-rectangle; atoms are numbers, booleans, and functions. (Arrays of functions are real and useful — see §3.5.)

An array's **shape** is the vector of its dimensions; its **rank** is the length of its shape. A 2×3 matrix has rank 2 and shape `[2 3]`. A scalar is an array of rank 0: its shape is the empty vector `[]`, and it contains exactly one atom (the product of an empty list of dimensions is 1).

The primitive notation for a literal array is the `array` form: shape first, then the atoms in row-major order:

```
(array [2 3] 7 1 2 2 0 5)   ; a 2x3 matrix
; ⇒ [[7 1 2] [2 0 5]]
```

```
(array [] 17)               ; the scalar seventeen
; ⇒ 17
```

Larger arrays are assembled from array-producing expressions with the `frame` form: `(frame [d1 ... dn] e1 ... ek)` evaluates the e_i (there must be $d_1 \cdot \ldots \cdot d_n$ of them, all producing arrays of one identical shape `[c1 ... cm]`) and stacks the results into an array of shape `[d1 ... dn c1 ... cm]`:

```
(frame [2] (i-app iota/static [3])
           (@reverse (Int) (3 []) (i-app iota/static [3])))
; ⇒ [[0 1 2] [2 1 0]]
```

(Don't worry about the unfamiliar operators yet: `(i-app iota/static [3])` builds the vector `[0 1 2]` — `iota/static` is covered with the other generators in §8.4, and `i-app` in §4 — while `@reverse` (§8.2) reverses it. The point here is just that frame entries are computed at run time, whereas `array` entries are literal atoms.)

Three pieces of syntactic sugar hide `array` and `frame` almost everywhere:

1. An atom literal in expression position stands for a scalar array: `17` means `(array [] 17)`.
2. `[e1 ... en]` means `(frame [n] e1 ... en)`.
3. A frame whose entries are all literals collapses into a single `array` literal, so `[[7 1 2] [2 0 5]]` is `(array [2 3] 7 1 2 2 0 5)`.

Brother elements of a bracket expression must have identical shapes — there are no ragged arrays. (`[[1 2] [3]]` is ill-shaped. When you genuinely need mixed sizes, you need boxes: §7.)

1.1 The atom literals

```
[1 -2 3]            ; Int
; ⇒ [1 -2 3]
```

```
(f.* 4.5 2.0)       ; Float (note the f. operator — see §8)
; ⇒ 9.0
```

```
[#t #f]             ; Bool literals
; ⇒ [#t #f]
```

Two honest warnings about the corners of this table:

- **Bool values exist but nothing consumes them.** The current prelude has no comparisons, no boolean operators, and the language has no `if`. (The original tutorial's `filter`, `select`, `grade`, conditionals, etc. all belong to the missing-builtins list in §10.)
- **Strings are Int vectors.** `"abc"` parses, but there is no `Char` type — a string literal denotes the vector of Unicode code points:

```
"abc"
; ⇒ [97 98 99]
```

An empty array poses a puzzle: with no atoms, nothing determines its type. So the `array`/`frame` forms have a variant whose body is a type instead of atoms:

```
(array [0] Int)
; ⇒ []
```

2. Types, shapes, and the sigil zoo

Remora's type system is a restricted dependent type system, parameterized two ways: over other types, and over **ispaces** ("index spaces") — the static dimensions and shapes of arrays. (The 2019 tutorial and the published papers call these indices; ispace is the current term, and not every corner of the implementation has caught up — the Haskell AST calls them `Extent`, which is where §5's `extent` keyword comes from.) Restricting the dependency to this little ispace language (rather than arbitrary run-time values) is what makes shape checking purely static; the price is that you can only say things about shapes that the ispace language can express. (One would like to add "and it makes type checking decidable" — but not quite: see §4.2.)

Dimensions (sort `Dim`) are single axis lengths: a literal like `3`, a dimension variable `$d`, or arithmetic `(+ ...)`, `(* ...)`, `(- ...)` over dimensions.

Shapes (sort `Shape`) are sequences of dimensions: a shape variable `@s`, an explicit `(dims d1 d2 ...)`, a concatenation `(++ s1 s2 ...)`, or — most commonly — the **splice notation** `[i1 i2 ...]`, which splices any mix of dimensions and shapes into one shape. `[2 @s 3]` means "2, then all of `@s`, then 3".

The **array type** is `(A atom-type shape)`. Its sugar absorbs most of the notation you'll actually write:

You write	It means
`[Int 2 3]`	`(A Int (dims 2 3))`
`(A Int [2 3])`	same
`[&t $m @s]`	`(A &t (++ (dims $m) @s))`
`Int` (in type position)	`(A Int (dims))` — a scalar
`[Int]`	also a scalar (empty splice)

Base atom types are `Int`, `Float`, and `Bool`. Function types are also atom types — `(-> (T1 ... Tn) R)` (ASCII) or `(→ ...)` — which is precisely why arrays of functions make sense. The remaining atom types are the binders `Forall`/`∀`, `Pi`/`Π`, and `Sigma`/`Σ`, which get all of §4.

Four kinds of variables appear in types, distinguished by sigil. Learn the sigils now and every error message becomes legible:

Sigil	Example	Stands for	Sort/kind
`$`	`$n`	one dimension	ispace sort: Dim
`@`	`@s`	a whole shape	ispace sort: Shape
`&`	`&t`	an atom type	type: atom
`*`	`*v`	an array type	type: array

(Readers of the 2019 tutorial: it wrote element-type variables bare (as `t`) and array-type variables with `@`. The current language uses `&t` for atom types and `*t` for array types, so as a variable sigil `@` now always marks a shape. Its other use — heading the combined-application form `(@f ...)` (§4.4) — is a syntactic prefix, not a sigil, so the two never collide.)

3. Functions and the lifting machinery

3.1 Writing and applying functions

A function literal is an atom (so a bare `(fn ...)` in expression position becomes a scalar array containing one function). Every parameter carries its full type:

```
((fn ((x Int) (y Int)) (+ x (* 2 y))) 10 4)
; ⇒ 18
```

`λ` is interchangeable with `fn`. Application is juxtaposition: `(f a b)` — no keyword. The parameter types are not decoration; they are the engine of the whole language, because they declare the **cell shape** the function consumes. A parameter `(x Int)` consumes scalar cells. A parameter `(v [Int 2])` consumes cells of shape `[2]`.

3.2 Frames and cells

The core mechanism of Remora — the thing that replaces every loop you are not writing — is that a function defined on cells of one rank is automatically applied across arrays of any higher rank.

When a function whose parameter expects cells of shape `c` meets an argument of shape `s`, the argument's shape is split into `s = f ++ c`: a **frame** part `f` (leading axes) and the **cell** part `c` (trailing axes, which must match the parameter type). The function is then applied once per frame position, and the results are collected back into the frame:

```
argument shape  [5 7 2 3]
parameter type  [Int 2 3]      (cell shape [2 3])
                ─────┬─────
frame [5 7]  ++  cell [2 3]
→ 35 independent applications, results collected in a [5 7] frame
```

A 2×3 matrix can therefore be viewed three ways: as a scalar frame around one 2×3 cell, as a 2-frame of 3-vector cells, or as a 2×3 frame of scalar cells. The function's parameter type decides which view is taken.

```
(+ [1 2] [3 4])      ; + has scalar cells; frame [2] on both sides
; ⇒ [4 6]
```

3.3 Frame agreement and the principal frame

With several arguments, each argument gets its own frame. The frames must agree — but not by being identical. The rule:

> In an application, the longest argument frame is the **principal frame**; every other argument's frame must be a **prefix** of it. Arguments with shorter frames are **replicated** (cell-wise) into the missing dimensions.

The everyday consequence: adding a scalar to anything replicates the scalar everywhere, and adding a vector to a matrix adds element `i` of the vector to row `i` of the matrix:

```
(+ 10 [7 1 4])
; ⇒ [17 11 14]
```

```
(+ [10 20] [[8 1 3]
            [5 0 9]])
; ⇒ [[18 11 13] [25 20 29]]
```

Here is that last one in slow motion. `+` takes scalar cells, so the first argument has frame `[2]` and the second has frame `[2 3]`:

```
frames:   [2]  vs  [2 3]        [2] is a prefix of [2 3]  ✓
principal frame: [2 3]

replicate the [2]-framed argument along the missing axis (length 3):

    10 ──────► 10 10 10            row 0 of the matrix gets 10
    20 ──────► 20 20 20            row 1 of the matrix gets 20

apply + at each of the 6 frame positions:

    [[10+8  10+1  10+3]      =   [[18 11 13]
     [20+5  20+0  20+9]]          [25 20 29]]
```

Replication only ever supplies entire cells; an argument must always contribute at least one complete cell. And a frame that is not a prefix of the principal frame is an error (the message you get is shown in §3.6).

3.4 The same data, two different splits

Everything above was driven by the parameter types, so changing the types changes the behavior — with the same argument data. This pair of verified examples is worth staring at:

```
(+ [10 100] [[1 2]
             [3 4]])   ; scalar cells: the vector's elements go to
                       ; the matrix's ROWS
; ⇒ [[11 12] [103 104]]
```

```
((λ ((x [Int 2]) (y [Int 2])) (+ x y))
 [10 100]
 [[1 2]
  [3 4]])                ; vector cells: the vector goes to
                         ; each row WHOLE
; ⇒ [[11 102] [13 104]]
```

In the first, +'s scalar cells make the frames `[2]` and `[2 2]`; `10` is added to all of row 0 and `100` to all of row 1. In the second, the λ's parameter types declare `[2]`-vector cells, so the frames become `[]` and `[2]`: now the whole vector `[10 100]` is replicated once per row, and inside each application `(+ x y)` lifts element-wise:

```
       x = [10 100]   (one cell, frame [])
       y = [[1 2]     (two cells, frame [2])
            [3 4]]

   frame position 0:  (+ [10 100] [1 2])  =  [11 102]
   frame position 1:  (+ [10 100] [3 4])  =  [13 104]

   collect into frame [2], cells [2]  →  shape [2 2]
```

Wrapping a function in a λ with bigger-cell parameter types is called **reranking** by η-expansion. The 2019 tutorial had dedicated sugar for it (`~(1 1)+`); the current language does not, so this λ wrapper is the idiom. §6 develops it.

3.5 The function position is an array too

The expression in function position is an ordinary expression, and it evaluates to an array of functions that participates in frame agreement like any argument. Usually it's a scalar array (one function) replicated across the principal frame — but it doesn't have to be:

```
([+ *] 2 3)         ; a 2-frame of functions, applied to scalars
; ⇒ [5 6]
```

The frame `[2]` of the function array becomes the principal frame; `2` and `3` are replicated; position 0 computes `(+ 2 3)`, position 1 computes `(* 2 3)`.

Closures fall out of the same story. Here `add-n` maps over `[1 2]` producing a 2-frame of functions, which is then applied:

```
(let ((fun (add-n (n Int) : (-> (Int) Int))
        (fn ((x Int)) (+ n x))))
  ((add-n [1 2]) [3 4]))
; ⇒ [4 6]
```

3.6 Reading the two shape errors

You will meet exactly two complaints from the lifting machinery, both reported on the desugared form (so `[1 2]` prints as `(frame [2] (array [] 1) (array [] 2))`). First, cells that cannot be carved out — or frames that disagree:

```
>> (+ [1 2] [1 2 3])
error: Ill-shaped application:
(+ (frame [2] ...) (frame [3] ...))
```

```
>> (+ [1 2 3] [[1 2] [3 4]])      ; [3] is not a prefix of [2 2]
error: Ill-shaped application:
(+ (frame [3] ...) (frame [2] (frame [2] ...) ...))
```

Second, the error that — by sheer frequency — deserves its own section of this tutorial: applying something that is not (yet) a function. That is the subject of §4.

---

4. The three kinds of abstraction: →, ∀, Π (and Σ)

Remora has three kinds of function-like values, one per kind of thing you can pass: values, types, and ispaces. Each has its own abstraction form, application form, and type constructor:

Abstracts over	Function form	Application form	Its type
values	`(fn ((x T) ...) e)` / `λ`	`(f a ...)`	`(-> (T ...) R)` / `→`
types	`(t-fn (&t ...) e)` / `tλ`	`(t-app f T ...)`	`(Forall (&t ...) R)` / `∀`
ispaces	`(i-fn ($d @s ...) e)` / `iλ`	`(i-app f i ...)`	`(Pi ($d @s ...) R)` / `Π`

(The fourth binder, `Sigma`/`Σ`, is not a function but a package; §7.)

4.1 ∀ — polymorphism over types

`Forall` is ordinary parametric polymorphism, as in ML or System F. It binds atom-type variables (`&t`) or array-type variables (`*v`); you eliminate it with `t-app`, and types are erased at run time:

```
(let ((t-fun (ident (&t)) (array [] (fn ((x [&t])) x))))
  ((t-app ident Int) 42))
; ⇒ 42
```

4.2 Π — dependent abstraction over ispaces

`Pi` is where the type system earns the word dependent: the body's type **mentions** the bound ispace variables, so supplying different ispaces produces a different type. Instantiating `append : Π ($m $n @s) ...` with `(i-app ... 2 2 [3])` computes the monomorphic type `(-> ([Int 2 3] [Int 2 3]) [Int 4 3])` by substituting into `$m ++ @s`, `$n ++ @s`, `($m + $n) ++ @s`. (When talking about shapes, this tutorial writes both ++ and + infix for readability; in actual Remora syntax both are prefix — `(++ ...)` and `(+ ...)` — as in the prelude types of §8.)

Crucially, Π ranges only over the static ispace language of §2 — not over run-time values (that would be full dependent typing, à la Agda). Even this restricted language outruns decidability, though: dimensions admit multiplication (`flatten`'s result dimension is `(* $m $n)`), and nonlinear integer arithmetic is undecidable in general. In practice the implementation hands ispace constraints to an SMT solver (z3, via the sbv library), which settles the constraints real programs generate, with no guarantee for adversarial ones. The restriction to a static ispace language is still exactly the expressiveness you are paying for with all these annotations: the compiler can know shapes at compile time that APL never knew at all.

4.3 Why ”product” and ”sum”? — an aside

The names Π and Σ are not arbitrary Greek: they are the indexed product ∏ and indexed sum ∑ of mathematics, one level up from the standard product and sum types.

A value of type `Π ($m) T($m)` can hand you a `T(m)` for **every** index `m` — morally a giant tuple with one component per natural number, where `i-app` is projection. That is an element of the infinite product `∏_m T(m)`. A value of type `Σ (@s) T(@s)` is one specific index paired with a `T` at that index — an element of the disjoint union `∑_s T(s)`, a tagged union with one variant per shape, the index being the tag.

Now call a Π or Σ **degenerate** when its body type does not actually mention the bound variable — `Π (x : A) B` or `Σ (x : A) B` with `B` constant. The indexed reading still applies, but it collapses into a simple type:

- A degenerate `Π (x : A) B` must hand you a `B` for every `x` in `A` — a tuple of `B`s with one slot per element of `A`. But choosing one `B` for each element of `A` is precisely what a function `A → B` is. (This is also why mathematicians write the function space as `B^A`, and why `|B|^|A|` counts the functions.) So the ordinary function arrow is the non-dependent special case of Π.
- A degenerate `Σ (x : A) B` is some `x` from `A` paired with a `B` — and since `B` no longer varies with the tag, the tag and the payload are just two independent components: the pair type `A × B`. So the ordinary pair is the non-dependent special case of Σ.

Notice the criss-cross — it is the standard trap:

		dependent (indexed)	non-dependent degenerate
Π	"product"	∏_x B(x): one component per index	A → B (function type!)
Σ	"sum"	∑_x B(x): one variant per index	A × B (pair type!)

The dependent product degenerates to a function type, and the dependent sum degenerates to a pair — a binary product. The everyday ”product type” thus sits in Σ's column, not Π's, which is exactly why the names feel backwards on first meeting.

4.4 There is no inference — and the error that tells you so

The prelude's polymorphic functions are ∀-wrapped, Π-wrapped values. Plain application requires an arrow type, so applying one unapplied is an error — the single most common error in current

Remora:

```
>> (append [[0 1 2] [3 4 5]] [[10 20 30] [40 50 60]])
error: Expected an array of functions in application:
(append (frame [2] ...) (frame [2] ...))
(A (∀ (&t_13) (A (Π ($m_14 $n_15 @s_16)
   (A (-> ((A &t_13 (++ $m_14 @s_16)) (A &t_13 (++ $n_15 @s_16)))
          (A &t_13 (++ (+ 0 $m_14 $n_15) @s_16)))
      (++))) (++))) (++))
```

Do not scroll past that type — it is the documentation. Annotated:

```
(∀ (&t)                                  ← needs a t-app to choose &t
   (Π ($m $n @s)                         ← needs an i-app to choose $m $n @s
      (-> ((A &t (++ $m @s))             ← arg 1: shape  $m ++ @s
           (A &t (++ $n @s)))            ← arg 2: shape  $n ++ @s
          (A &t (++ (+ $m $n) @s)))))    ← result: (m+n) ++ @s
```

(Incidental notation: `(++)` is the empty shape — every ∀/Π value is itself a scalar array, hence the `(A ... (++))` wrappers; the `_13` suffixes are internal renaming; the `(+ 0 $m $n)` zero is a harmless artifact of ispace normalization.)

The fix is to instantiate, outside-in — `t-app` first, then `i-app`, then values:

```
((i-app (t-app append Int) 2 2 [3])
 [[0 1 2] [3 4 5]]
 [[10 20 30] [40 50 60]])
; ⇒ [[0 1 2] [3 4 5] [10 20 30] [40 50 60]]
```

This triple application is so common it has **combined application sugar**: `(@f (T ...) (i ...) arg ...)`, with _ standing for an empty list:

```
(@append (Int) (2 2 [3])
  [[0 1 2] [3 4 5]]
  [[10 20 30] [40 50 60]])
; ⇒ [[0 1 2] [3 4 5] [10 20 30] [40 50 60]]
```

```
(unbox ($d xs (@iota _ (1) [5])) xs)   ; iota has no ∀: type list is _
; ⇒ [0 1 2 3 4]
```

(Readers of the 2019 tutorial: two conventions flipped. The old tutorial nested Π outside ∀ and wrote `(t-app (i-app append 2 7 [3 5]) bool)`; the current prelude nests ∀ outside Π, so it's `(i-app (t-app append Int) ...)`.)

4.5 Higher-order argument positions are exact

Lifting happens at application sites. When a function is passed **as an argument**, its type must match the parameter type exactly — nobody η-expands it for you. `reduce`'s operator parameter has type `(-> ([&t @s] [&t @s]) [&t @s])`; with `@s = []` the scalar `+` fits:

```
(@reduce (Int) (3 []) + [1 4 9 16])
; ⇒ 30
```

but with `@s = [3]` you must hand it a function on `[3]`-cells — η-expanding `+` yourself:

```
(@reduce (Int) (2 [3])
  (fn ((x [Int 3]) (y [Int 3])) (+ x y))
  [[1 2 3] [10 20 30] [100 200 300]])
; ⇒ [111 222 333]
```

5. `let` and the six binding forms

A Remora program is one expression. `let` is how you build anything bigger: `(let (bind ..) body)`, where each binding is one of six forms:

Form	Binds	Example
`(val x e)`	a value	`(val v [10 100])`
`(val (x : T) e)`	a value, with ascription	`(val (v : [Int 2]) [10 100])`
`(fun (f (x T) .. [: R]) e)`	a function	below
`(fun (@f (&t ..)|_ ($i ..)|_ (x T) .. : R) e)`	a ∀/Π-wrapped function	§5.2
`(t-fun (f (&t ..) [: R]) e)`	a type function	§4.1
`(i-fun (f ($i ..) [: R]) e)`	an ispace function	§6, §9
`(type &n T)` / `(type *n T)`	a type alias	below
`(extent $n d)` / `(extent @n s)`	an ispace alias	below

A flavor of most of them at once — aliases, `extent`, and a typed `fun`, all in one verified program:

```
(let ((type &MyInt Int)
      (extent $d 3)
      (type *IntVec3 [&MyInt $d])
      (fun (add-int-vec (x *IntVec3) (y *IntVec3) : *IntVec3) (+ x y)))
  (add-int-vec [1 2 3] [4 5 6]))
; ⇒ [5 7 9]
```

Bindings are **sequential**, not parallel: each right-hand side sees every binding above it (Lisp/Scheme's `let*`, not its `let`). The example above depends on this — `*IntVec3` mentions both `&MyInt` and `$d`. A one-liner that shows it:

```
(let ((val x 1)
      (val x (+ x 10)))   ; sees, and then shadows, the first x
  x)
; ⇒ 11
```

The scope runs strictly downward, and a binding is not in scope in its own right-hand side — so there are no recursive definitions: `let` desugars to a nest of applications, not a `letrec`, and a self-referential `(fun (f ...) ... (f ...))` fails with `Unknown text var: f`. (With no `if` in the language, recursion would have little to do anyway.)

`extent` aliases live in the ispace world, so dimension arithmetic works there — `(extent $d (+ 5 (- 3)))` binds `$d` to 2. Note the sigil discipline: `type` names get `&` (atom) or `*` (array) sigils; `extent` names get `$` (dim) or `@` (shape).

5.1 The parenthesis pitfall

The body is **outside** the bindings list. Forgetting to close the list before the body produces a parse error whose wording you can learn to recognize:

```
>> (let ((val v [10 100]) (val m [[1 2] [3 4]]) (+ m v))
error: unexpected "+ m v)"
expecting "extent", "fun", "i-fun", "t-fun", "type", or "val"
```

The parser is still inside the bindings list, so it expects another binding keyword — that "expecting extent, fun, i-fun, t-fun, type, or val" list is the fingerprint. Close the bindings, then write the body:

```
(let ((val v [10 100])
      (val m [[1 2] [3 4]]))
  (+ m v))
; ⇒ [[11 12] [103 104]]
```

5.2 Polymorphic definitions: the @-signature

The binding form `(fun (@name (&t ...)|_ ($i ...)|_ params... : R) body)` — where each variable list is either parenthesized or replaced wholesale by _ — defines a name already wrapped in `t-fn`/`i-fn`, ready for `@`-application. The return type is mandatory here. A definition we will dissect in §9.1:

```
(let ((fun (@replicate (&t) (@f @s) (in [&t @s]) : [&t @f @s])
        ((array [] (fn ((x [Int])) in)) (i-app iota/static [@f]))))
  (@replicate (Int) ([3] []) 7))
; ⇒ [7 7 7]
```

---

6. Reranking without a rerank operator

The 2019 tutorial devoted a section to reranking — retargeting which axes a function consumes — with dedicated sugar `~(r1 r2)f`. That operator does not exist in the current language. You rerank manually, by two mechanisms, and between them you recover everything the sugar did.

6.1 Rerank by ispace choice

For a Π-polymorphic function, the ispace arguments decide where the cell/frame boundary falls in the arguments. You saw `append`'s type in §4.4: arguments `$m ++ @s` and `$n ++ @s`. The choice of `@s` decides which axis gets appended:

```
(@append (Int) (2 2 [3])
  [[0 1 2] [3 4 5]]
  [[10 20 30] [40 50 60]])
; ⇒ [[0 1 2] [3 4 5] [10 20 30] [40 50 60]]
```

```
(@append (Int) (3 3 [])
  [[0 1 2] [3 4 5]]
  [[10 20 30] [40 50 60]])
; ⇒ [[0 1 2 10 20 30] [3 4 5 40 50 60]]
```

Same function, same data:

```
Case A:  $m=2 $n=2 @s=[3]   — cell shapes [2 3]: each WHOLE matrix is a cell
──────────────────────────────────────────────────────────────────────────
  m1 : [2 3] = frame []  ++ cell [2 3]
  m2 : [2 3] = frame []  ++ cell [2 3]
  empty frame → ONE append, joining leading axes of the cells:

        ┌ [ 0  1  2] ┐
        │ [ 3  4  5] │    result cell (2 + 2) ++ [3] = [4 3]
        ├────────────┤
        │ [10 20 30] │    (stacked vertically)
        └ [40 50 60] ┘
```

```
Case B:  $m=3 $n=3 @s=[]    — cell shapes [3]: each ROW is a cell
──────────────────────────────────────────────────────────────────────

  m1 : [2 3] = frame [2] ++ cell [3]
  m2 : [2 3] = frame [2] ++ cell [3]
  frames [2] agree → append runs once per frame position, on row pairs:

   frame 0:  append [0 1 2] [10 20 30]  =  [0 1 2 10 20 30]
   frame 1:  append [3 4 5] [40 50 60]  =  [3 4 5 40 50 60]

  collect: frame [2] ++ cell [6] = [2 6]      (joined horizontally)
```

Notice that `$m` and `$n` changed between the two cases (`2 2` vs `3 3`): they name the leading dimension of the cells, so when the choice of `@s` moves the cell boundary, they move with it.

The ispace arguments never change what `append` does — it always joins the leading axis of its cells. They change where the cell boundary sits, and therefore how much frame is left for the implicit map.

6.2 Rerank by η-expansion

When the function isn't Π-polymorphic in the right way — or when it's an argument to something else (§4.5) — wrap it in a λ whose parameter types name the cells you want. That was the §3.4 pair. The old tutorial's `(~(1 1)+ v m)` is today's:

```
((λ ((x [Int 2]) (y [Int 2])) (+ x y)) [10 100] [[1 2] [3 4]])
; ⇒ [[11 102] [13 104]]
```

The same move re-aims `reduce`. By default `reduce` collapses the leading axis (its argument is one cell — sum the rows, i.e. collapse columns). To reduce each row independently, η-expand around the rows:

```
((fn ((row [Int 3])) (@reduce (Int) (2 []) + row))
 [[0 1 2] [0 10 100]])
; ⇒ [3 110]
```

The λ declares `[3]`-cells, so the matrix becomes a `[2]`-frame of rows, and each row gets its own reduction. Compare `(@reduce (Int) (1 [3]) (fn ...) m)`, which is the ispace-choice way of saying "items are `[3]`-vectors" and sums down the columns instead — `[0 11 102]` for this input. Both mechanisms, one function.

---

7. Boxes: Σ in practice

Sometimes a shape cannot be known statically: it depends on computed values (the shape argument of `iota`), on input, or it simply varies across elements (ragged data — the day names of a week). The type system's answer is the existential binder `Σ`: a **box** packages an array together with the ispaces its type abstracts over, and the type of the box hides them.

`(box (i ...) e T)` is an atom: `e` is the payload, the `i ...` are the witness ispaces, and `T` is the `Sigma` type ascribed to the box. `unbox` opens one, binding the witnesses and the payload in a body expression:

```
(let ((val (my-box : (Sigma ($d) (A Int (dims $d))))
        (box (3) [1 2 3] (Sigma ($d) (A Int (dims $d))))))
  (unbox ($d xs my-box) xs))
; ⇒ [1 2 3]
```

Inside the `unbox` body, `$d` is a real ispace variable — you can `i-app` things with it. You cannot learn anything about it statically (that's the point), but you can pass it along to functions that need it. The dynamic `iota` is the canonical producer: its type

```
iota : Π ($d) (-> ([Int $d]) (Σ (@s) [Int @s]))
```

takes a value-level shape vector (length `$d`) and returns a boxed array of some shape `@s`. Open it and hand the witness to `sum` (`sum : Π (@s) (-> ([Int @s]) Int)`):

```
(unbox (@s contents (@iota _ (2) [2 3]))
  ((i-app sum @s) contents))
; ⇒ 15
```

That example is the whole boxes story in miniature: the shape `[2 3]` was a run-time value, so the array's type couldn't promise it — but the witness `@s` still lets typed code consume the payload.

Contrast `iota/static : Π (@s) [Int @s]` — not a function at all, but a Π-family of arrays, usable whenever the shape is statically known, with no box to unwrap:

```
(i-app iota/static [2 3])
; ⇒ [[0 1 2] [3 4 5]]
```

Prefer `iota/static` whenever you can; reach for `iota` + `unbox` only when the shape is genuinely dynamic. (This is the current language's version of the old tutorial's "avoid `iota`" advice.)

7.1 Ragged data: a frame of boxes

The 2019 tutorial's plural `boxes` form is gone, but you don't need it: `box` is an atom, so a bracket of boxes is an array of them. The weekdays example, modernized — with day names as strings (Int vectors!) of different lengths behind a uniform `Σ` type, and a lifted function unboxing each:

```
(let ((type *str (Sigma ($n) [Int $n]))
      (fun (strlen (s *str) : Int)
        (unbox ($n cs s) (@length (Int) ($n []) cs))))
  (strlen [(box (6) "Monday" *str)
           (box (7) "Tuesday" *str)
           (box (9) "Wednesday" *str)
           (box (8) "Thursday" *str)
           (box (6) "Friday" *str)]))
; ⇒ [6 7 9 8 6]
```

`strlen` consumes scalar `*str` cells, so it lifts over the `[5]`-frame of boxes; each application opens its own box with its own private witness `$n`. (The original computed `[#t #f #f #f #t]` by comparing with 6 — no comparisons in the current prelude, so we report the lengths themselves.)

---

8. The prelude

The entire prelude — every name the interpreter actually provides — with verified examples. Types are written with the sugar of §2; remember every entry is `∀`-then-`Π` nested, eliminated as `(@name (types) (ispaces) args ...)`.

8.1 Arithmetic (monomorphic, scalar)

Name	Type
`+ - *`	`(-> (Int Int) Int)`
`f.+ f.- f.* f./ f.^`	`(-> (Float Float) Float)`
`sqrt`	`(-> (Float) Float)`
`i->f`	`(-> (Int) Float)`
`truncate round ceiling floor`	`(-> (Float) Int)`

Int and Float arithmetic are disjoint families — no overloading, no coercion except explicit `i->f`. There is no Int division, no `expt`, and (worth repeating) no comparisons. All of them lift, of course:

```
(truncate (f.* (i->f 3) 1.5))
; ⇒ 4
```

```
(sqrt (f.* 4.5 2.0))
; ⇒ 3.0
```

8.2 Shape surgery

```
head    : ∀ (&t) Π ($d @s)    (-> ([&t (+ 1 $d) @s]) [&t @s])
tail    : ∀ (&t) Π ($d @s)    (-> ([&t (+ 1 $d) @s]) [&t $d @s])
length  : ∀ (&t) Π ($d @s)    (-> ([&t $d @s]) Int)
reverse : ∀ (&t) Π ($d @s)    (-> ([&t $d @s]) [&t $d @s])
append  : ∀ (&t) Π ($m $n @s) (-> ([&t $m @s] [&t $n @s]) [&t (+ $m $n) @s])
flatten : ∀ (&t) Π ($m $n @s) (-> ([&t $m $n @s]) [&t (* $m $n) @s])
transpose2d : ∀ (&t) Π ($m $n) (-> ([&t $m $n]) [&t $n $m])
```

All of these consume their argument whole (the argument's full shape is the cell — that's what the `$d @s` split in the types says), and all operate on the leading axis (except `transpose2d`, which is matrix-only). The `(+ 1 $d)` in `head`/`tail` is how the type demands a non-empty leading axis: a length must be expressible as `1 + $d`.

```
(@head (Int) (3 []) [3 1 4 1])
; ⇒ 3
```

```
(@tail (Int) (3 []) [3 1 4 1])
; ⇒ [1 4 1]
```

```
(@length (Int) (4 []) [3 1 4 1])
; ⇒ 4
```

```
(@reverse (Int) (4 []) [3 1 4 1])
; ⇒ [1 4 1 3]
```

```
(@flatten (Int) (2 3 []) [[1 2 3] [4 5 6]])
; ⇒ [1 2 3 4 5 6]
```

```
(@transpose2d (Int) (2 3) [[1 2 3] [4 5 6]])
; ⇒ [[1 4] [2 5] [3 6]]
```

Note how the instantiation encodes the argument's shape: for `head` on a 4-vector of scalars, `(1 + $d) ++ @s = [4]` forces `$d = 3`, `@s = []`. When the leading dimension is what's being measured (`length`), `$d` is the dimension itself. `length` is also the bridge from the ispace world to the value world: it turns a static dimension into an `Int` you can compute with.

8.3 Reduction and indexing

```
reduce  : ∀ (&t) Π ($d @s)
          (-> ((-> ([&t @s] [&t @s]) [&t @s])      ; op
               [&t (+ 1 $d) @s])                   ; (1+$d) items of shape @s
              [&t @s])

fold    : ∀ (&t &t2) Π ($d @s @s2)
          (-> ((-> ([&t2 @s2] [&t @s]) [&t2 @s2])  ; op : acc, item -> acc
               [&t2 @s2]                           ; initial accumulator
               [&t (+ 1 $d) @s])                   ; (1+$d) items of shape @s
              [&t2 @s2])

sum     : Π (@s) (-> ([Int @s]) Int)               ; no ∀ — Int only

index2d : ∀ (&t) Π ($m $n) (-> ([&t $m $n] [Int 2]) &t)

f.reduce3 : (-> ((-> (Float Float) Float) [Float 3]) Float)
```

`reduce` combines the items of the leading axis with an associative operator (§4.5 showed both the scalar-cell and vector-cell cases; §6.2 showed reranking it). Its `(+ 1 $d)` makes empty reductions a type error — there is no `reduce/zero` in the current prelude.

`fold` is the general accumulator version (operator takes `(acc, item)`, left to right). **Caveat, verified:** the current interpreter only implements `fold` correctly for scalar items and accumulator (`@s = @s2 = []`). With vector items it folds over the flattened atoms — `(@fold (Int Int) (1 [3] [3]) (fn ((a [Int 3]) (e [Int 3])) (+ a e)) [0 0 0] [[1 2 3] [10 20 30]])` returns `[66 66 66]`, not the `[11 22 33]` its type promises. Use `reduce` for non-scalar items.

```
(@fold (Int Int) (3 [] []) + 0 [1 2 3 4])
; ⇒ 10
```

`sum` adds up all atoms of an Int array of any shape — handy with a Σ witness (§7):

```
((i-app sum [2 3]) [[1 2 3] [4 5 6]])
; ⇒ 21
```

`index2d` fetches one element of a matrix — and because its index parameter is a `[Int 2]` cell, an array of index pairs lifts it into a gather:

```
(@index2d (Int) (2 3) [[1 2 3] [4 5 65]] [1 2])
; ⇒ 65
```

```
(@index2d (Int) (2 3) [[1 2 3] [4 5 65]] [[0 0] [1 2]])
; ⇒ [1 65]
```

(Indexing remains a communication-heavy operation to be used in bulk, not a way to visit elements one at a time — the 2019 tutorial's advice stands.)

8.4 Generators, debugging, escape hatches

```
iota/static : Π (@s) [Int @s]                ; an array family, not a function (§7)
iota        : Π ($d) (-> ([Int $d])
                         (Σ (@s) [Int @s]))  ; dynamic shape, boxed result (§7)
undefined   : ∀ (&t) Π (@s) [&t @s]          ; typechecks at any type; crashes if
                                             ;   evaluated — for type experiments
trace       : ∀ (&t &r) Π (@s @q)
```

```
              (-> ([&t @s] [&r @q]) [&r @q]) ; prints first arg, returns second
trace-file  : like trace, with a filename    ; appends to that file instead
              (an Int vector) prepended
read-file   : ∀ (&t) Π ($d @s)
              (-> ([Int $d]) [&t @s])        ; parse + run the named file;
                                             ;   you assert the result type
```

8.5 Names that typecheck but never run

Source files in the wild (the darknet examples especially) mention names like `replicate/i/3x3-2`, `iota/608`, `flatten/f/3-9-_`, `append/f/608-1-610`, `reduce/f/26`, `transpose2d/f/m-n`, `undefined-input/...`. These are name-encoded monomorphic hacks: a helper in the type checker (`hackyPrelude` in `src/TypeCheck.hs`, marked `FIXME: delete once monomorphization done`) assigns any such name a type parsed out of the name itself, and the Futhark backend lowers them — but the interpreter only knows the handful with real prelude entries (`iota/3`, `iota/608`, `iota/610`, `reduce/f/26`, `index2d/f/610`, four `append/f/...`, five `flatten/f/...`, `transpose/f/27-32`). Everything else in that family typechecks and then dies at evaluation. For interpreter work, treat the whole family as off-limits; there is no general `replicate` builtin at all — define your own (§9.1).

---

9. Worked examples

All verified, all self-contained — paste any block into `remora interpret`.

9.1 `replicate`, from nothing

There is no replicate builtin, but the lifting machinery is a replicator — replication into a frame is exactly what happens to a smaller-framed argument. So: manufacture a throwaway array with the frame you want (`iota/static`), and map a constant function over it.

```
(let ((fun (@replicate (&t) (@f @s) (in [&t @s]) : [&t @f @s])
       ((array [] (fn ((x [Int])) in)) (i-app iota/static [@f]))))
  (@replicate (Int) ([2 3] [4]) [1 2 3 4]))
; ⇒ [[[1 2 3 4] [1 2 3 4] [1 2 3 4]]
;    [[1 2 3 4] [1 2 3 4] [1 2 3 4]]]
```

Read the body inside-out: `(i-app iota/static [@f])` is an Int array of shape `@f`; the constant function takes scalar Int cells, so that array contributes frame `@f`; each application returns `in` (shape `@s`); and collecting results yields `@f ++ @s`. The iota's values are never used — it is purely a shape donor driving the implicit map.

9.2 Vector magnitude

The 2019 tutorial's running example `vmag`, in current dress:

```
(let ((fun (vmag (v [Float 2]) : Float)
       (sqrt (@reduce (Float) (1 []) f.+ (f.* v v)))))
  (vmag [[3.0 4.0] [6.0 8.0]]))
; ⇒ [5.0 10.0]
```

`(f.* v v)` lifts the scalar multiply across the vector (squares), `reduce` sums, `sqrt` lifts over the scalar result. Applied to a matrix, `vmag`'s `[Float 2]` cells put the rows in a `[2]`-frame: two magnitudes, no loop.

9.3 Polynomial evaluation by Horner's rule

The old tutorial's `fold-right` Horner scheme, adapted: `fold` is a left fold here, so reverse the coefficients first — and note the inner λ closing over `x`:

```
(let ((fun (horner (x Int) : (-> ([Int 3]) Int))
        (fn ((coeffs [Int 3]))
          (@fold (Int Int) (2 [] [])
            (fn ((acc Int) (c Int)) (+ (* acc x) c))
            0
            (@reverse (Int) (3 []) coeffs)))))
  ((horner 2) [2 0 -3]))   ; 2 + 0x - 3x² at x = 2
; ⇒ -10
```

9.4 Matrix multiplication

The original tutorial's showpiece, where rank polymorphism, replication, reduction, and reranking all land at once. Start with vector-times-matrix. For `v : [Int $n]` and `m : [Int $n $p]`, the scalar `*` gives `v` frame `[$n]` and `m` frame `[$n $p]` — prefix agreement scales row `i` of `m` by v_i — and reducing the leading axis sums the scaled rows into `v·m : [Int $p]`:

```
(let ((i-fun (v*m ($n $p))
        (fn ((v [Int $n]) (m [Int $n $p]))
          (@reduce (Int) ((- $n 1) [$p])
            (fn ((x [Int $p]) (y [Int $p])) (+ x y))
            (* v m)))))
  ((i-app v*m 2 2) [10 100] [[1 2] [3 4]]))
; ⇒ [310 420]
```

Three things to notice: the `i-fun` binding makes `v*m` length-polymorphic; `reduce`'s leading dimension must be written as `1 + d`, so we pass `$d = (- $n 1)` — dimension arithmetic in an instantiation; and the operator is the η-expanded `+` on `[$p]`-cells (§4.5).

Now the punchline, just as in the original: matrix-times-matrix is `v*m` lifted. Give it a matrix as first argument — a `[$m]`-frame of row-vector cells — and each row independently multiplies `b`:

```
(let ((i-fun (v*m ($n $p))
        (fn ((v [Int $n]) (m [Int $n $p]))
          (@reduce (Int) ((- $n 1) [$p])
            (fn ((x [Int $p]) (y [Int $p])) (+ x y))
            (* v m))))
      (i-fun (m*m ($m $n $p))
        (fn ((a [Int $m $n]) (b [Int $n $p]))
          ((i-app v*m $n $p) a b))))
  ((i-app m*m 2 2 2) [[1 2] [3 4]] [[5 6] [7 8]]))
; ⇒ [[19 22] [43 50]]
```

`m*m`'s body contains no computation at all — only a type-level re-description of `a` as a frame of rows. That is rank-polymorphic programming.

9.5 The iteration space: `idxs-of`

The old `indices-of` (reify each position of an array as an index vector) isn't a builtin, but the static version is definable per rank — this is `tests/idxs-of.remora`, lightly annotated:

```
(let ((i-fun (idxs-of/2 ($m $n) : [Int $m $n 2])
        ((fn ((i [Int]))
```

```
          ((fn ((j [Int])) [i j]) (i-app iota/static [$n])))
        (i-app iota/static [$m])))))
  (i-app idxs-of/2 2 3))
; ⇒ [[[0 0] [0 1] [0 2]]
;    [[1 0] [1 1] [1 2]]]
```

Two nested constant-ish maps: the outer λ ranges `i` over `iota [$m]`, the inner ranges `j` over `iota [$n]`, and `[i j]` packs each pair. Combined with the gather form of `index2d` (§8.3), this recovers much of what `rotate/indices-of` did in the old tutorial.

---

10. Porting guide: the 2019 tutorial → current Remora

How to read the original (still excellent for the ideas) against the current implementation.

Syntax correspondences. Superscripts point to the notes below.

2019 tutorial	Current language
`(define x e)`	`(let ((val x e)) ...)`
`(define (f [x 1] [y 0]) e)`	`(fun (f (x [Int $n]) ...) e)` [1]
`(λ ([x 1] [y 1]) e)`	`(λ ((x [Int n]) (y [Int n])) e)`
`~(r1 r2)f` rerank sugar	η-expansion (§6.2) or ispace choice (§6.1)
`(t-app (i-app f i ...) T)`	`(i-app (t-app f T) i ...)` [2]
— (no equivalent)	`(@f (T ...) (i ...) args ...)` [3]
type vars `t`, array vars `@t`	atom vars `&t`, array vars `*t` [4]
`(shape 3 4)`	`(dims 3 4)`, or splice `[3 4]`
`int`, `bool`, `float`	`Int`, `Bool`, `Float`
`(boxes (i ...) T [n] ...)`	a bracket of `box` atoms (§7.1)
`(box ((len 3)) [int len] e)`	`(box (3) e (Sigma ($len) [Int $len]))` [5]
`(unbox arr (x len) body)`	`(unbox ($len x arr) body)` [6]
`iota0`…`iota9`, `indices-of/N`	not provided [7]

Notes: [1] rank annotations become full types, and a leading `i-fun` adds length-polymorphism; the whole thing lives in a `let`. [2] nesting flipped — `∀` is now outermost. [3] combined-application sugar (new); no 2019 equivalent. [4] and `@` as a variable sigil now always marks a shape (its other use is the `(@f ...)` prefix above). [5] the `Sigma` type is written out in full, and comes last. [6] witnesses and value-binder share one head, witnesses first. [7] use `iota/static`, dynamic `iota`, or a hand-rolled `idxs-of/N` (§9.5).

Whole features of the old dynamic dialect with no current counterpart (the type system parts of the old tutorial fare much better than the library parts; the dialect itself lives on, unmaintained, as `#lang remora/dynamic` in the Racket implementation at github.com/jrslepak/Remora): `if/cond` and all conditional code; comparisons and boolean operators (`#t`/`#f` parse, nothing consumes them); `filter`, `partition`, `select`, `count-replicate`, `grade`, `sort`; `rotate`, `index`, `index-item`, `subarray` and friends; `scan/iscan/open-scan` (all scans), `fold-right`, `reduce/zero`, `with-shape`, `expt`, `zero?`; characters and real strings. Where the original leans on these (its §§ on conditionals, indexing, sorting, and the scan-based poly-eval), read for the concepts and expect to write the data-flow differently — or not at all — today.

Semantics that did carry over intact: everything in the original's first half — frames, cells, frame agreement, principal-frame replication, function-position arrays, `reduce`'s leading-axis behavior, the `v*m`/`m*m` story — behaves identically (this edition's examples are the proof), and the original's

typed half (§§ "Explicitly typed Remora" onward) describes the same ∀/Π/Σ machinery the current language makes you write everywhere.

Appendix A. The doctest harness

Every `remora`-fenced block in this file is executable. The harness:

```
#!/usr/bin/env bash
# remora-doctest.sh FILE.md — verify every ```remora block.
# A block's expected output is its trailing "; ⇒ ..." comment lines
# (continuation lines start "; ").
set -u
md=${1:?usage: remora-doctest.sh FILE.md}
tmp=$(mktemp -d); trap 'rm -rf "$tmp"' EXIT
awk -v dir="$tmp" '
  /^```remora$/ {n++; f=dir "/" sprintf("%03d",n); inblk=1; next}
  inblk && /^```$/ {inblk=0; next}
  inblk {
    if ($0 ~ /^; ⇒/) { sub(/^; ⇒ ?/,""); print > (f ".want"); want=1 }
    else if (want && $0 ~ /^;    ?/) { sub(/^;  */,""); print > (f ".want") }
    else { print > (f ".remora"); want=0 }
  }' "$md"
fail=0
for f in "$tmp"/*.remora; do
  base=${f%.remora}
  got=$(remora interpret -f "$f" 2>&1)
  want=$(tr "\n" " " < "$base.want" 2>/dev/null | tr -s " " | sed 's/ $//')
  gotn=$(printf "%s" "$got" | tr "\n" " " | tr -s " " | sed 's/ $//')
  if [ "$gotn" != "$want" ]; then
    fail=1; echo "FAIL $(basename "$base"):"; echo "  want: $want"; echo "  got:  $got"
  fi
done
[ "$fail" = 0 ] && echo "all $(ls "$tmp"/*.remora | wc -l | tr -d ' ') blocks pass"
exit $fail
```

Run as `nix shell ~/remora#remora-wrapped --command ./remora-doctest.sh remora-tutorial-2.md`. (Blocks whose expected output spans multiple `;`-continuation lines are compared with whitespace normalized.)

Appendix B. Surface grammar, condensed

The authoritative grammar is the ABNF at `books/kestrel/remora/grammar.abnf` (derived from `src/Parser.hs`); this is the working subset:

```
program  ::= exp
exp      ::= atom | [exp ...] | id | "string"
           | (array shape-lit atom ...) | (array shape-lit type)
           | (frame shape-lit exp ...)  | (frame shape-lit type)
           | (exp exp ...)                          ; application
           | (t-app exp type ...) | (i-app exp ispace ...)
           | (@ exp (type ...)|_ (ispace ...)|_ exp ...)
           | (unbox (ispace-var ... id exp) exp)
           | (let (bind ...) exp)
```

```
atom     ::= #t | #f | int | float
           | (fn|λ ((id type) ...) exp)
           | (t-fn|tλ (tvar ...) exp) | (i-fn|iλ (ispace-var ...) exp)
           | (box (ispace ...) exp type)
bind     ::= (val id exp) | (val (id : type) exp)
           | (fun (id (id type) ... [: type]) exp)
           | (fun (@id (tvar ...)|_ (ispace-var ...)|_ (id type) ... : type) exp)
           | (t-fun (id (tvar ...) [: type]) exp)
           | (i-fun (id (ispace-var ...) [: type]) exp)
           | (type tvar type) | (extent ispace-var ispace)
type     ::= &id | *id | Int | Bool | Float
           | [type ispace ...]                      ; (A type [ispace ...])
           | (A type shape)
           | (->|→ (type ...) type)
           | (Forall|∀ (tvar ...) type)
           | (Pi|Π (ispace-var ...) type) | (Sigma|Σ (ispace-var ...) type)
dim      ::= $id | nat | (+ dim ...) | (* dim ...) | (- dim ...)
shape    ::= @id | dim | (dims dim ...) | (++ shape ...) | [ispace ...]
ispace   ::= dim | shape
tvar     ::= &id | *id
ispace-var ::= $id | @id
```