The Name in the Bracket

einlang

2026-05-18

Contents

Part I · Primitives

Chapter 1 · The Ghost in the Name

Primitives · Naming and permutation

> Every time you write `dim=-1`, you know what it means. The compiler doesn't. This is about what happens when it does.

Here is a story about a bug.

A tensor has shape `(32, 64, 256)`. The data loader author knows these dimensions are `batch`, `channel`, and `spatial`. There is a comment. There is a variable name: `spatial_features`. Then:

```
x = x.mean(dim=1)
```

`dim=1` erases a dimension. At the time of writing, position 1 holds `channel`. The intent is "average over channels." The text says `dim=1`. A position. A number.

Three months later, the data pipeline is refactored. Channel moves to position 2. The new shape is `(32, 256, 64)`. `mean(dim=1)` now silently erases `spatial`. No errors. No warnings. The loss descends. The model deploys. The customer complaint arrives on Thursday.

The notation had no slot for the fact that would have caught it. The fact—"erasing `channel`, not `spatial`"—was in a comment, in a variable name, in the author's head. It was absent from the one place the compiler could see: the source text of the operation itself.

Positional notation is not wrong. It is insufficient. It records the arithmetic of shapes. It does not record the identity of coordinates. When a shape is correct but a coordinate is wrong, positional notation gives no place to notice. When shapes and types were both correct, what information was missing?

A coordinate has three properties. The framework records two of them. The third—the name—exists only in the programmer's head, in comments, or in variable naming conventions.

Two columns are machine-readable. One column is not. That dashed line is the shape-meanings gap—the space between what the framework verifies and what the programmer intended. The shape says *how many*. The role says *which one*. The framework checks the first. Only you know the second.

Now imagine a different notation:

```
let y[b, s] = mean[channel](x[b, channel, s]);
```

The bracket after `mean` names the coordinate being **consumed**—eliminated from the output, its values collapsed into a single number. The brackets after `y` and `x` name the coordinates that survive. That `channel` exists on `x` is statically checked. The reader sees the consumption without reconstructing it. The fact that was previously in a comment—"average over channels"—is now in the syntax, where it can be enforced and the reader can audit it.

Now consider the reverse situation. Instead of eliminating a coordinate, suppose a value needs to be copied *along* one:

```
let out[b, c, s] = x[b, c, s] + bias[c];
```

`bias` is indexed only by `c`. It has no `b` and no `s` in its brackets. The absence declares: `bias` is silent on `b` and `s`. Its value is copied across every batch element and every spatial position. Not because broadcasting

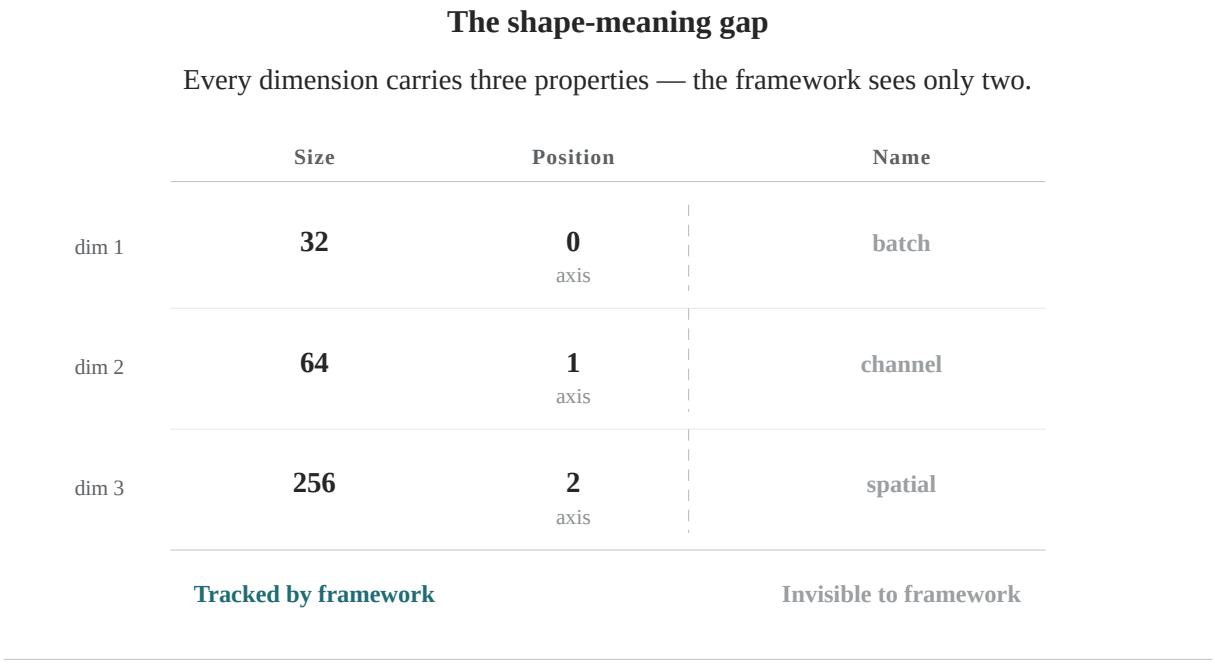


Figure 1: Three properties of a coordinate: domain and position are checked; the name is not

is a convenient default. Because the indexing pattern makes a semantic claim—`bias` does not depend on the batch or the spatial coordinate—and that claim is honored.

Think of a tensor as a person holding a **repeater** (a megaphone). `bias[c]` speaks on coordinate `c`: at `c=0` it says one value, at `c=1` another. On every coordinate not in its brackets—`b`, `s`—it says nothing. Silence. The notation, encountering this silence in the indexing pattern, fills it by repeating the value. The repetition is not a convenience feature. It is the notation honoring the promise that `bias` made by omitting those coordinates: "my value is independent of `b` and `s`. Ask me a thousand times with different `b`, and I give the same answer."

This is the megaphone model: a tensor speaks on the coordinates in its brackets, and stays silent on all others. Broadcasting is repeating the silent message wherever it is asked. Reduction is the inverse—pointing the megaphone at a coordinate and speaking it out of existence.

A coordinate has three properties. First, a **name**: `batch`, `channel`, `time`, `feature`. The name carries the semantic role. Second, a **domain**: the set of values the coordinate can take. For a tensor of shape `(32, 64, 256)`, the `batch` coordinate ranges from `0` to `31`, `channel` from `0` to `63`, and `spatial` from `0` to `255`. Third, a **position**: where this coordinate sits in the tensor's shape tuple. In `(32, 64, 256)`, `batch` is at position 0, `channel` at position 1, `spatial` at position 2.

Positional notation records only the domain and the position—`(32, 64, 256)` tells you the sizes and their order, but not their names. Named notation records all three: `[batch: 32, channel: 64, spatial: 256]`.

When you write `x.mean(dim=1)`, you are asking the position to stand in for the name. It works until the position changes. When you write `mean[channel](x)`, you are using the name directly. The position

becomes an implementation detail—the compiler's problem, not yours.

An Analogy: The Parking Lot

You park your car in Row D, Slot 7. The ticket in your pocket says "D-7." You return after dinner to find the lot has been repainted. The rows now run perpendicular to their old orientation. Row D is now somewhere else entirely. Your ticket, which records a *position* in a fixed coordinate system, sends you to the wrong car.

The lot's *shape* hasn't changed. It is still an 8×20 grid. A shape checker would tell you everything is fine. But the *role* of each row—which row is "D"—has moved.

This is what happens when you write `x.transpose(1, 2)`. The shape is still `(32, 256, 64)`. A shape checker sees the same three numbers. But the positions have been reassigned. Dimension 1 is no longer `channel`. Dimension 2 is no longer `spatial`. The ticket in your pocket—`dim=1`—now points to the wrong car.

A named-coordinate notation is like a ticket that says "the blue Honda Civic" instead of "D-7." The car may move, but the description finds it.

Now extend this analogy. Imagine the parking lot has three underground levels—B1, B2, B3—each an 8×20 grid. Your ticket says "D-7-B1": row D, slot 7, basement level 1. You return to find the lot has been renovated. The levels have been renumbered—B1 is now B3. The rows on each level have been rotated 90 degrees. Slot numbers run in the opposite direction. Your ticket now points to a space that doesn't exist, or worse, to someone else's car.

A shape checker would tell you the lot still has three levels of 8×20. Correct. The shape is the same. A position checker would tell you "D-7-B1" is a valid coordinate in the new system—it's just a *different* space. Also correct. Neither checker can tell you that your ticket describes the wrong space, because neither checker knows which space is yours.

This is what happens when you write `x.permute(1, 2, 0)` on a tensor with shape `(32, 64, 256)`. The shape after permutation is `(64, 256, 32)`. A shape checker sees three numbers and approves. But which dimension is batch now? Which is channel? The shape doesn't tell you. The permutation numbers don't tell you. They tell you that dimension 1 moved to position 0, dimension 2 to position 1, dimension 0 to position 2. They don't tell you that `channel` moved to the front, that `spatial` is now in the middle, that `batch` is now last.

A named permutation tells you all of that:

```
let y[c, s, b] = x[b, c, s];
```

You don't need to decode `(1, 2, 0)`. You read `y[c, s, b]` and you know: channel first, then spatial, then batch. If the lot gets renovated—if the upstream tensor changes its internal order—the named permutation still finds your car, because it looks for the blue Honda Civic, not D-7-B1.

The 3D parking lot teaches something the 2D version couldn't: when you compose multiple layout changes—renumbering levels AND rotating rows AND reversing slots—the positional ticket becomes wrong in multiple independent ways. Each way must be diagnosed separately. The named ticket is right about all of them simultaneously, because it never depended on any of them to begin with.

Permutation: Moving Without Losing

A permutation changes the order of dimensions without changing any values. It is the simplest tensor operation there is—no arithmetic, no reduction, just relabeling positions.

It is also a reliable source of 11 PM debugging sessions.

The problem is not that permutation is hard. The problem is that positional permutation describes *mechanics* rather than *intent*. Here is a concrete example. An image-processing pipeline takes input in `(batch, height, width, channel)` and needs it in `(batch, channel, height, width)`:

```
x = x.permute(0, 3, 1, 2)
```

The programmer writes this while looking at a diagram that says "channel moves from position 3 to position 1." The diagram is correct. The code is correct. Six months later, upstream changes its output convention to `(batch, width, height, channel)`. Height and width have swapped. `permute(0, 3, 1, 2)` still executes without complaint. Channel still ends up at position 1—correct. But height and width are now in positions the programmer did not intend. The shapes are identical. The values are wrong.

No shape checker catches this. No type checker catches this. The bug will surface in production as "the model is slightly worse on images with non-square aspect ratios," and it will take a human being several hours to trace the silent swap back to this one line.

The root cause: `(0, 3, 1, 2)` describes a rearrangement of *positions*. What the programmer needed to describe was a rearrangement of *identities*—"move the dimension called `channel` to the front, and keep everything else in order."

Here is the same refactoring under both notations:

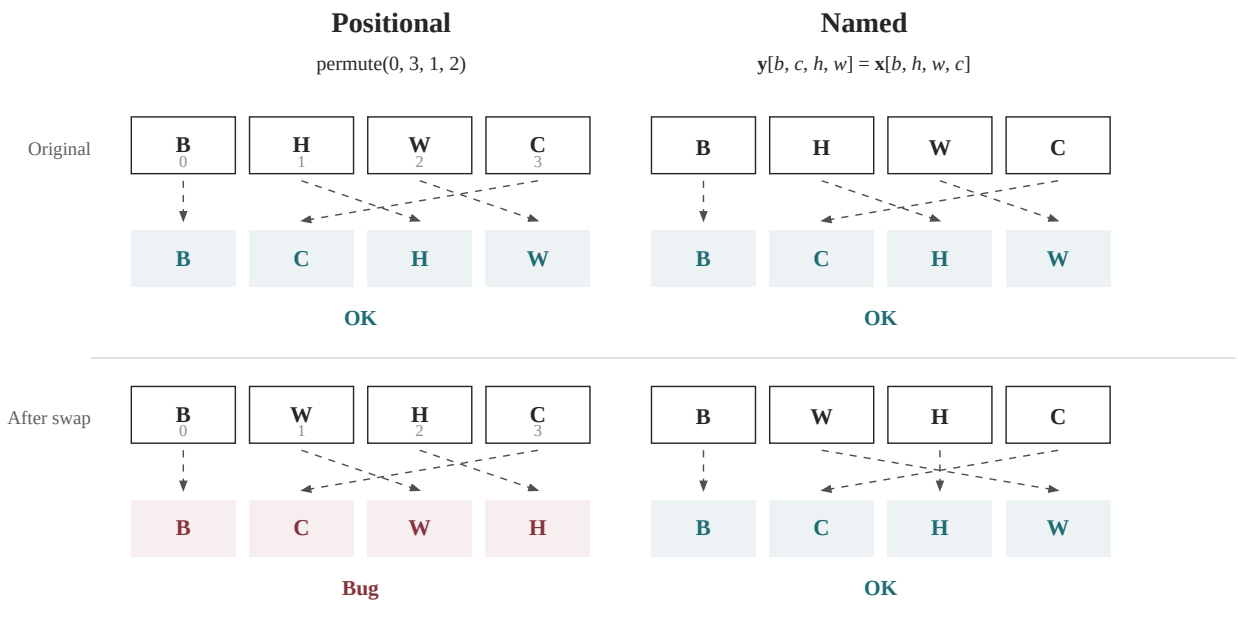


Figure 2: Same refactoring, two notations. On the left: positional permute. On the right: named.

The figure tests both notations against a common refactoring. Top row: the original pipeline maps BHWC to BCHW. On the left, `permute(0,3,1,2)`—read "old axis 0 stays at 0, old axis 3 moves to 1,

old axis 1 moves to 2, old axis 2 moves to 3"—produces the correct result. On the right, the named expression `y[b,c,h,w] = x[b,h,w,c]` produces the same correct result. Both pass. Bottom row: upstream swaps height and width, so the input is now BWHC. The positional instruction executes identically—`permute(0,3,1,2)` is still the same four numbers—but the output is now B,C,W,H. Height and width are silently exchanged. The named expression `y[b,c,h,w] = x[b,h,w,c]` adapts automatically: `h` maps to the second axis in the input regardless of where height landed, `w` maps to the third. The instruction did not change. The meaning did.

Einops addresses this with a string-based notation:

```
y = rearrange(x, "batch height width channel -> batch channel height width")
```

This is better. The names survive renaming of upstream positions, because `rearrange` matches by name, not by index. But the string is still a string. The names `height` and `width` are not checked against any declaration. They are local to this one call. If the tensor actually contains `time` rather than `height`, the string won't catch it—it will happily treat `time` as if it were `height`, because the names in the string are just pattern variables, not coordinate declarations.

What we want is for the coordinate names to be **checked facts**, not comments embedded in syntax:

```
let y[b, c, h, w] = x[b, h, w, c];
```

This is an Einlang rectangular declaration. The left-hand side declares the output coordinates. The right-hand side indexes the input by those same coordinate names. `b` appears on both sides in the same position—it survives unchanged. `h` appears on the left at position 2 and on the right at position 1—it has been moved. Every coordinate on the right is checked to exist on `x`, and every coordinate on the left must appear somewhere on the right.

You don't need a `permute` function. You don't need a `rearrange` string. You just write where each coordinate goes, and the movement is inferred. The code says *what you want*, not *how to achieve it.*

This is a pattern that will recur through the entire book: **when coordinate names appear in the syntax, operations become self-documenting.** The same line of code that instructs the machine also informs the reader. There is no separate channel of documentation that can drift out of sync.

Before you move on, try this. Find a `permute`, `transpose`, or `swapaxes` in your own code — any line where you rearranged dimensions by position. Translate it into the named form: `y[coords] = x[coords]`. Did you have to look up the dimension order to know which coordinate goes where? The lookup is the bug surface. The named form removes it.

For a compiler pass that only needs to know "move this stride to that position," positional permutation is the right abstraction. But source code is not written for compilers. It is written for the human who will debug it at 11 PM, three months after the original author left the team. That human needs to know *what moved where and why.* Position numbers answer the first question, but not the second. Names answer both.

> Every example in this book runs. The compiler is at **github.com/einlang/einlang** — clone it, open Chapter 1, and start typing.

Chapter 2 · The Megaphone's Promise

"The absence of a signal is itself a signal."

— Geoffrey Hinton (apocryphal)

Primitives · Reduction and broadcasting

A permutation moves coordinates around. A reduction makes one disappear. A broadcast makes one appear where it wasn't.

Reduction and broadcasting are inverses. They govern which coordinates a value depends on—and which it doesn't. The intuition: **a tensor is a speaker that speaks on some coordinates and stays silent on others.** The ones it stays silent on, it gets copied. The ones it speaks on, it can be summed away.

Chapter 1's parking lot showed what happens when a position moves: your ticket says D-7 but the car that was there is gone. Now face the more common case. The position didn't move. The car did. You sit in the same seat, but the class changed from math to history. `dim=1` still points to axis 1—but axis 1 used to be `channel`, and now it's `spatial`. The number is stable. The meaning drifted.

This is the megaphone model. Once you have it, you have the core of every tensor computation.

What Is a Tensor?

Ask a framework documentation and it will tell you: a multidimensional array. Ask a tensor's `.shape` attribute and it will tell you: `(32, 64, 256)`. Ask a compiler and it will tell you: a pointer to a contiguous block of memory with strides and a dtype.

All true. All missing the point.

A tensor is a function from coordinates to values. You give it a `batch` index, a `channel` index, and a `spatial` index; it gives you back a number. The three coordinates together form an address. Every element in the tensor lives at exactly one address.

This definition is not exotic. It is how mathematicians have written tensor operations for a century:

$$C_{ij} = \sum_{k} A_{ik} B_{kj}$$

The letters `i`, `j`, and `k` are not axis numbers. They are coordinate names. `i` walks the rows of `A`. `j` walks the columns of `B`. `k` walks the dimension they share—the one that gets summed away. You can rename `i` to `row`, `j` to `col`, `k` to `inner`, and the mathematics is unchanged.

Now look at how we write the same operation in a modern framework:

```
C = torch.matmul(A, B)
```

Where are `i`, `j`, and `k`? They are gone. The names that gave the operation its meaning are not present in the source text. The compiler knows the shapes of `A` and `B`. It checks that the inner dimensions agree. It does not know—cannot know—that `A`'s second axis represents `feature` and not `time`, or that `B`'s first axis represents `feature` and not `vocab_size`. It only knows that both are `64`.

The Megaphone

Imagine a tensor `bias[j]` as a person holding a megaphone. The megaphone is pointed at coordinate `j`. On `j`, the value speaks—`bias[0]` is one number, `bias[1]` is another, each position carries its own meaning. On every other coordinate—coordinates not in the bracket—the megaphone is silent.

What happens when silence meets a coordinate? The value gets copied. If you write:

```
let out[i, j] = A[i, j] + bias[j];
```

`bias` has no `i` in its brackets. It is silent on `i`. This silence is a declaration: "the value of `bias` does not depend on `i`. Whatever `i` you ask for, the answer is the same." So the value is copied across all 32 values of `i`—not because it saves keystrokes, but because the indexing pattern makes a semantic claim: `bias` is independent of batch identity.

This is broadcasting. Not a shape-compatibility hack. A semantic declaration: "this value does not depend on that coordinate." The claim is **statically verifiable**: every use of `bias` is traced, and if any context requires `bias` to vary with `i`, the omission is flagged. Broadcasting is a promise, and the promise is checked.

Now stop. Look at that line again: `let out[i, j] = A[i, j] + bias[j]`. Ask yourself: does `bias` depend on `i`? How do you know?

You probably just looked at the bracket after `bias`. It says `[j]`—no `i`. That is how you know. You compared `bias`'s coordinate set `{j}` against the output's coordinate set `{i, j}` and noticed that `i` is missing. You performed **coordinate set subtraction** in your head, without being taught the procedure.

What you just did—comparing coordinate sets, finding the missing ones—is exactly what can be done by static analysis. Let's give this operation a name:

```
paths(X, Out) = coordinates(Out) \ coordinates(X)
```

`paths(bias, out)` is the set of coordinates in `out` that `bias` is silent on. These are the broadcast coordinates. In the backward pass, they become the reduction coordinates—the coordinates the gradient sums over. Call it the path set. Remember it.

Take the output coordinate set. Subtract each operand's coordinate set. The difference is the coordinates that operand broadcasts over:

```
Output coordinates: {i, j}
A's coordinates:    {i, j}  → paths(A, out) = {}     (no omission)
bias's coordinates: {j}     → paths(bias, out) = {i}  (omitted i)
```

No execution required. The brackets contain all the information needed. Every broadcast is verified consistent across all uses of the broadcast value. If one expression claims `bias` is independent of `i` and another requires it to vary with `i`, it is a coordinate contract violation—caught before a single value is computed. Not magic. Set subtraction.

The Broadcast Derivation

Here is the expression:

```
let out[b, c, h, w] = x[b, c, h, w] + scale[c, w];
```

`x` is a 4D tensor. `scale` is 2D. The output `out` is 4D. Broadcasting must be happening—`scale` has two coordinates, but the output has four. Which coordinates does `scale` broadcast over?

Look at what the brackets contain. The output has `{b, c, h, w}`. `x` has `{b, c, h, w}`. `scale` has `{c, w}`. The difference—output set minus operand set—is the broadcast:

```
Output coordinates:  {b, c, h, w}
x's coordinates:     {b, c, h, w}  → broadcasts over: {}      (no omission)
scale's coordinates: {c, w}        → broadcasts over: {b, h}  (omitted b and h)
```

No arithmetic, no execution. The brackets contain the answer. `scale` broadcasts over `b` and `h`. It is silent on batch and height. It speaks on channel and width.

What just happened: you compared coordinate sets. `x`'s brackets contain all four output coordinates—nothing missing. `scale`'s brackets contain only `{c, w}`—the missing two, `{b, h}`, are the broadcast. This is coordinate set subtraction, performed directly from the brackets, without shape arithmetic.

Now the crucial follow-up question: **is this broadcast semantically correct?**

`scale[c, w]` declares that the scale factor depends on channel and width, but not on batch or height. Does that make sense for your application? Maybe. Maybe `scale` should actually depend on `h`—perhaps it's a height-dependent normalization factor. If it should depend on `h`, the broadcast over `h` is a bug.

Here's the key: in the Einlang version, you can *see* the broadcast and *audit* it. The omission of `h` from `scale[c, w]` is visible. You can ask: "should `scale` really be silent on `h`?" The question has a place to land.

Derive it yourself. Before continuing, take this expression:

```
let out[batch, class] = logits[batch, class] - max_logit[class];
```

`logits` has coordinates `{batch, class}`. `max_logit` has `{class}`. Output has `{batch, class}`. What coordinate does `max_logit` broadcast over? Is the broadcast semantically justified? What should the gradient sum over?

Now look at the positional equivalent:

```
out = x + scale[:, None, :, None]  # or some permutation of None
```

Where does `scale` broadcast? You'd need to decode the `None` positions, map them to the dimension order of `x`, and then check whether those dimensions are the ones `scale` should be silent on. If the dimension order of `x` changes, the `None` positions must change. The broadcast is there, but it's encoded as a shape manipulation, not as a semantic claim.

The Einlang version records the claim: `scale` is silent on `b` and `h`. The claim is visible. The claim is auditable. If the claim is wrong, the reader can see it. If the claim is right, the reader can verify it. Either way, the information is in the notation.

That is coordinate set subtraction—mechanical, exhaustible, requiring only the brackets. The compiler does the same thing for every expression in your program, every time, without fatigue. The difference is that the compiler does it for all of them.

Now the inverse. If broadcasting is silence—staying quiet on a coordinate so you are copied along it—reduction is speaking: naming the coordinate you consume, marking what disappeared.

```
let total = sum[i](data[i]);
```

`sum[i]` picks up the megaphone and points it at `i`. "I am going to speak on `i`—by summing over it. After this line, `i` is consumed." The coordinate `i` appears in the reduction bracket and is absent from the result. `total` is a scalar.

Reduction and broadcasting are the same megaphone, pointed in opposite directions. Broadcasting says "I am silent on `i`—copy me." Reduction says "I am speaking on `i`—consume me."

Draw the megaphone. For each expression below, determine which way the megaphone points. If the megaphone points AT a coordinate, that coordinate is being consumed (reduction). If the megaphone points AWAY from a coordinate, the value is being copied along it (broadcast). If the megaphone is quiet—the coordinate appears in the brackets without a `sum`—the value speaks on that coordinate normally.

1. `let out[i, j] = A[i, j] + bias[j];` — what does `bias`'s megaphone do on `i`? On `j`?
2. `let total = sum[k](data[k]);` — where is the megaphone pointing? What happens to `k`?
3. `let col_max[j] = max[i](matrix[i, j]);` — two coordinates. Which one gets consumed? Which one survives?
4. `let out[b, c, h, w] = x[b, c, h, w] + scale[c, w];` — `scale` has two coordinates but the output has four. Which coordinates does `scale` speak on, and which is it silent on?

The answers are not the point. The point is that you can answer all four questions by looking at the brackets, without knowing the shapes. Broadcasting is silence. Reduction is speech. The bracket tells you which is which.

Rectangular Declarations

To eliminate or broadcast a coordinate, name the ones being kept. In Einlang, you name coordinates with a **rectangular declaration**:

```
let doubled[i, j] = matrix[i, j] * 2.0;
```

The `let` binds a new, immutable tensor. The `[i, j]` on the left declares the output coordinates—the new tensor will have two dimensions, named `i` and `j`. The `matrix[i, j]` on the right indexes the input tensor `matrix` by those same coordinates. It is inferred that `i` ranges from `0` to `matrix.shape[0]` and `j` from `0` to `matrix.shape[1]`.

This is not a loop. It is a declaration. You are stating a fact: "for all `i` and `j` in their respective domains, `doubled[i, j]` equals `matrix[i, j] * 2.0`." Iteration is handled automatically. You handle the meaning.

Reduction

Now the main event. A reduction iterates over a coordinate and combines all the values along it using an associative operation. The coordinate appears in the reduction bracket—and then it is gone from the result:

```
let total = sum[i](data[i]);
```

`sum[i](...)` says: for every value of `i`, evaluate the body `data[i]`, and sum the results. The coordinate `i` is introduced by the `sum`, used in the body, and consumed by the reduction. It does not appear in `total`—`total` is a scalar.

Here is a subtlety that matters later. The `i` in `sum[i](data[i])` is a **local index variable**, not a coordinate identity. If `data` was declared as `let data[k in 0..5] = ...`, then `sum[i](data[i])` still works—`i` is just the name you chose for the loop variable inside this reduction. It does not need to match the name `data` was declared with. `sum[k](data[k])`, `sum[i](data[i])`, `sum[q](data[q])` are all equivalent.

You can even omit the brackets on the tensor entirely: `sum[i](data)`. No `[i]` on `data`—the compiler knows `data` has one coordinate and `sum[i]` consumes it. This is the **implicit** form of Einstein access. The explicit form (`data[i]`) and the implicit form (`data`) are both legal inside a reduction body. The index variables belong to the expression, not to the tensor.

This is different from the **rectangular access** notation used in coordinate-aware functions. When you write `x[..b, class]` inside a function body, `class` is not a local variable—it must match a coordinate that actually exists on `x`. Einstein index variables are scoped to the reduction body. Rectangular coordinates are scoped to the tensor's type. The distinction is invisible in simple examples, but it becomes the foundation of the type-checking system.

The four reduction operations are `sum`, `max`, `min`, and `prod`. Each has an identity element: `sum` starts from `0`, `prod` from `1`, `max` from negative infinity, `min` from positive infinity.

A reduction can leave some coordinates intact—producing a tensor rather than a scalar:

```
let row_sums[i] = sum[j](matrix[i, j]);
let col_sums[j] = sum[i](matrix[i, j]);
```

These two lines produce the same output shape (a 1D tensor of length equal to the surviving coordinate). But they mean completely different things. `row_sums[i]` sums over columns, leaving rows. `col_sums[j]` sums over rows, leaving columns. The difference is entirely in the bracket after `sum`—one character, carrying the full semantic weight of the operation.

In a positional API, these would be `matrix.sum(dim=1)` and `matrix.sum(dim=0)`. The reader must remember which position is rows and which is columns. The code does not help.

The same pattern scales to matrix multiplication. Here is the full picture:

A row of A and a column of B share `k`. The sum consumes it. A single element of C remains. The ledger on the right records the transaction: survivors and consumed. Those are the only two facts any reduction ever produces. The diagram is the ledger, drawn instead of tabulated. The five-step procedure is the ledger, written instead of drawn. They are the same check.

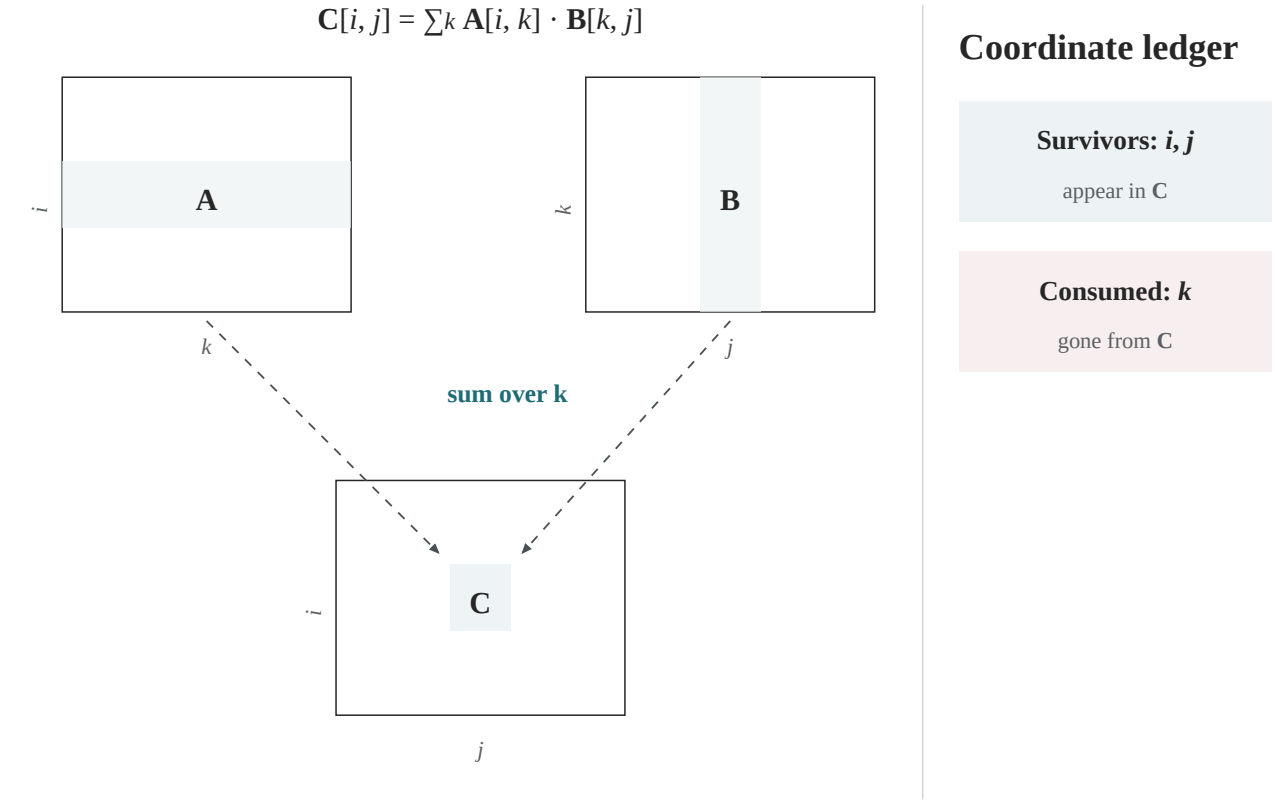
Beyond the four numeric reductions, two more complete the set: `all` and `any`. They are boolean quantifiers. `all[i](x[i] > 0)` asks whether every position along `i` is positive. `any[i](x[i] > 0)` asks whether at least one is. The coordinate `i` is the quantified variable—exactly as in mathematical notation. The quantifier reduces over `i` and produces a boolean scalar. `i` is consumed.

Multi-coordinate quantifiers work the same way:

```
let is_symmetric = all[i, j](matrix[i, j] == matrix[j, i]);
```

`all[i, j]` is the universal quantifier over two coordinates: for all `i` and `j`, check that `matrix[i, j]` equals `matrix[j, i]`. The coordinates `i` and `j` are the bound variables. Both are consumed. The result is a single boolean.

Quantifiers are reductions. Their identity elements are `true` (for `all`) and `false` (for `any`). They compose with the same coordinate set subtraction, the same broadcasting rules, and the same gradient machinery as `sum` and `max`. The bracket names the quantified variable. The notation mirrors the mathematics. The distance is zero.



A row slice of A and a column slice of B converge to a single element of C.

Figure 3: Matrix multiplication with coordinate labels. The ledger on the right tracks survivors and consumed.

Before moving on, try this. `all[i](x[i] > 0)` is true only when every element satisfies the condition. How would you express the same check using only the numeric reductions you already know — `min`, `prod`, `sum`? What about `any[i]` using `max`? Take a minute. Write your answers. Then read on.

`all[i](x[i] > 0)` equals `min[i]((x[i] > 0) as i32) == 1`. It also equals `prod[i]((x[i] > 0) as i32) == 1`. And `sum[i]((x[i] > 0) as i32) == len(x)`.

`any[i](x[i] > 0)` equals `max[i]((x[i] > 0) as i32) == 1`. It also equals `sum[i]((x[i] > 0) as i32) > 0`.

Three different numeric paths to the same boolean result. `all` is a minimum in disguise — false if any element is false. `any` is a maximum in disguise — true if any element is true. The quantifier names the variable. The reduction does the work. Same coordinate. Same consumption. Same ledger.

When Names Match: Broadcast or Contract?

You've seen `A[i, j] + bias[j]` — `j` matches on both sides, so `bias` broadcasts along `i`. You've seen `sum[k](A[i, k] * B[k, j])` — `sum[k]` contracts `k`, consuming it.

Now remove the sum:

```
let C = A[i, k] * B[k, j];
```

What is the output shape? If you come from NumPy einsum, your fingers type `ik,kj->ij` and `k` disappears. But there is no `sum` here. Does `k` contract just because it appears twice?

Before reading on, decide what you think the answer should be.

`C` has shape `(i, k, j)`. Three free indices. No contraction.

In einlang, matching coordinate names **broadcast**. Only an explicit reduction — `sum[k]`, `max[k]`, `prod[k]` — **contracts**. The absence of `sum` means no contraction. The presence of `sum` means contraction. There is no middle ground where matching names silently consume.

In NumPy einsum, the convention `ik,kj->ij` means "repeated indices are summed." Compact. Expressive. Also invisible — you cannot tell from `ik,kj` whether `k` is being broadcast or contracted, because the convention collapses both into the same notation.

In einlang, `A[i, k] * B[k, j]` broadcasts along `k`. `sum[k](A[i, k] * B[k, j])` contracts along `k`. The notation distinguishes the two because the operations are different. Broadcast copies a value across positions that already exist. Contraction eliminates positions and replaces them with a single value. Hiding this difference behind a naming convention costs understanding.

You now know the three operations a coordinate can undergo: it can be left free (survive), consumed by a reduction (contract), or omitted from a term (broadcast). Every coordinate in every expression is in exactly one of these three states. The states are visible in the source. The compiler checks them.

How the Compiler Reads Your Mind

You've been writing `let C = sum[k](A[i, k] * B[k, j])` without specifying the output shape. The compiler infers it. How?

Here are two expressions. They differ only in the order of multiplication:

```
let C = sum[k](A[i, k] * B[k, j]);    // A × B
let D = sum[k](B[k, j] * A[i, k]);    // B × A
```

Before reading on: what shapes do `C` and `D` have? Are they the same?

`C` is `[i, j]`. `D` is `[j, i]`.

The compiler determines output coordinate order by scanning the expression **left to right, depth first, first occurrence**. In `A[i, k] * B[k, j]`, it encounters `i` first (inside `A[i, k]`), then `j` (inside `B[k, j]`). Output: `C[i, j]`. In `B[k, j] * A[i, k]`, it encounters `j` first. Output: `D[j, i]`.

The order you write the indices is the order they appear in the output. The compiler reads your expression in exactly the order you do — and the output layout follows your reading order.

You control the output layout by controlling the order of terms. If you want `C[j, i]`, write `B[k, j] * A[i, k]`. If you want `C[i, j]`, write `A[i, k] * B[k, j]`. No transposition needed. No `permute` call. The layout follows from the expression structure itself.

Every tensor library has a concept of "output shape inference." NumPy einsum requires you to write `ik,kj->ij` — the `->ij` is mandatory if you want control. PyTorch infers shapes from input dimensions. Both treat output layout as something the library decides for you, or something you specify separately from the computation.

Einlang treats output layout as something that falls out of the computation itself. The compiler reads left to right, first occurrence wins. You read left to right. The same rule. No separate notation. No hidden inference. The reading order *is* the layout.

The Two-Column Ledger

A reduction is the most semantically loaded operation in tensor programming. Every reduction makes two claims: which coordinate is being *consumed*, and which coordinates are *surviving*. When you read a reduction, draw an imaginary line down the middle of the page:

Survivors	Consumed
Coordinates that appear in the result	Coordinates introduced in the reduction bracket
They keep their identity	They are gone from the output
They can be used by later operations	They exist only within the reduction body

For `let row_sums[i] = sum[j](matrix[i, j]);`:

- **Survivors**: `i` (appears on the left-hand side)
- **Consumed**: `j` (introduced by `sum[j]`, gone)

Five steps for reading any reduction:

1. **Identify the operation**: `sum`, `max`, `min`, or `prod`—and which coordinates are in its bracket?
2. **Identify the survivors**: which coordinates appear on the left-hand side of the `let`?
3. **Identify the consumed**: which coordinates appear in the reduction bracket but not on the left?
4. **Verify alignment**: do the consumed coordinates index matching positions across all terms in the body?
5. **State the claim**: in one sentence, what does this reduction assert?

This takes five seconds. It catches the bug where `sum[class]` silently became `sum[batch]` after a refactoring.

Broadcasting: The Explicit Omission

Broadcasting is the inverse of reduction. A reduction consumes a coordinate. A broadcast copies along one.

In Einlang, broadcasting is not a shape-compatibility rule that triggers automatically when dimensions happen to align. It is a visible omission in the indexing pattern:

```
let out[i, j] = A[i, j] + bias[j];
```

The coordinate `i` appears on `A` and `out`, but not on `bias`. Its absence from `bias` is the notation for broadcasting: `bias` is indexed only by `j`, so it is replicated across all values of `i`. The code states the semantic claim directly—`bias` does not depend on `i`—and the claim is visible to both the reader and static analysis.

Compare this to the implicit version: `A + bias`. The shapes match. The broadcast happens. But *which coordinate was broadcast over*? You have to know the shapes to answer that. And if the shapes change upstream, the answer changes with them, silently.

This is the principle of explicit omission: **if a term is independent of a coordinate, the indexing should show it.** When the indexing shows it, the reader can audit it. When the indexing hides it, the reader must guess.

Now look at a pair of broadcasts side by side:

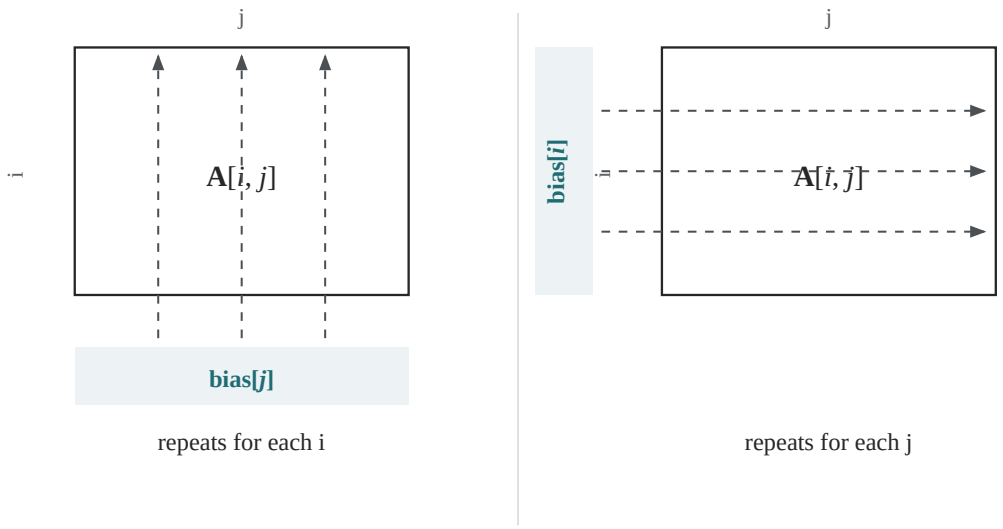
On the left, `bias[j]` omits `i` — the value repeats for each row. On the right, `bias[i]` omits `j` — the value repeats for each column. Both produce `out[i, j]`. The output shape is the same. If you saw only the shape, you could not tell which broadcast happened. The coordinate name in the bracket is the only thing that records the difference.

The verification that follows formalizes what the eye just saw:

Take the output coordinate set. Subtract each operand's coordinate set. The difference for each operand is the set of coordinates that operand broadcasts over:

```
Output coordinates: {i, j}
A's coordinates:    {i, j}  → broadcasts over: {}     (no omission)
bias's coordinates: {j}     → broadcasts over: {i}    (omitted i)
```

This set subtraction is statically computable from the indexing patterns alone—no execution required. The brackets are read, the differences are computed, and every broadcast is verified consistent across all uses of the broadcast value. If one expression claims `bias` is independent of `i` and another expression requires it to vary with `i`, it is a coordinate contract violation—caught before a single value is computed.



Both produce **out**[*i*, *j*]. Identical shape, indistinguishable by shape.

The coordinate name is the semantics.

Figure 4: Two broadcasts. Both produce out[i,j]. One omits i, the other omits j.

Named Rest: `..b`

So far our coordinates have been single, explicit names. But real tensor code often needs to be polymorphic over how many batch dimensions there are. A normalization function shouldn't care whether the input is `(batch, feature)`, `(batch, time, feature)`, or `(batch, head, time, feature)`. It only cares that `feature` is the last dimension and everything else is batch-like.

Einlang provides **named rest indices** for this:

```
let result[..b, j] = x[..b, j] + bias[j];
let row_sum[..b] = sum[j](x[..b, j]);
```

The notation `..b` stands for zero or more adjacent axes, collectively referred to as `batch`. The same rest name must describe the same axis span everywhere it appears within an expression. Which concrete axes `..b` covers is inferred from the shape of `x`.

This is not a wildcard. It is a named group. The name `batch` carries semantic weight—it says "these leading dimensions are all batch-like, and the operation treats them uniformly." If upstream adds a `head` dimension between `batch` and `time`, `..b` absorbs it automatically.

The Where Clause

Sometimes a computation should only apply to a subset of coordinate values. In a positional API, you'd create a mask tensor, multiply, and hope the mask doesn't silently broadcast into the wrong dimension. In Einlang, you attach a **where clause** directly to the declaration:

```
let pos_sum = sum[i](data[i]) where data[i] > 0;
```

The where clause is evaluated for each combination of the enclosing index variables. For reductions, elements where the guard is false are skipped—the reduction's identity element is used instead.

A where clause can also bind intermediate variables to avoid recomputation:

```
let output[i, j] = activated
    where z = sum[k](input[i, k] * weight[k, j]) + bias[j],
          activated = if z > 0.0 { z } else { 0.0 };
```

Without the where clause, you'd write the `sum[k](...)` expression twice. With the where clause, you name the shared subexpression `z` and refer to it in `activated`. The bindings are evaluated in order; later bindings can reference earlier ones.

The where clause is not bolted onto tensor operations. It is the natural extension of the idea that declarations state facts over coordinate domains. A where clause narrows the domain over which the fact holds.

The Inversion Rule

Broadcast and reduction are inverses. What you broadcast over in the forward pass, you reduce over in the backward pass. `bias[j]` omits `i` in the forward direction—broadcast. The gradient `dbias[j] = sum[i](dy[i, j])` consumes `i` in the backward direction—reduction. The omitted coordinate and the consumed coordinate are the same coordinate.

This pairing catches more bugs than any other single rule in this book. If a broadcast is shape-correct but semantically wrong, the gradient will sum over the wrong coordinate. If a reduction consumes `class` but the broadcast was over `batch`, the shapes might still match—but the gradient will silently compute a different quantity.

In PyTorch, write `x = torch.randn(8, 10); b = torch.randn(10); y = x + b`. The bias `b` broadcasts over the first dimension—but nothing in the code says so. If `x` is transposed upstream to shape `(10, 8)`, `x + b` still runs, but now broadcasts over the *second* dimension silently. In Einlang, `let out[b, c] = logits[b, c] + class_bias[c]` makes the broadcast visible: `class_bias` depends only on `c`, so the reader knows it is independent of `b`. The code says what it means.

The Two-Column Ledger Revisited

Every reduction and broadcast can be read through the same lens: which coordinates survive, and which are consumed or copied. The Two-Column Ledger from Section 4 is the tool. Let's apply it to a more complex expression:

```
let norms[i] = sum[j]( (A[i, j] - mean[j](A[i, j])) ** 2.0 ) ** 0.5;
```

Survivors	Consumed
`i` (appears on the left-hand side)	`j` (consumed by `sum[j]` AND `mean[j]`)

Two reductions, both consuming `j`. The ledger tells us: `j` is gone from the output. `i` survives. If `j` should have survived—if this was supposed to be a per-element normalization rather than a row normalization—the ledger catches it. The `j` in the reduction brackets says "I am consuming `j`." The `j`'s absence from the output says "I am gone." The reader can verify the intent against the ledger.

Now read the positional equivalent:

```
norms = ((A - A.mean(dim=1, keepdim=True)) ** 2).sum(dim=1) ** 0.5
```

`dim=1` appears twice. Which coordinate is position 1? The code doesn't say. If `A` is transposed upstream, `dim=1` silently consumes the other coordinate. The two-column ledger for the positional version is empty—there are no names to record in the Survivors and Consumed columns. The ledger exists in the programmer's head. The named version puts it in the syntax.

The ledger is not a tool you run once. It is a reading habit. Every time you see a reduction bracket or an omitted coordinate in an index pattern, draw the line. Write the survivors on the left, the consumed on the right. If the survivors don't match the output coordinates, something is wrong. If the consumed doesn't match what you intended to consume, the bracket is wrong. The ledger catches both errors before the program runs.

The Ledger for Broadcasting and Permutation

The Two-Column Ledger was introduced for reduction. But the same technique works for every tensor operation. The columns change, but the principle—write down the coordinate sets, compare them—stays the same.

For broadcasting, draw three columns:

Output coordinates	Operand A's coordinates	Operand B's coordinates	B broadcasts over
`{b, c, h, w}`	`{b, c, h, w}`	`{c}`	`{b, h, w}`

The "broadcasts over" column is the output coordinates minus the operand's coordinates. The difference is the claim—the coordinates the operand declares independence from.

For permutation, draw two rows—input and output—and align the coordinates:

```
Input:  [b,  h,  w,  c]
          |   |   |   |
Output: [b,  c,  h,  w]
```

The arrows trace where each coordinate goes. `b` stays. `c` moves from position 4 to position 2. `h` and `w` shift. The ledger records the mapping, not the positions. If the input layout changes—if it becomes `[b, w, h, c]`—the arrows still find their targets, because the arrows connect names, not numbers.

The ledger applied to a complex expression:

```
let norm[i] = sum[j]( (A[i, j] - mean[j](A[i, j])) ** 2.0 ) ** 0.5;
```

Look at the brackets. The left-hand side names `i`—that is the survivor. `sum[j]` and `mean[j]` both name `j`—that is consumed. Now the subtraction `A[i, j] - mean[j](A[i, j])`: `mean[j](A[i, j])` produces `{i}` because `mean` consumed `j`. But `A[i, j]` has `{i, j}`. Set subtraction: `{i, j} - {i} = {j}`. The mean result broadcasts back over `j`—the coordinate it just consumed.

What just happened: the ledger reveals three things from the brackets alone: - Survivors: `{i}` (appears on the left-hand side) - Consumed: `{j}` (consumed by both `mean[j]` and `sum[j]`) - Broadcasts: `{j}` — the mean's result broadcasts back over the coordinate it consumed

The positional equivalent of this three-column ledger is `A.mean(dim=1, keepdim=True)`. The `keepdim=True` is the positional way of saying "broadcast back over the consumed coordinate." The ledger for the named version records *which* coordinate. The ledger for the positional version records *that* a coordinate was kept—but not which one.

Common Errors: What the Brackets Catch

Every new notation has characteristic mistakes. Here are the ones the megaphone model catches—and the ones it can't. Learn to recognize both.

Error 1: Reduction bracket disagrees with index

```
// ERROR: sum over j, but the index uses k
let wrong[i] = sum[j](matrix[i, k]);
```

The reduction bracket says `j`. The index pattern says `k`. `j` is not in the index list. The compiler reports: "reduction coordinate `j` does not appear in the tensor's index list. Did you mean `k`?"

The fix:

```
let right[i] = sum[j](matrix[i, j]);
```

This error is a typo—an easy one to make, an easy one to catch. In the positional version, this would be `matrix.sum(dim=1)` vs `matrix.sum(dim=0)`. Both compile. Only one is correct. The typo is invisible because there is no name to mismatch—there is only an integer, and all integers are valid.

Error 2: Implicit broadcast that should be explicit

```
// DANGER: temperature declared as scalar, but should vary by class
let scaled[batch, class] = logits[batch, class] / temperature;
```

`temperature` is a scalar. It broadcasts over both `batch` and `class`. The shapes match. The output is correct shape. But `temperature` was supposed to be `temperature[class]`—a per-class temperature scaling. The scalar version averages all classes with the same temperature, silently.

The compiler catches this if `temperature` is declared with the wrong coordinate set. If `temperature` is declared as `temperature[class]` but used as `temperature` (without brackets), the compiler reports: "scalar used where `[class]` was expected." If `temperature` is declared as a scalar and that's wrong, the compiler can't catch it—but the declaration is visible, and the reviewer can ask: "shouldn't temperature vary by class?"

Error 3: Broadcast over the wrong set of coordinates

```
// scale depends on c and h, but should also depend on w
let out[b, c, h, w] = x[b, c, h, w] * scale[c, h];
```

`scale[c, h]` broadcasts over `b` and `w`. The shapes work. But `scale` should vary with `w`—it's a per-width scaling factor. The omission of `w` from the bracket is a semantic error.

The compiler won't catch this. The broadcast is internally consistent. But the bracket makes the claim visible: `scale` is silent on `w`. The reader sees the claim and can challenge it. In `x * scale[:, None, :, None]`, the claim is encoded as shape manipulation. The reader has to decode the `None` positions, map them to the dimension order, and then ask the same question. The extra decoding step is where bugs hide.

Error 4: Keepdim forgotten

In a positional API, forgetting `keepdims=True` is one of the most common tensor bugs:

```
# Bug: mean returns (batch,) not (batch, 1), broadcast is wrong
x = x - x.mean(dim=1)  # missing keepdim=True
```

In Einlang, the broadcast requirement is derived from the coordinate structure, not from a flag:

```
let centered[b, c] = x[b, c] - mean[c](x[b, c]);
```

`mean[c](x[b, c])` produces a result with coordinates `{b}`. The subtraction expects `{b, c}`. The coordinate sets differ: `{b, c}` vs `{b}`. The missing coordinate is `c`. The compiler infers that the mean must broadcast back over `c`. `keepdims=True` is not a flag the programmer writes—it's a requirement the compiler derives from the coordinate structure.

The four errors form a gradient. Error 1 is caught unconditionally (name mismatch). Error 2 is caught if declarations are consistent (wrong coordinate set). Error 3 is not caught by the compiler, but the bracket makes the claim visible to the reader. Error 4 is eliminated entirely—the compiler derives the keepdims requirement from the coordinate sets, so the programmer cannot forget it.

The Four Operations

Here is what each primitive looks like in two notations, and what each notation records:

Operation	PyTorch/NumPy	Einlang	What the name records
Reduce	`x.mean(dim=1)`	`mean[channel](x[b, c, s])`	Which coordinate is consumed
Broadcast	`x + bias[:, None, :]`	`x[b, c, s] + bias[c]`	Which coordinates bias is silent on
Permute	`x.permute(0, 3, 1, 2)`	`y[b, c, h, w] = x[b, h, w, c]`	Where each coordinate ends up
Contract	`torch.matmul(A, B)`	`sum[k](A[b, k] * B[k, f])`	Which coordinate is shared and consumed

In every case, the name records a fact about identity. In the positional version, that fact is not recorded—it lives in the programmer's head. When a refactoring changes the dimension order, the PyTorch column requires updates to every affected line. The Einlang column does not. `channel` is still `channel`, regardless of its position. The positional column records *how*. The named column records *what*.

Three Questions Before You Continue

Every broadcast is a claim. Before you move on, ask these three questions of any broadcast you encounter:

1. **Which coordinate does this broadcast copy over?** Can you name it?
2. **Is it genuinely independent of that coordinate?** Does the value make sense without it?
3. **If the dimension order changed, would this broadcast still be correct?**

If you can't answer all three with confidence, the broadcast is an accident of shape alignment, not a defended claim. For now, the questions themselves are the habit.

Comprehensions: The Mirror of Reduction

You now have two operations on coordinate domains. Reduction consumes a coordinate—it walks along `i`, combines every value, and `i` is gone from the result. Broadcasting copies along a coordinate—the value is silent on it, so it gets replicated. Together they form the Inversion Rule: what broadcasts forward is summed backward.

But there is a third operation. And once you see it, you will wonder why it took so long to appear.

A **comprehension** traverses a coordinate without consuming it. It walks along `i`, applies an expression to each position, and produces a new array—one where `i` still exists, but with transformed values:

```
let squared = [data[i] * data[i] | i in 0..len(data)];
```

The bracket `[...]` on the right is a comprehension. It says: for every `i` in the given range, compute `data[i] * data[i]`, and collect the results into a new array. The coordinate `i` is traversed, not consumed. The result `squared` carries the same coordinate as `data`—same length, same identity. Nothing disappeared. Every position was visited and transformed.

Compare this to a reduction:

```
let total = sum[i](data[i]);            // i is consumed - total has no i
let squared = [data[i] * data[i]        // i survives - squared has i
              | i in 0..len(data)];
```

In the reduction, `sum[i]` points the megaphone at `i` and consumes it. In the comprehension, the megaphone is quiet—`i` is used but not eaten. The traversal leaves the coordinate intact.

Now add a condition:

```
let positives = [data[i] | i in 0..N, data[i] > 0];
```

This is a filtered comprehension. It traverses `i` and keeps only the positions where the condition holds. The result has the same coordinate name `i`, but its extent may be smaller. The coordinate identity survives; its domain shrinks.

Three operations on a single coordinate. When you face a coordinate `i` attached to a tensor, you have exactly three things you can do with it:

Operation	What it does to the coordinate	Notation
Reduce	Consume it — `i` is gone from the result	`sum[i](data[i])`
Broadcast	Copy along it — the value is silent on `i`	`bias[j]` in `out[i, j]`
Comprehend	Traverse it — `i` survives, values are transformed	`[f(data[i]) | i in 0..N]`

Reduction and broadcast are inverses—the Inversion Rule already showed you that. Comprehension is the missing sibling, the one that traverses without consuming. Together, the three form a complete language for describing what happens to coordinates in a tensor program.

Why does this matter? Because when you first learn tensor operations, you are taught two stories. Story one: "sum along an axis to get a smaller tensor." Story two: "broadcast a smaller tensor to match a larger one." These are presented as independent features—`sum` for reduction, shape alignment for broadcast. The coordinate that disappears in one and gets copied in the other is the same coordinate, but nothing in the notation connects them.

Then the Inversion Rule connects them, and you see they are a pair. But the pair is incomplete. Where is the operation that walks a coordinate without losing it? Where is the traversal?

It was always there. In Python:

```
squared = [data[i] * data[i] for i in range(len(data))]
```

That is a list comprehension. It traverses `i` and produces a new list with the same number of elements. In NumPy, vectorized operations do this implicitly—`data ** 2` visits every position without consuming the axis. The traversal is buried in the operator. The coordinate story is the same either way: `data` has a coordinate, the operation visits every position along it, and the result has the same coordinate. Nothing was consumed. Nothing was broadcast. The coordinate was *traversed*.

Einlang makes the traversal explicit:

```
let squared = [data[i] * data[i] | i in 0..len(data)];
```

The comprehension bracket says: "I am walking `i`. I am not consuming it. I am producing a new array with `i` intact." The notation records the traversal the same way `sum[i]` records the consumption and the omitted `[i]` on `bias[j]` records the broadcast.

Here is the symmetry:

```
                  consume
sum[i](data[i])       total          (i disappears)

                  traverse
[data[i]*2 | i]       doubled[i]     (i survives)

                  copy
bias[j]  +  out[i,j]                     (i absent from bias,
                                           copied into existence)
```

Three operations. Three ways the megaphone relates to a coordinate. Consume it. Copy along it. Traverse it. Every tensor program you will ever write is some combination of these three.

The boundary between traversal and reduction is thinner than it looks. Consider computing the L2 norm of every row in a matrix:

```
let norms = [sum[j](A[i, j] * A[i, j]) | i in 0..N];
```

The comprehension traverses `i`. For each `i`, the inner `sum[j]` reduces over `j`. The result is `norms[i]`—each position contains the squared norm of the corresponding row. Two operations, two coordinates. The comprehension handles `i`—traversal. The sum handles `j`—reduction. The brackets record which is which.

Now the same expression, rearranged:

```
let norms[i] = sum[j](A[i, j] * A[i, j]);
```

Einlang infers the traversal over `i` from the output declaration `norms[i]`. The comprehension is implicit—the output coordinate `i` tells the compiler: "traverse `i` to produce this array." The explicit form and the implicit form are equivalent. The traversal and the reduction are distinct operations on distinct coordinates. `i` is traversed. `j` is consumed. The brackets record the distinction.

You have been doing this all along. Every `let doubled[i, j] = matrix[i, j] * 2.0` is a traversal over `i` and `j`—two coordinates traversed, zero consumed. The rectangular declaration *is* a comprehension in the common case. The explicit comprehension bracket is for when you need to make the traversal visible—when you are filtering, when you are mixing traversal and reduction in the same expression, or when the traversal range is different from the coordinate's full domain.

Return to the Transformer

Look at this line again:

```
let attn_out[head, seq_q, d] = sum[seq_k](weights[head, seq_q, seq_k] * V[head, seq_k, d]);
```

Read each bracket. Find every sum. Find every omission.

The sum over `seq_k` — that is a reduction. `seq_k` appears on the right, in both `weights` and `V`, but it does not appear on the left. It is consumed. Gone after this line.

`weights` carries `head`, `seq_q`, `seq_k`. `V` carries `head`, `seq_k`, `d`. Neither carries `head` on the left of its own brackets — yet `head` appears in the output. That is a broadcast: `head` copies from the input to every cell of the output without being consumed.

`d` appears in `V` and in the output. It does not appear in `weights`. The multiplication broadcasts `weights` over `d`. The bracket records the omission.

`seq_q` appears in `weights` and in the output. It does not appear in `V`. `V` is silent on the query position. The same value tensor answers every query. That silence is a broadcast — and the design claim of attention.

`seq_q` survives. `head` and `d` survive. `seq_k` is consumed.

You couldn't read this line in Chapter 1. Now you can. Not because you memorized terminology — because the brackets speak.

The three questions apply to single operations. But real programs compose operations: softmax is a max, a subtract, an exp, a sum, and a divide—five steps, each involving coordinates with distinct roles. The question "does this broadcast make sense?" becomes "does this function's coordinate contract match its body?"

Part II · Combinations

Chapter 3 · Names as Contracts

> "The art of programming is the art of organizing complexity."
>
> — Edsger Dijkstra

Combinations · Coordinate-aware functions

A softmax is not one operation. It is a max reduction, a subtraction, an exponentiation, a sum reduction, and a division. Five steps, each involving coordinates with distinct roles. Five chances for a coordinate to go missing.

In a positional framework, each step's `dim=` argument must be correct independently. If the max is over `dim=-1` but the sum is over `dim=0`, the shapes might still align — and the bug ships. The individual operations are correct. The composition is wrong. Nothing in the positional notation records that the five `dim` arguments are supposed to refer to the same coordinate.

Part I taught us to name coordinates at individual operations. Part II asks: when operations compose, can the names survive the composition? The answer is yes — if the composition itself has a name, and that name is a contract.

The Softmax Decomposition

Softmax is the workhorse of classification. It takes a vector of logits and returns a probability distribution:

$$\text{softmax}(x)_j = \frac{e^{x_j}}{\sum_k e^{x_k}}$$

Decomposed into primitives:

```
let m = max[j](logits[j]);
let e[j] = exp(logits[j] - m);
let z = sum[k](e[k]);
let probs[j] = e[j] / z;
```

Every reduction states what it consumes. `max[j]` consumes `j`. `sum[k]` consumes `k`. The reader can see, at each line, which coordinate is being collapsed.

Now look at that first line again. `max[j]`—a built-in reduction that accepts a coordinate in brackets. You have been writing these since Chapter 1: `mean[channel](x)`, `sum[batch](x)`, `max[j](logits[j])`. Built-in reductions take a coordinate parameter. The bracket after the operation name holds the coordinate identity. The compiler checks that the tensor has that coordinate.

But the softmax above is four separate `let` statements. Write it once, fine. Write it in twelve places across a codebase, and the coordinate story scatters. The reader must reconstruct, at each call site, that `j` and `k` are the same underlying coordinate. The compiler cannot check it, because `j` and `k` are local to each statement.

Can a User Function Take a Coordinate?

The built-in reductions—`mean`, `sum`, `max`, `min`, `prod`—all accept coordinate parameters. You write `mean[channel](x)` and the compiler checks that `x` has a `channel` coordinate.

The question: can a user-defined function do the same thing?

Here is the standard positional approach:

```
def softmax(logits, dim=-1):
    m = logits.max(dim=dim, keepdim=True)
    e = (logits - m).exp()
    return e / e.sum(dim=dim, keepdim=True)
```

`dim=-1` says "the last one." If the last dimension is `class`, correct. If upstream changes the dimension order, `dim=-1` silently normalizes over whatever happens to be last. The code runs. The output is a valid probability distribution—over the wrong coordinate.

The positional function cannot name which coordinate it normalizes over. The built-in `mean[channel]` can. The gap is not about convenience. It is about first-class status: in Einlang, coordinate parameters are available to any function—built-in or user-defined. The bracket is not a privilege reserved for `mean` and `sum`. It is a language mechanism, and user functions get the same mechanism.

Derive it yourself. You want to write `softmax` so it accepts a coordinate parameter the way `mean[channel]` does. What must the signature declare? The input has unknown surrounding dimensions plus the normalization coordinate. The output must preserve all input coordinates — the normalization doesn't remove any. The body needs a max, an exp, a sum, and a division. Which operation consumes the coordinate? Which broadcasts it back? Write the signature before turning the page.

The Coordinate-Aware Function

Here is the same softmax, written so that it accepts a coordinate parameter the same way `mean[channel]` does:

```
fn softmax[j](x: [f32; ..left, j, ..right])
    -> [f32; ..left, j, ..right]
{
    let m[..left, ..right] = max[j](x[..left, j, ..right]);
    let e[..left, j, ..right] = exp(x[..left, j, ..right] - m[..left, ..right]);
    let z[..left, ..right] = sum[j](e[..left, j, ..right]);
    e[..left, j, ..right] / z[..left, ..right]
}
```

`fn softmax[j]`—the `j` in brackets after the function name is a coordinate parameter. The same bracket position that `mean[channel]` uses. This is not a new language feature. It is the existing coordinate-parameter mechanism, extended to user-defined functions. Coordinate parameters are first-class: any function, built-in or user-defined, can accept a coordinate in brackets.

`x: [f32; ..left, j, ..right]`—the parameter `x` is a tensor whose shape includes the coordinate `j`, plus zero or more leading coordinates (`..left`) and trailing coordinates (`..right`). These are packs—they stand for whatever coordinates surround `j` in the actual argument.

`-> [f32; ..left, j, ..right]`—the return type has the same coordinate structure as the input. The function preserves the normalized coordinate.

Now the call:

```
let logits[b, class] = model(x[b, feature]);
let p[b, class] = softmax[class](logits[b, class]);
```

The caller writes `softmax[class](...)`. The same bracket syntax as `mean[channel]`. The `class` in brackets is a coordinate argument—the name of the dimension the function normalizes over. That `logits` has a `class` coordinate is checked: the compiler subtracts `{class}` from `logits`'s coordinate set `{batch, class}` and finds a match. If `class` were absent, the set subtraction would produce an empty intersection, and the call would be a compile error. This is coordinate set subtraction, introduced in Chapter 2, applied to function calls.

Compare to the standard API:

```
p = torch.softmax(logits, dim=-1)
```

The same positional call. The same silent failure mode: if the last dimension changes, `dim=-1` follows the position, not the identity. The code runs. The output is a valid probability distribution—over the wrong coordinate.

Now compare to a different Einlang call:

```
let p[b, class] = softmax[b](logits[b, class]);
```

This normalizes over `b`—the batch dimension. It is a one-character bug (`b` instead of `class`). It is also a compile error, because `softmax[b]` would attempt to consume `b`, and the function signature says `b` should survive in `..left`. The coordinate contract catches the error at the call site.

Now pause and think about a different kind of failure. You write `softmax[class](logits)` and it works. Three months later, a colleague refactors the model. They rename the `class` coordinate to `category`—a better name, more consistent with the rest of the codebase. They update twenty-three files. They miss one: the call to `softmax[class]`. What happens?

The call is now `softmax[class](logits[b, category])`. The compiler checks: does `logits` have a `class` coordinate? No—it has `batch` and `category`. The error is not "shape mismatch." It is not "dimension 1 out of bounds." It is: **`logits` has no coordinate named `class`.** The error message names the missing coordinate. The fix is to change one character in the brackets: `softmax[category](...)`.

In the positional equivalent, `dim=1` would stay correct if only the name changed—but if the dimension order also changed, `dim=1` would silently begin normalizing over the wrong coordinate. No error. No warning.

Here is the distinction in one sentence: **when positions change without names changing, `dim=-1` silently becomes wrong. A named coordinate fails loudly. A positional API fails silently.**

Neither notation prevents all errors. But named coordinates make the errors *visible*. A compile error is visible. A silent semantic drift is not.

One Coordinate, Three Jobs

Inside `softmax[j]`, the coordinate `j` plays three distinct roles:

```
let m[..left, ..right] = max[q](x[..left, q, ..right]);    // Role 1: stability scan
let e[..left, k, ..right] = exp(x[..left, k, ..right] - m); // Role 2: exponentiate
let z[..left, ..right] = sum[j](e[..left, j, ..right]);    // Role 3: normalize
```

All three—`q`, `k`, `j`—range over the same domain (the class axis). But each carries a different **gradient contract**. The stability scan passes a sparse gradient: only the maximum element receives a signal. The denominator scan passes a dense gradient: every element contributes to the sum. The output has a diagonal-plus-off-diagonal structure: each element's gradient depends on itself and on every other element. The coordinate `j` is consumed by `max`, consumed again by `sum`, and reconstructed by the division—two different consumption events on the same coordinate, with the coordinate surviving both.

In a `dim=-1` API, these three roles collapse into a single integer. The reader cannot see which role `dim=-1` plays at each step. In the named-coordinate version, the roles are given distinct letters, and a reader can audit whether the gradient contracts are satisfied.

Now ask yourself: why use three different letters (`q`, `k`, `j`) for the same coordinate? Why not just `j` everywhere? Because the *binding site* of each occurrence carries different gradient implications. `max[q]` says: "I consume `q` and return a scalar per batch element. The backward pass through me will broadcast the gradient signal to only the maximum element." `sum[j]` says: "I consume `j` and return a scalar per batch element. The backward pass through me will broadcast the gradient signal to *all* elements." Same coordinate domain. Different gradient contracts. Different letters make each contract's scope visible: `q` is consumed by `max` and never seen again. `k` is used in the exponent and survives. `j` is consumed by `sum` and reconstructed by the division.

The letters are not decoration. They are the scope markers for the coordinate's three lives. In `dim=-1`, all three are the same integer. The scopes are invisible.

Function Composition

Coordinate-aware functions compose. The output of one becomes the input of another, and the coordinate contracts chain.

Consider a pipeline: linear layer, then softmax:

```
fn linear[in, out](x: [f32; ..b, in], W: [f32; out, in], b: [f32; out])
    -> [f32; ..b, out]
{
    sum[in](x[..b, in] * W[out, in]) + b[out]
}

fn pipeline[in, class](x: [f32; ..b, in], W: [f32; class, in], b: [f32; class])
    -> [f32; ..b, class]
{
    let logits[..b, class] = linear[in, class](x[..b, in], W[class, in], b[class]);
    softmax[class](logits[..b, class])
}
```

The coordinate `class` flows from the pipeline's signature through `linear[in, class]` into the result `logits`, then into `softmax[class]`. At each step, the compiler checks: does the argument carry the coordinate the function expects? `linear` expects `in` and `out`—the caller binds `class` to `out`. `logits` now carries `class`. `softmax[class]` expects `class` on its argument—`logits` has it. The chain is verified.

If a refactoring changes `linear`'s output coordinate from `class` to `category`, the pipeline still compiles—`linear[in, category]` produces a tensor with `category`, and `softmax[class]` complains that `category`

is not `class`. The error is at the composition boundary. The compiler names both coordinates. The mismatch is visible.

Positional composition has no such check. The chain is unverified.

The Contract in One Question

Every coordinate-aware function can be audited with a single question: **does the caller's argument carry the coordinate that the function claims to operate on?**

If yes: the call compiles, and the coordinate flow is guaranteed consistent. If no: the call is rejected, and the error message names the missing coordinate.

This is the difference between a type-level contract and a documentation-level contract. `def softmax(logits, dim=-1)` has a documentation-level contract: the docstring says `dim` is the class dimension. But nothing checks it. `fn softmax[j](x)` has a type-level contract: `j` is a coordinate parameter, and the compiler verifies that the argument carries it. The contract is not a hope. It is a check.

The rest of this book builds on this distinction. Every chapter from here forward—skeletons, recurrences, gradients, comparisons—assumes that coordinate identities can be checked. The coordinate-aware function is the mechanism that makes checking possible.

Domain and Extent

A coordinate has three properties. You've seen the first two: a **name** (`batch`, `class`) and a **domain** (the set of values it ranges over). The third is **extent** — the size of the domain, the number you see in a shape tuple.

Extent and domain are different things. `class` and `expert` may both have extent 1024. They are not the same coordinate. A tensor of shape `(1024, 1024)` could be `[class, expert]`, `[expert, class]`, `[batch, class]`, or `[seq, hidden]`. The extents are identical. The domains are different.

Positional notation records only the extent: `(1024, 1024)` tells you the sizes. It does not tell you which is which. Named notation records both: `[class: 1024, expert: 1024]`. The distinction is not academic — it is the root cause of every Square Matrix Test failure in this book. When two extents are equal, shape checkers become domain-blind. Names restore sight.

Keep this distinction in mind through the next section. The Square Matrix Test probes exactly this gap.

The Square Matrix Test

There is a simple, brutal test for whether a piece of tensor code is robust to coordinate swaps. Set all dimension sizes equal. Swap two axes. Ask: does the program still mean the same thing?

For a square input where `batch_size == num_classes == 128`:

```
let probs[batch, class] = softmax[class](logits[batch, class]);   // correct
let probs[class, batch] = softmax[batch](logits[class, batch]);   // bug
```

Both lines produce a `(128, 128)` matrix where every row sums to 1. The cross-entropy loss descends identically. The training curves overlay perfectly. But the first normalizes classes against each other. The second normalizes examples against each other.

When `batch_size == num_classes`, the probability matrix is square. Softmax over rows and softmax over columns produce the same numbers when the matrix has symmetric structure. The loss curves overlay. The calibration reports pass. Six weeks later, a deployed model silently normalizes examples against each other instead of classes against each other.

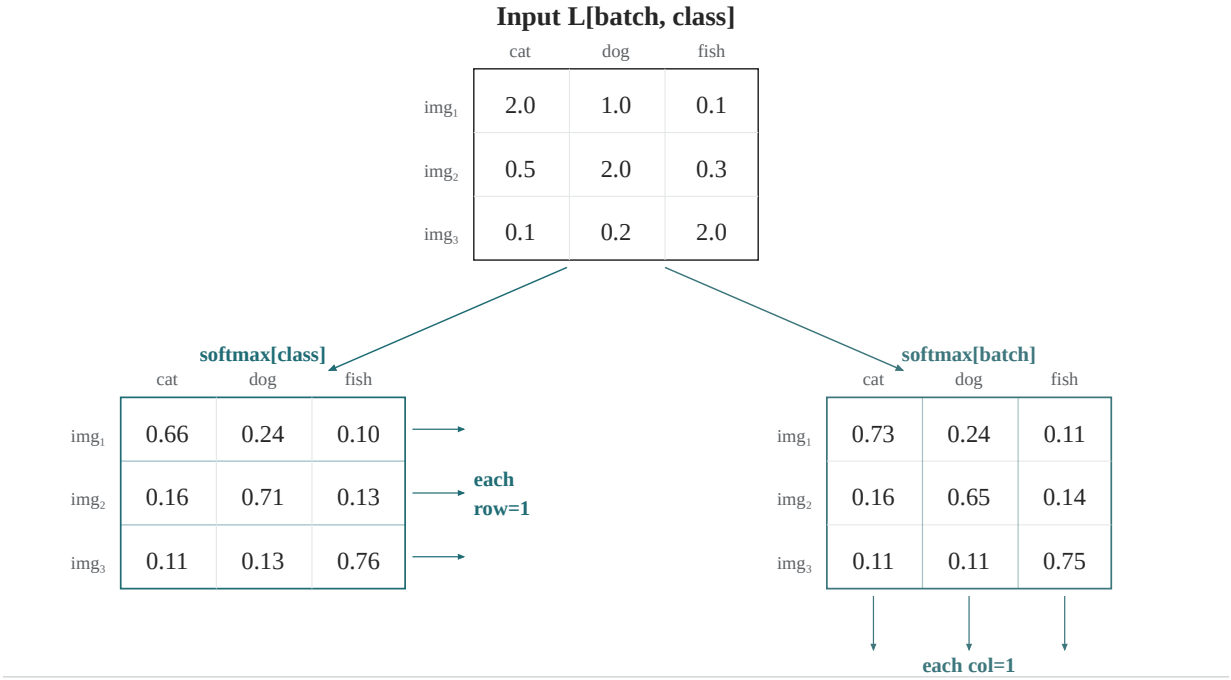


Figure 5: Same input, same shape, softmax over two different coordinates.

Horizontal dividers: each row sums to 1. Vertical dividers: each column sums to 1. Same input, same output shape, different numbers. The only difference in the source code is the name inside the bracket. `class` versus `batch`. One word. `dim=-1` does not contain that word. It cannot. The word is in your head—exactly where the shape-meanings gap puts it.

No shape checker catches this. No gradient check catches this. Only a notation that records *which coordinate is the distribution* catches this.

The Square Matrix Test is named after this property: when all extents are equal, a coordinate swap can hide inside shape compatibility. If square matrices fool shape checkers—and they do, routinely—what can prevent this class of error?

The Square Matrix Test is not specific to softmax. It applies to any operation where two coordinates can have equal extents. Consider three more cases:

Matrix multiplication. `C[i, j] = sum[k](A[i, k] * B[k, j])` and `C[i, j] = sum[k](A[k, i] * B[k, j])` produce the same shape when `A` is square. The first uses `A`'s rows. The second uses `A`'s columns (equivalent to `A^T @ B`). When `A` is a square matrix of size `128 × 128`, both expressions produce

a `128 × 128` output. Shape checkers see the same shape. Only the coordinate names distinguish the two computations.

Broadcast. `let out[i, j] = A[i, j] + bias[j]` and `let out[i, j] = A[i, j] + bias[i]` both produce shape `(128, 128)` when `i` and `j` have the same extent. The first broadcasts `bias` over rows—bias depends on columns. The second broadcasts `bias` over columns—bias depends on rows. Semantically opposite. Shape-identical. Distinguishable only by which coordinate appears in `bias`'s bracket.

Attention. Self-attention and cross-attention use the same `matmul(Q, K.transpose(-2, -1))` operation. When `seq_q == seq_k`, the code is textually identical. The distinction between attending to yourself and attending to a different sequence lives in the tensor shapes at runtime—not in the source code. When the sequence lengths differ, the shapes diverge and the bug surfaces. During development, when they happen to be equal, the bug is invisible.

In every case, the Square Matrix Test reveals the same gap: shape compatibility checks the arithmetic of dimensions. It does not check the identity of dimensions. When two dimensions have the same size, shape compatibility becomes identity-blind. Named coordinates restore sight.

Take a breath. Four cases. Softmax, matmul, broadcast, attention. All of them can fail silently when two extents match. The Square Matrix Test is not a trick — it is the normal state of deep learning models, where embedding dimensions, hidden sizes, and projection dimensions are routinely set to the same value. If your `d_model` is 512 and your `d_ff` is also 512 — which happens in the original Transformer — positional notation cannot distinguish them. Your code works. Your shapes match. Your coordinate identities are invisible to every tool you have.

The test is not hypothetical. It is a property of your current codebase. Every pair of equal-sized dimensions is a potential Square Matrix Test that your tooling is failing right now. The only question is whether a bug has found one yet.

The Refactoring: A Detailed Demonstration

Derive it yourself. You have three files. `data.ein` declares `logits[batch, class]`. `model.ein` calls `softmax[class](logits)`. `loss.rs` calls `cross_entropy[class](probs, labels)`. A colleague renames `class` to `category` in `data.ein`. Which files break? What do the error messages say? Write the two errors on paper. Then read on.

A refactoring in both notations, to see where the errors surface—and where they don't.

The colleague makes the rename. Two errors appear:

```
error[E0425]: model.ein:42: tensor `logits` has no coordinate named `class`
  --> model.ein:42:20
   |
42 |     softmax[class](logits);
   |                    ^^^^^^ `logits` has coordinates: batch, category
   |     help: did you mean `category`?

error[E0425]: loss.rs:15: tensor `probs` has no coordinate named `class`
  --> loss.rs:15:25
```

```
   |
15 |     cross_entropy[class](probs, labels);
   |                          ^^^^^ `probs` has coordinates: batch, category
   |     help: did you mean `category`?
```

The colleague fixes both: `class` → `category`. The project compiles. Done. The compiler verified every use of the old name was updated.

Now replay with a positional API. The colleague changes the comment in `data.py`: the shape is now `(batch, category)`. `dim=1` is still `1`. Every `dim=1` is still correct—because the position didn't change. The refactoring compiles silently.

But if the dimension order also changed—say `(category, batch)`—some `dim=1`s should become `dim=0`s. There are twenty-three of them across eight files. The colleague updates the ones they remember. The ones they forget compile silently. `dim=1` is always a valid integer. The compiler cannot distinguish the ones that should have changed from the ones that shouldn't.

The Einlang refactoring emits two errors—one per call site. The positional refactoring emits zero—even when the dimension order changes. Zero errors is not zero bugs. It is zero *detected* bugs. The coordinate name is the audit trail. The positional integer has none.

The Language Gets a Name

The preceding chapters have been written in a notation that puts coordinate names in brackets, that requires reductions to state what they consume, that makes broadcasting explicit in the indexing pattern. This notation needs a name.

It is called **Einlang**—a contraction of "Einstein" and "language," acknowledging the debt to Einstein summation notation while distinguishing itself as a full programming language rather than a string-based convention.

It is called **Einlang**—a contraction of "Einstein" and "language," acknowledging the debt to Einstein summation notation while distinguishing itself as a full programming language rather than a string-based convention. A language where coordinates are first-class syntactic entities, not comments embedded in variable names. Where coordinate contracts are statically checked. Where the reader can audit coordinate flow without reconstructing it from shape arithmetic.

A coordinate-aware function does something that no positional API can do: it makes the **identity** of the operated-on coordinate part of the function's type-level contract. The caller must name the coordinate. The name is checked against the argument's layout. The function body uses the name without knowing its position.

This is the combination layer. The primitives—naming, permuting, reducing, broadcasting—are composed into a function whose coordinate behavior is specified in its signature. The function can be called, passed around, and composed further, without losing the coordinate information that the primitives established.

But there is a subtler consequence. When coordinates are part of the type-level contract, refactoring becomes checkable. Rename a coordinate from `class` to `category`, and every call site that passes `class` becomes a compile error. The error message names the call site and the missing coordinate. The

refactoring is systematic: change all `class` to `category`, and the errors disappear one by one. No silent breakage. No "hope I found all the places." The compiler is the audit trail.

In a positional API, renaming a dimension's role (e.g., changing what "position 1" means) produces no errors. The code compiles. The integer didn't change. Only the meaning changed. The compiler can't track meaning. It tracks integers. The refactoring is silent—and its bugs are silent with it.

The coordinate identity, made part of the function's contract, becomes auditable by the compiler rather than by the programmer alone.

A coordinate-aware function makes the coordinate part of the type-level contract. But the contract is not just a promise between programmer and compiler. It is a promise between the programmer who wrote the function and the programmer who calls it six months later. The bracket `softmax[class]` tells the future reader: "this function normalizes over `class`." The bracket `softmax(logits, dim=-1)` tells the future reader: "this function normalizes over the last dimension—whatever that is." One is a recorded decision. The other is a puzzle.

What does it take to make this contract real? Two pieces: the syntax that separates coordinates from values, and the mechanical steps the compiler uses to check them. Both are worth seeing once—not because you'll do them by hand, but because knowing what the compiler checks tells you what the contract guarantees.

What Changes

If you look at a `dim=` argument in your own code right now, you can probably name the coordinate it refers to. `dim=1` means `channel`—you know that. The compiler doesn't. The name is in your head, not in the code.

Three things change when the name moves from your head into the bracket. First, the compiler can check it: if `dim=1` was supposed to be `class` but the data loader put `spatial` at position 1, the bracket `mean[class]` catches the mismatch. The positional `dim=1` catches nothing. Second, refactoring becomes safe: if `channel` moves from position 1 to position 2, `mean[channel]` follows the name. `dim=1` follows the position—silently. Third, a reader six months later sees `mean[channel]` and knows what was intended. They see `dim=1` and have to reconstruct the intent from context.

The coordinate habit is noticing that gap. The rest of this book is about what happens when you fill it.

Under the Hood

Two pieces make the contract checkable. First, the syntax separates coordinate parameters from value parameters. Second, the compiler follows a six-step mechanical procedure for every call. Neither is complex. Both are worth seeing once.

Brackets and Parentheses

You have been writing two kinds of things without naming the distinction. `softmax[class](logits)` — the `class` in brackets names a coordinate. The `logits` in parentheses holds data. They are different syntactic positions because they are different kinds of arguments. The bracket position says: "this is

a coordinate identity—check it against the tensor's layout." The parenthesis position says: "this is a value—type-check it normally."

In `softmax(logits, dim=-1)`, `dim=-1` is syntactically identical to any other integer argument. If you pass `dim=42` by accident, the syntax has no opinion. It's just an integer. The bracket creates a syntactic position that the compiler recognizes as "coordinate argument." Everything in that position gets checked. The syntax carves out a space for coordinate verification—and the compiler fills it.

The Six Steps

Here is the function signature:

```
fn softmax[j](x: [f32; ..left, j, ..right]) -> [f32; ..left, j, ..right]
```

And the call:

```
let probs[b, c] = softmax[c](logits[b, c]);
```

The six steps the compiler performs, mechanically, for every call:

Step	Answer
1. Coordinate parameter	`j`
2. Coordinate argument	`c`, bound to `j`
3. Does `logits` carry `c`?	Yes—`logits[b, c]`
4. Pack bindings	`..left` = `[b]`, `..right` = `[]`
5. Return type	`[f32; b, c]`
6. Valid?	Yes

No intuition, no shape arithmetic, no guessing. If any step fails, the compiler emits an error naming the missing coordinate.

Now a call that should fail:

```
let probs[b, f] = softmax[c](logits[b, f]);
```

Step 3 catches it: `logits` carries `b` and `f`. `c` is not in `{b, f}`. The compiler reports: "`logits` has no coordinate named `c`. Available coordinates: `b`, `f`."

Five wrong calls, and where each one breaks:

Call	What goes wrong	Caught by
`softmax[class](logits[batch, feature])`	`logits` has no `class`	Step 3: index existence
`softmax[batch](logits[batch, class])`	`batch` would be consumed and returned—contract violation	Step 5: return type
`softmax[class](logits)` where `logits` is 1D	`class` exists, packs are empty—valid	No error (correct)
`softmax[class](logits[batch, class, extra])`	`..right` binds to `[extra]`—valid	No error (correct)
`softmax[class](logits[batch, class], wrong_arg)`	Wrong number of value arguments	Step 1: arity check

Each check is a mechanical verification. You won't perform these steps by hand. But knowing they exist changes how you read a function call. `softmax[class](logits)` is not a request. It is a contract submission. The compiler either stamps it or rejects it. The stamp means the coordinate story is consistent.

Where Coordinates Come From

Step 3 asks "does `logits` carry `c`?" But the compiler asks an even more fundamental question first: does `c` exist at all? A coordinate that appears from nowhere — referenced but never declared — is not a type error. It is a grounding error. The compiler calls it E0701.

You have now seen every way a coordinate can enter scope. There are exactly four:

1. **Declaration.** `let x[i, j]` — the output coordinates `i` and `j` are grounded by the `let`.
2. **Reduction.** `sum[k](...)` — the bracket introduces `k` as a bound variable inside the reduction body.
3. **Parameter type.** `x: [f32; ..left, j, ..right]` — the parameter's shape grounds `..left`, `j`, and `..right` in the function body.
4. **Coordinate parameter.** `fn softmax[j]` — the bracket after the function name grounds `j` as the coordinate the function operates on.

If a coordinate can't be traced to one of these four, the compiler stops. Not "maybe it's a free variable." Not "let's infer it from context." Error E0701, compile time, with the coordinate's name in the message.

The six steps check that coordinates are used *consistently*. The grounding check verifies that they *exist*. Together they guarantee that every coordinate you read has been declared somewhere — and that every declaration is checked against every use.

Coordinate-aware functions compose. But before seeing the pattern they form together, there is a more immediate question: when you write a broadcast, you are making a claim about coordinate independence. What is the claim? And who checks it?

Chapter 4 · The Broadcast Self-Audit

"Silence is not absence. Silence is a claim. And claims can be checked."

Combinations · The inversion rule: what broadcasts forward collects backward

Every broadcast you write is a claim you didn't know you were making.

The claim is: *this value does not depend on the coordinate I am omitting.* When you write `out[i, j] = A[i, j] + bias[j]`, the omission of `i` from `bias[j]` claims the bias is the same for every `i`. When you write `scaled[batch, class] = logits[batch, class] / temperature[class]`, the omission of `batch` from `temperature[class]` claims the temperature is the same for every batch element.

Most of the time, the claim is true. The bias genuinely doesn't depend on the batch element. The temperature genuinely doesn't depend on the class. But when the claim is false, the code still runs. The shapes still match. The loss still descends — just to a higher plateau. And you spend an afternoon wondering why your adaptive class weights aren't adapting.

The broadcast self-audit is three questions you ask before the broadcast becomes a bug. Thirty seconds. It catches the afternoon.

Now read two lines:

```
// Forward
let out[i, j] = A[i, j] + bias[j];

// Backward: gradient of bias
let d_bias[j] = sum[i](d_out[i, j]);
```

Forward: `bias[j]` omits `i` in its index pattern. The coordinate `i` is absent, so `bias` is copied along `i`. Broadcast.

Backward: `d_bias[j]` is the gradient with respect to `bias`. `d_out[i, j]` carries gradient signals from every `(i, j)` position — `bias` contributed to all of them equally. To update `bias`, collect all those signals: sum over `i`. Reduction.

The coordinate that was broadcast forward (`i`) is the coordinate reduced backward. This is the Inversion Rule. Every forward operation has a backward dual. Broadcast becomes reduction. Reduction becomes broadcast.

```
// Forward: reduction consumes j
let row_sum[i] = sum[j](matrix[i, j]);

// Backward: broadcast j back
let d_matrix[i, j] = d_row_sum[i];
```

Forward: `sum[j]` consumes `j`. Every `j` position collapses into a single sum. Backward: `d_row_sum[i]` broadcasts along `j` — every `j` position receives the same gradient signal. The consumed coordinate is reborn as a broadcast.

Two lines. Two directions. One rule. The Inversion Rule is not mathematics bolted onto the coordinate system. It is the coordinate system, read in reverse.

Derive it yourself. Given `let scaled[batch, class] = logits[batch, class] / temperature[class]`,

write the backward gradient for `temperature`. Which coordinate does the sum go over? Why? Write it down before reading further.

The Self-Audit: Three Questions

Now apply this to your own code. Every broadcast you write is a claim. The claim is: *this value does not depend on the coordinate I am omitting.* If the claim is false, the forward pass is wrong. If the claim is true but the backward pass doesn't reduce over the omitted coordinate, the gradient is wrong.

Three questions for every broadcast:

Question 1: What coordinate am I broadcasting over? Is the name visible in the code, or is it inferred from position?

In `out[i, j] = A[i, j] + bias[j]`, the omitted coordinate is `i`. The code says so: `A` has `(i, j)`, `bias` has `(j)`. The difference is `{i}`. The broadcast is visible in the index patterns.

In `out = A + bias`, the broadcast is invisible—the shapes determine what happens. The code doesn't say which coordinate is being broadcast over.

Question 2: Is independence genuinely justified? Does the broadcast value genuinely not depend on that coordinate?

A bias term in a linear layer should not depend on the batch index. Each sample gets the same bias. The broadcast over `batch` is semantically justified.

A temperature scaling factor in a softmax should not depend on the class index. The broadcast over `class` is semantically justified.

But what about a mask that you broadcast over the sequence length? If the mask depends on the sequence position—if later positions are masked differently than earlier ones—then broadcasting a single mask value over all positions is semantically wrong. The shapes would work. The code would run. But the mask would not encode the position-dependent pattern you intended.

The broadcast self-audit asks: *is this broadcast a computational convenience, or a semantic claim?* If it is a semantic claim, is the claim true?

Question 3: What will the gradient do? Does the backward reduction produce the right shape for the parameter update?

In `d_bias[j] = sum[i](d_out[i, j])`, the sum over `i` produces a gradient of shape `(j)`—exactly `bias`'s shape. The parameter update `bias -= lr * d_bias` is well-shaped.

But what if you wrote the broadcast differently? What if `bias` had shape `(1, j)` and broadcasting expanded it to `(i, j)`? The gradient would still be `sum[i](d_out[i, j])`, producing `(j,)`. If your optimizer expects `(1, j)`, you need a reshape. The reshape is a positional hack to make the shapes align. The named version produces the correct shape by construction—the gradient has the same coordinates as the parameter.

The Auditor's Toolkit

A systematic procedure. Given any expression that contains a broadcast, you can audit it with these steps:

1. **List the coordinate sets.** Write down the coordinates of every tensor in the expression.
2. **Compute the broadcast sets.** For each term, subtract its coordinate set from the output coordinate set. The difference is what that term broadcasts over.
3. **Check justification.** For each broadcast, ask: is it semantically correct for this term to be independent of these coordinates?
4. **Predict the gradient.** For each broadcast, write the backward reduction: sum over the broadcast set. Verify that the result has the same coordinates as the parameter.

Let's apply this to a realistic example. Here is a layer normalization with a learnable scale and shift:

```
fn layer_norm[feature](x: [f32; ..b, feature],
                       gamma: [f32; feature],
                       beta: [f32; feature])
    -> [f32; ..b, feature]
{
    let mean[..b] = mean[feature](x[..b, feature]);
    let centered[..b, feature] = x[..b, feature] - mean[..b];
    let var[..b] = mean[feature](centered[..b, feature] ** 2.0);
    (centered[..b, feature] / (var[..b] ** 0.5 + 1e-5)) * gamma[feature] + beta[feature]
}
```

Step through the auditor's toolkit.

Step 1: Coordinate sets.

- `x`: `{..b, feature}`
- `mean`: `{..b}` — `feature` was consumed by `mean[feature]`
- `centered`: `{..b, feature}`
- `var`: `{..b}` — `feature` was consumed by `mean[feature]`
- `gamma`: `{feature}`
- `beta`: `{feature}`

Step 2: Broadcast sets.

The final expression is `centered / (var ** 0.5 + eps) * gamma + beta`. The output coordinates are `{..b, feature}`.

- `centered`: has `{..b, feature}`. Broadcast set = `{}`. No broadcast.
- `var`: has `{..b}`. Broadcast set = `{feature}`. `var` broadcasts over `feature`.
- `gamma`: has `{feature}`. Broadcast set = `{..b}`. `gamma` broadcasts over `..b`.
- `beta`: has `{feature}`. Broadcast set = `{..b}`. `beta` broadcasts over `..b`.

Step 3: Justification.

- `var` broadcasts over `feature`: justified. The variance is computed per-batch-element, then applied to all features. This is the definition of layer normalization.
- `gamma` broadcasts over `..b`: justified. `gamma` is a per-feature parameter. Every batch element gets the same scale.
- `beta` broadcasts over `..b`: justified. Same reasoning as `gamma`.

Step 4: Gradient prediction.

- `d_var[..b] = sum[feature](d_out[..b, feature] * ...)`. The gradient sums over `feature`—the broadcast set. Result: `{..b}`, matching `var`.

- `d_gamma[feature] = sum[..b](d_out[..b, feature] * ...)`. The gradient sums over `..b`—the broadcast set. Result: `{feature}`, matching `gamma`.
- `d_beta[feature] = sum[..b](d_out[..b, feature])`. Same. Result: `{feature}`, matching `beta`.

Every gradient has the same coordinates as its parameter. The broadcast sets from Step 2 become the reduction sets in Step 4. The Inversion Rule, applied mechanically.

When the Audit Fails

A programmer writes a temperature-scaled softmax. The intent is per-class temperatures:

```
let temperature[class] = get_per_class_temperature();
let scaled[batch, class] = logits[batch, class] / temperature[class];
```

`temperature` broadcasts over `batch` but not `class`. The auditor asks: is `temperature` independent of `batch`? Yes. Independent of `class`? No—and the index pattern correctly omits `batch` but includes `class`.

Now suppose the programmer accidentally wrote:

```
let temperature = get_per_class_temperature();  // returns scalar by mistake
let scaled[batch, class] = logits[batch, class] / temperature;
```

`temperature` is a scalar—broadcasts over everything. The shapes work. The loss descends but plateaus higher. The auditor's Question 2 catches it: the broadcast claims `temperature` is independent of `class`. The claim is false.

A second example. Adaptive class weights for a weighted loss:

```
let class_weights = compute_adaptive_weights(losses);  // BUG: returns scalar by accident
let weighted[batch] = mean[class](losses[batch, class] * class_weights);
```

`compute_adaptive_weights` was supposed to return `[f32; class]` but returns a scalar. The scalar broadcasts over `class`—every class gets the same weight. The adaptive weighting is silently disabled. The auditor asks: is `class_weights` independent of `class`? It shouldn't be.

In a positional framework, both bugs survive because `(batch, class) / scalar` and `(batch, class) * scalar` are perfectly valid. The broadcast is silent. The audit makes it speak.

The Inversion Rule in One Diagram

```
Forward                          Backward
-------                          --------

Reduction consumes {j}            →    Broadcast {j} back
    sum[j](A[i, j])                          d_sum[i] → d_A[i, j]

Broadcast omits {i}               →    Reduction collects over {i}
    A[i, j] + bias[j]                      d_bias[j] = sum[i](d_out[i, j])

Permute rearranges {i, j}         →    Permute rearranges back
```

```
    y[j, i] = A[i, j]                    d_A[i, j] = d_y[j, i]

Elementwise preserves              →    Elementwise preserves
    y[i, j] = f(x[i, j])                 d_x[i, j] = f'(x[i, j]) * d_y[i, j]
```

Every forward operation has a backward dual. The dual is not a separate rule. It is the forward rule, read backward, with the coordinate names as the thread connecting the two directions.

Reduction → Broadcast. Broadcast → Reduction. Permute → Permute. Elementwise → Elementwise.

Think of the forward pass as shopping: you walk through the aisles, items enter your cart, some are consumed (reduction), some are copied (broadcast). The backward pass is restocking: the manager reads the record backward, replenishing what was consumed and collecting what was copied.

The coordinate names are on both sides of the receipt. The Inversion Rule is the guarantee that the two sides match.

The Audit as a Habit

The broadcast self-audit is not a tool. It is a habit. You don't run it. You ask it.

Before you merge a pull request that contains a broadcast, ask the three questions. Before you write a custom backward pass, trace the Inversion Rule for every broadcast in the forward pass. Before you debug a gradient that's the wrong shape, check whether the broadcast set and the reduction set match.

The questions cost seconds. The bugs they catch cost hours. The ratio—as the epilogue will remind you—is favorable.

But there is a deeper reason to practice the audit. Every time you ask "what coordinate am I broadcasting over?" you are doing something that positional notation makes difficult and named notation makes easy: you are connecting the operation to its intent. The broadcast is not just a shape compatibility check. It is a semantic claim. The audit makes the claim explicit.

The Consumption Self-Audit

Broadcast and consumption are duals. The broadcast self-audit asks: *what coordinate am I silent on?* The consumption self-audit asks: *what coordinate am I erasing?* Both deserve their own diagnostic tool.

Just as a broadcast is a claim of independence, a reduction is a claim of dispensability. `sum[class](x[batch, class])` claims: *the coordinate `class` can be collapsed without losing information that other coordinates depend on.* If `class` carries structure that downstream operations rely on, the reduction is semantically wrong—even if the shapes match.

Three questions for every reduction:

Question 1: What coordinate am I consuming? Is the name visible in the code?

In `let row_sums[i] = sum[j](matrix[i, j])`, the consumed coordinate is `j`. The reduction bracket says so. In `x.mean(dim=1)`, the consumed coordinate is "whatever is at position 1." The name is absent.

Question 2: Does this coordinate appear in every operand of the reduction body?

In `sum[k](A[i, k] * B[k, j])`, `k` appears in both `A` and `B`. The reduction is well-formed. In `sum[class](x[batch, channel] + bias[channel])`, `class` appears nowhere—the compiler reports "reduction coordinate `class` not found." The check is mechanical.

Question 3: What will the backward pass do? The consumed coordinate becomes a broadcast in the gradient.

Forward: `let row_sums[i] = sum[j](matrix[i, j])`. Consumed: `j`. Backward: `d_matrix[i, j] = d_row_sums[i]`. The forward reduction over `j` becomes a backward broadcast over `j`. The Inversion Rule, applied to consumption.

Now put the two audits side by side:

```
BROADCAST SELF-AUDIT                    CONSUMPTION SELF-AUDIT
--------------------                    ----------------------
Q1: What coordinate am I silent on?     Q1: What coordinate am I consuming?
Q2: Is independence genuinely true?     Q2: Is it in every operand?
Q3: What will the gradient collect?     Q3: What will the gradient broadcast back?

Forward: omit coordinate → broadcast   Forward: consume coordinate → reduce
Backward: sum over omitted coordinate   Backward: broadcast the consumed coordinate
```

The two audits are the same audit, read in opposite directions. A broadcast claims independence. A reduction claims dispensability. Both claims are recorded in the brackets. Both claims are checkable. Both claims have backward consequences that the Inversion Rule predicts.

The broadcast audit catches the bug where a value is copied over a coordinate it should depend on. The consumption audit catches the bug where a coordinate is erased that downstream operations need. Together, they cover the two ways a coordinate's identity can be lost: by being ignored, or by being destroyed.

The Double Audit: When Broadcasts Compose

Most real code has more than one broadcast. A linear layer with bias has one. A layer normalization has four. When broadcasts compose, their backward reductions compose too. The auditor's toolkit handles them mechanically—one broadcast at a time. But the interactions are worth tracing once, so you develop an instinct for finding the hidden ones.

Here is a complete attention projection block:

```
let context[..b, head, seq_q, d] =
    sum[seq_k](weights[..b, head, seq_q, seq_k] * V[..b, head, seq_k, d]);
let output[..b, seq_q, head, d_out] =
    sum[d](context[..b, head, seq_q, d] * W_o[head, d, d_out]);
let final[..b, seq_q, d_out] = output[..b, seq_q, d_out] + b_o[d_out];
```

Three lines. Let the auditor walk through each.

Line 1: `sum[seq_k](weights * V)`

```
Output coordinates: {..b, head, seq_q, d}
weights: {..b, head, seq_q, seq_k}  → broadcast: {} (no omission, seq_k is reduced)
V:       {..b, head, seq_k, d}      → broadcast: {seq_q} (V omits seq_q)
```

`V` broadcasts over `seq_q` inside the reduction. This is correct: `V` provides values at each `seq_k` position regardless of which `seq_q` is querying. The backward reduction: `dV[..b, head, seq_k, d] = sum[seq_q](d_context[..b, head, seq_q, d] * weights[..b, head, seq_q, seq_k])`. The broadcast set `{seq_q}` becomes the reduction set.

Line 2: `sum[d](context * W_o)`

```
Output coordinates: {..b, seq_q, head, d_out}
context: {..b, head, seq_q, d}  → broadcast: {}
W_o:     {head, d, d_out}       → broadcast: {..b, seq_q}
```

`W_o` broadcasts over `..b` and `seq_q`. Correct: the weight is the same for all batch elements and query positions. Backward: `dW_o[head, d, d_out] = sum[..b, seq_q](d_output[..b, seq_q, head, d_out] * context[..b, head, seq_q, d])`.

Line 3: `output + b_o`

```
Output coordinates: {..b, seq_q, d_out}
output: {..b, seq_q, d_out}  → broadcast: {}
b_o:    {d_out}              → broadcast: {..b, seq_q}
```

Backward: `db_o[d_out] = sum[..b, seq_q](d_final[..b, seq_q, d_out])`.

Putting it together. Three expressions. Five broadcasts—two of them hidden inside reductions. Every broadcast has a backward reduction over the same coordinate set. The Inversion Rule holds for all of them.

Now ask yourself: in the PyTorch version of this block, how many would you notice? Bias over batch is obvious. Weight over batch and sequence is visible but easy to miss. `V` broadcasting over `seq_q` inside a reduction over `seq_k`—that's nearly invisible. Two of five broadcasts escape notice entirely. The audit reveals them.

Take a breath. This was the densest section of the chapter. Three lines of code, five broadcasts, five backward reductions. If it felt like a lot—good. It is a lot. But here is what matters: you never have to do this audit for these three lines again. The next time you see `sum[seq_k](weights * V)` in a transformer, you already know: `V` broadcasts over `seq_q`, and the backward pass sums over `seq_q`. The audit isn't a procedure you run every time. It's an instinct you build by running it once and then remembering the answer.

The procedure worked. It gave you the answer. Now the answer is yours.

The Audit Without Einlang

You do not need Einlang to perform a broadcast self-audit. You need to know which coordinate is being broadcast over. The question is the same in any framework. The difference is how hard the framework makes it to answer.

In PyTorch, broadcasting is shape-driven. `(32, 64) + (64,)` broadcasts along axis 0. Which coordinate is axis 0? The code doesn't say. You infer it from context: axis 0 is probably `batch`, axis 1 is probably `feature`. But "probably" is not a check.

In Einlang, broadcasting is name-driven. `out[i, j] = A[i, j] + bias[j]` omits `i`. The omitted coordinate is the broadcast coordinate. The code says it.

The audit questions are the same. But in PyTorch, answering Question 1 ("what coordinate am I broadcasting over?") requires shape reconstruction. In Einlang, answering Question 1 requires reading a bracket. The audit is the same. The effort is not.

In PyTorch, `output = projected + b_o` requires knowing that `projected` has shape `(batch, seq_q, d_out)` and `b_o` has `(d_out,)`. Transpose `projected` upstream, and the broadcast alignment silently changes. The Einlang version `projected[..b, seq_q, d_out] + b_o[d_out]` is layout-independent—the name `d_out` identifies the shared axis regardless of position. The audit is the same for 2D or 6D tensors, because the number of names, not the number of axes, determines the work.

What the Audit Reveals

The last broadcast you wrote—intentionally or not—is in your code right now, in some `A + b` or `scale * x` or `mean[dim]` with `keepdim=True`. Apply the three questions. Every broadcast is a claim of independence. The broadcast set—the coordinates the operand omits—is the claim written in set-subtraction notation.

In a typical codebase, at least one broadcast fails the semantic question: does the broadcasting operand genuinely not depend on those coordinates? "Probably" or "I think so" is not a yes. That broadcast is a claim of independence that is not confidently true. It is a bug waiting for the right input shape.

The audit reveals it. Not because the audit is sophisticated—it is three questions and a set subtraction. Because the audit asks a question the code itself does not. Positional notation records that a broadcast happened. Named notation records which coordinate it happened over. The difference is whether the claim was recorded.

The audit catches individual broadcasts. But normalization functions—LayerNorm, RMSNorm, GroupNorm, InstanceNorm—share a deeper structure: a reduce-broadcast-elementwise skeleton that is identical across all of them, differing only in which coordinates play which roles.

Chapter 5 · Blocks and Skeletons

> "If you have a procedure with ten parameters, you probably missed some."
>
> — Alan Perlis

Combinations · Pack polymorphism and normalization skeletons

You've written four normalization functions in the past month.

LayerNorm for the Transformer. RMSNorm for the memory-efficient run. GroupNorm for the convolutional front-end. InstanceNorm for style transfer. Four functions, four different `dim` arguments, four different reshape chains. You checked each one in, wrote the tests, moved on.

Now look at them side by side. Really look. They are the same function.

Different `dim` values. Different internal statistics. But the structure — reduce, broadcast, elementwise, scale — is identical. The four functions differ only in *which coordinates* the reduction consumes and *which coordinates* the parameters broadcast over. Every other line is boilerplate that the skeleton demands.

What you discovered by inspection — "these four functions share a skeleton" — is invisible in the positional code. The `dim` arguments are different integers. The reshape chains are different lengths. The parameter shapes follow different conventions. The skeleton is there, but it's encoded as shape arithmetic. Only the coordinate names make it visible.

A coordinate pattern wrapped in a function signature produces a reusable block. Several such blocks reveal a common skeleton, dressed in different coordinate names. This chapter extracts that skeleton and shows what it buys.

The Anatomy of a Coordinate-Aware Function

Here is the complete form:

```
fn normalize[coord](x: [f32; ..left, coord, ..right])
    -> [f32; ..left, coord, ..right]
{
    let m[..left, ..right] = max[coord](x[..left, coord, ..right]);
    let centered[..left, coord, ..right] = x[..left, coord, ..right] - m[..left, ..right];
    let scale[..left, ..right] = sum[coord](centered[..left, coord, ..right] ** 2.0);
    centered[..left, coord, ..right] / (scale[..left, ..right] ** 0.5 + 1e-5)
}
```

Look at the coordinate parameter `[coord]` in the signature. It appears in brackets after the function name and in the type signatures. It is not a value—you cannot do arithmetic on it. It is a coordinate identity. When the function body uses `max[coord]` and `sum[coord]`, it is verified that `coord` is the same parameter declared in the signature.

Now look at the rest packs `..left` and `..right`. They stand for whatever coordinates surround `coord` in the actual argument. If the caller passes `x[b, t, f]` and writes `normalize[f](x)`, then `..left` binds to `[b, t]` and `..right` binds to nothing. If the caller passes `x[b, h, f, d]` with `normalize[f](x)`, then `..left` binds to `[b, h]` and `..right` binds to `[d]`. The function body is polymorphic over the surrounding structure.

Trace the coordinate flow. Inside the function body, the coordinate `coord` is in scope. It can be used in reductions (`max[coord]`, `sum[coord]`), in indexing, and implicitly in the output shape. The packs `..left` and `..right` flow from the input signature to the output signature.

Finally, the return type annotation `-> [f32; ..left, coord, ..right]` tells you which coordinates survive. If the function body accidentally consumed `coord` without reconstructing it, or dropped a pack, a coordinate mismatch is reported.

Packs and Polymorphism

Packs are what make coordinate-aware functions reusable across different tensor ranks.

A pack that comes before the coordinate of interest absorbs leading dimensions. Write `..left` or `..b` and all leading axes collapse into one named group.

A pack after the coordinate absorbs trailing dimensions. Write `..right` or `..rest` and everything after the named coordinate is captured.

A named spatial pack absorbs spatial dimensions as a group rather than individually. Write `..s` and a function that reshapes spatial coordinates can treat them as a unit:

```
fn move_channel[channel, ..s](x: [f32; channel, ..s])
    -> [f32; ..s, channel]
{
    x[..s, channel]
}
```

When a pack is ambiguous at the call site, the caller disambiguates by grouping:

```
id_axes[(height, width)](x)
```

Pack parameters make coordinate-aware functions rank-polymorphic: the same function works on 2D, 3D, 4D, or higher-dimensional tensors, as long as the coordinate of interest exists somewhere in the shape.

How the Compiler Resolves Packs

The compiler resolves packs by finding the coordinate argument in the layout and splitting the surrounding axes. The signature is `fn layer_norm[coord](x: [f32; ..left, coord, ..right])`. The caller writes `layer_norm[channel](x)` where `x` has layout `[batch, channel, height, width]`.

The compiler walks `x`'s layout and finds `channel` at position 1. `..left` — everything before position 1 — binds to `[batch]`. `..right` — everything after — binds to `[height, width]`. The function body rewrites: `..left` → `[batch]`, `..right` → `[height, width]`, `coord` → `channel`. The function is now monomorphic for this call site.

If `channel` is nowhere in the layout — `x` has `[batch, seq, height, width]` — the compiler reports:

What if the layout doesn't match? The caller writes `layer_norm[channel](x)` but `x` has layout `[batch, seq, height, width]` — `channel` is nowhere in the layout. Step 1 fails. The compiler reports:

```
error: coordinate `channel` not found in argument layout
  → argument layout: [batch, seq, height, width]
  → called as: layer_norm[channel](x)
  → the coordinate parameter `coord` must match one coordinate in the argument
```

No guessing. No silent mismatch. The anchor is missing, and the compiler says so.

What about multiple named coordinates? A function can accept more than one coordinate in brackets:

```
fn max_over[j, k](x: [f32; ..left, j, ..right, k, ..rightmost])
    -> [f32; ..left, ..right, ..rightmost]
{
    max[j, k](x[..left, j, ..right, k, ..rightmost])
}
```

Two named coordinates — `j` and `k` — each acts as its own anchor. `..left` is everything before `j`. `..right` is everything between `j` and `k`. `..rightmost` is everything after `k`. Three surrounding packs, two anchors, one deterministic resolution. No caller grouping needed — the caller writes `max_over[h, w](x)`, two named coordinate arguments in brackets, and the compiler places each pack by position relative to the anchors.

The rule: **brackets hold what the caller must specify.** Named coordinates and packs that need caller disambiguation go in brackets. Packs that can be determined by elimination stay out — they appear in the value parameter's shape but not in the bracket list.

Selection Reductions

You know `max[class]`. It gives you the largest value.

Now meet its cousin:

```
let pred[b] = argmax[class](logits[b, class]);
```

Both operate on the same input. Both consume `class`. They return different things.

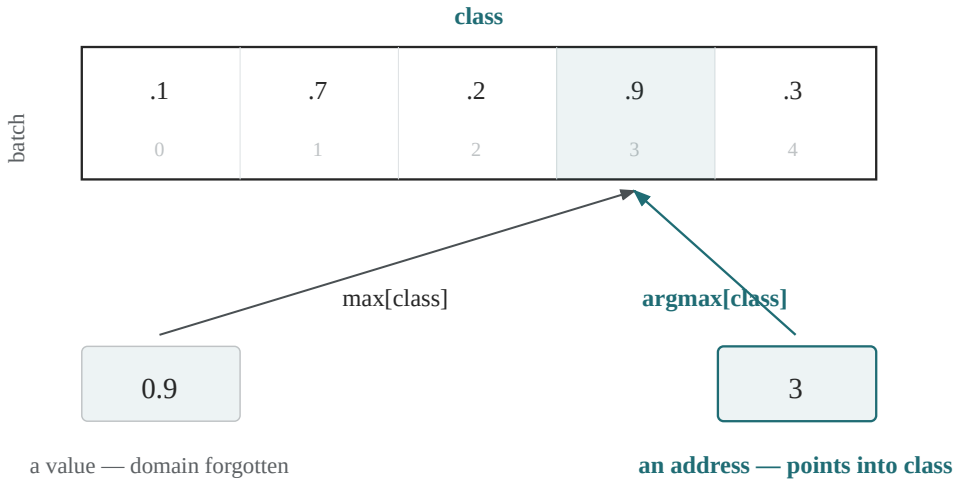


Figure 6: max vs argmax on the same input tensor.

Look at the figure. What does `max` return? What does `argmax` return? Before reading on: one returns a value extracted FROM the domain. The other returns an address that points INTO the domain. Which is which, and why does it matter?

`max[class]` extracts. The result is a scalar—the value 0.9, stripped of its `class` identity. You could pass that scalar to any function. The fact that it came from `class` position 3 is forgotten.

`argmax[class]` returns a pointer. The result is the integer 3—not just any integer, but an address in the `class` coordinate space. This integer carries a **domain contract**: the compiler knows it is only meaningful when used to index into a tensor that also carries `class`.

This contract is enforced. If you write:

```
let pred[b] = argmax[class](logits[b, class]);
let best[b] = embeddings[pred[b], class];
```

The compiler checks: does `pred[b]` carry the `class` address domain? Yes—it came from `argmax[class]`. Does `embeddings` have a `class` coordinate? Yes—it appears in the indexing expression. Do the domains match? Yes. The indexing is valid.

Now suppose you accidentally write:

```
let pred[b] = argmax[class](logits[b, class]);
let best[b] = vocabulary[pred[b], token];
```

The compiler reports:

```
error: address domain mismatch
  → `pred` carries addresses in domain `class` (from argmax[class] at line 1)
  → `vocabulary` has coordinate `token` at this position
  → expected coordinate `class`, found `token`
  → these domains are different; indexing across domains is not allowed
```

In PyTorch, this is silent. `pred` is a tensor of integers. `vocabulary[pred]` is a tensor of gathered values. Whether `pred`'s integers refer to class indices or token indices or pixel positions is invisible to the type system. Nothing prevents you from using classification predictions to index into a token embedding table. The bug survives testing until a user passes an out-of-range class index to a token embedding and the program crashes with a cryptic CUDA error, or worse, silently returns garbage.

The address domain contract is what makes `argmax` conceptually different from `max`. `max` says "give me the biggest thing in this set." `argmax` says "tell me WHERE the biggest thing is, because I need to go there." The WHERE is only meaningful relative to the coordinate domain it points into. The compiler tracks which domain that is.

This distinction propagates. If you have:

```
let top_k[b, k] = topk[class](logits[b, class], 5);
```

`top_k` is a tensor of five addresses per batch element, each an address in `class`. You can use it to index into any tensor that carries `class`. The compiler knows. The reader knows. The contract travels with the value.

Selection reductions—`argmax`, `argmin`, `topk`—are the only operations that produce address-typed values. Their results are not interchangeable with plain integers. They are typed by the coordinate domain they reference. This is the domain contract from Chapter 2, applied to the concept of "where" instead of "what."

Four Normalizations

Here are four normalization implementations. Read them. Find what they share.

LayerNorm:

```
def layer_norm(x, gamma, beta, eps=1e-5):
    mean = x.mean(dim=-1, keepdim=True)
    var = ((x - mean) ** 2).mean(dim=-1, keepdim=True)
    return (x - mean) / (var + eps).sqrt() * gamma + beta
```

RMSNorm:

```
def rms_norm(x, gamma, eps=1e-5):
    rms = (x ** 2).mean(dim=-1, keepdim=True).sqrt()
    return x / (rms + eps) * gamma
```

GroupNorm:

```
def group_norm(x, num_groups, gamma, beta, eps=1e-5):
    N, C, H, W = x.shape
    x = x.reshape(N, num_groups, C // num_groups, H, W)
    mean = x.mean(dim=(2, 3, 4), keepdim=True)
    var = x.var(dim=(2, 3, 4), keepdim=True)
    x = (x - mean) / (var + eps).sqrt()
    return x.reshape(N, C, H, W) * gamma + beta
```

InstanceNorm:

```
def instance_norm(x, gamma, beta, eps=1e-5):
    N, C, H, W = x.shape
    mean = x.mean(dim=(2, 3), keepdim=True)
    var = x.var(dim=(2, 3), keepdim=True)
    return (x - mean) / (var + eps).sqrt() * gamma + beta
```

Look at the code. The common structure is visible across all four functions. Can you see the specific sequence of operations they all perform? What varies between them?

Here is what they share:

1. **Compute a statistic** (mean, rms, or both) by reducing over one or more dimensions.
2. **Broadcast the statistic back** over the reduced dimensions (via `keepdim=True`).
3. **Apply the statistic elementwise** (subtract and divide).
4. **Scale and shift** with learned parameters that broadcast over the non-feature dimensions.

Every one of these functions does: reduce, broadcast, elementwise, scale. The difference is only *which* dimensions are reduced and *which* parameters broadcast over *which* remaining dimensions.

But look at the code again. Can you *see* the shared structure? In LayerNorm, it's `x.mean(dim=-1, keepdim=True)`. In GroupNorm, it's `x.mean(dim=(2,3,4), keepdim=True)`. In InstanceNorm, it's `x.mean(dim=(2,3), keepdim=True)`. The `dim` arguments are different integers. The `keepdim=True` flag is the same. The `* gamma + beta` ending is the same.

The pattern IS there—but it's encoded as shape arithmetic. This shared structure is a **skeleton**: a reduce-broadcast-elementwise-scale template parameterized by which coordinates are reduced and which are preserved. `dim=-1` means one thing in LayerNorm ("the last dimension") and a completely different set

of integers in GroupNorm ("dimensions 2, 3, and 4"). The skeleton is visible to a human who understands the dimension layout. It is invisible to a compiler. And it changes when the layout changes.

Now here is the same skeleton in Einlang:

Function	Reduction coords	Broadcast params	Survivors
Softmax	`q` (max), `k` (sum)	none	`..b`, `j`
LayerNorm	`f` (mean ×2)	`gamma[f]`, `beta[f]`	`..b`, `f`
RMSNorm	`f` (mean)	`gamma[f]`	`..b`, `f`
GroupNorm	`c_in_group`, `..s`	`gamma[g, c_in_group]`, `beta[g, c_in_group]`	`..b`, `g`, `c_in_group`, `..s`
InstanceNorm	`..s`	`gamma[c]`, `beta[c]`	`..b`, `c`, `..s`

The skeleton is visible in the table because the coordinates are named. Each column says *what*, not *where*. The reduction column names the consumed coordinates. The broadcast column names the parameters and their coordinate sets—the difference between the output set and the parameter set is the broadcast claim. This is coordinate set subtraction from Chapter 2, applied to four functions at once. The survivors column names what's left.

You might wonder: how does the compiler know that `mean[f]` reduces over `f` and not over `..b`? The answer is that `f` is a named coordinate — the bracket says `mean[f]`, not `mean[..b]`. Packs can appear in reduction brackets too (`mean[..s]`), but the compiler resolves them to concrete coordinates the same way it resolves them in signatures: from the anchor. The reduction bracket is coordinate flow. The pack is a group. The group resolves to its members. The reduction consumes them all.

Here is the same pattern drawn instead of tabulated:

Read across any row: the five columns are the same. The coordinate names change. That is the skeleton.

In a positional API, all four collapse to a single `dim` argument whose meaning shifts with the surrounding layout. `LayerNorm` and `RMSNorm` both use `dim=-1`—but normalize different statistics. `GroupNorm` uses three reduction dimensions buried in a `reshape` chain. The skeleton is invisible.

In a named-coordinate API, the skeleton is a template you can check. The reduction bracket names the consumed coordinates. The indexing pattern names the survivors. The broadcast parameters name the omission. A reviewer can verify that the broadcast coordinate in LayerNorm matches the broadcast coordinate in the gradient without reconstructing both from positional offsets.

This is abstraction: recognizing a pattern, naming it, and reusing it. The pattern is "normalize with named coordinates." Each instance fills in the specific coordinates. The skeleton is constant.

The discovery exercise—comparing four implementations, finding their shared structure—is what you do every time you read unfamiliar tensor code. Names carry the structure. Positions hide it.

Derive it yourself: Spot the broadcast set. Take the LayerNorm row from the table above. The output has `{..b, f}`. `gamma[f]` has `{f}`. What is the broadcast set for `gamma`? Compute it: `{..b, f} \ {f} = {..b}`. `gamma` broadcasts over every batch dimension. Now take GroupNorm. `gamma[g, c_in_group]` has `{g, c_in_group}`. The output has `{..b, g, c_in_group, ..s}`. Broadcast set: `{..b, ..s}`. `gamma` is silent on batch and spatial—exactly as intended. Try the same for InstanceNorm: what does `beta[c]` broadcast over? The answer is in the set subtraction. The table above tells you the answer if you're stuck. But do the subtraction yourself first. The subtraction is the check.

Now let's put this claim to the test. Here is a real GroupNorm implementation in PyTorch:

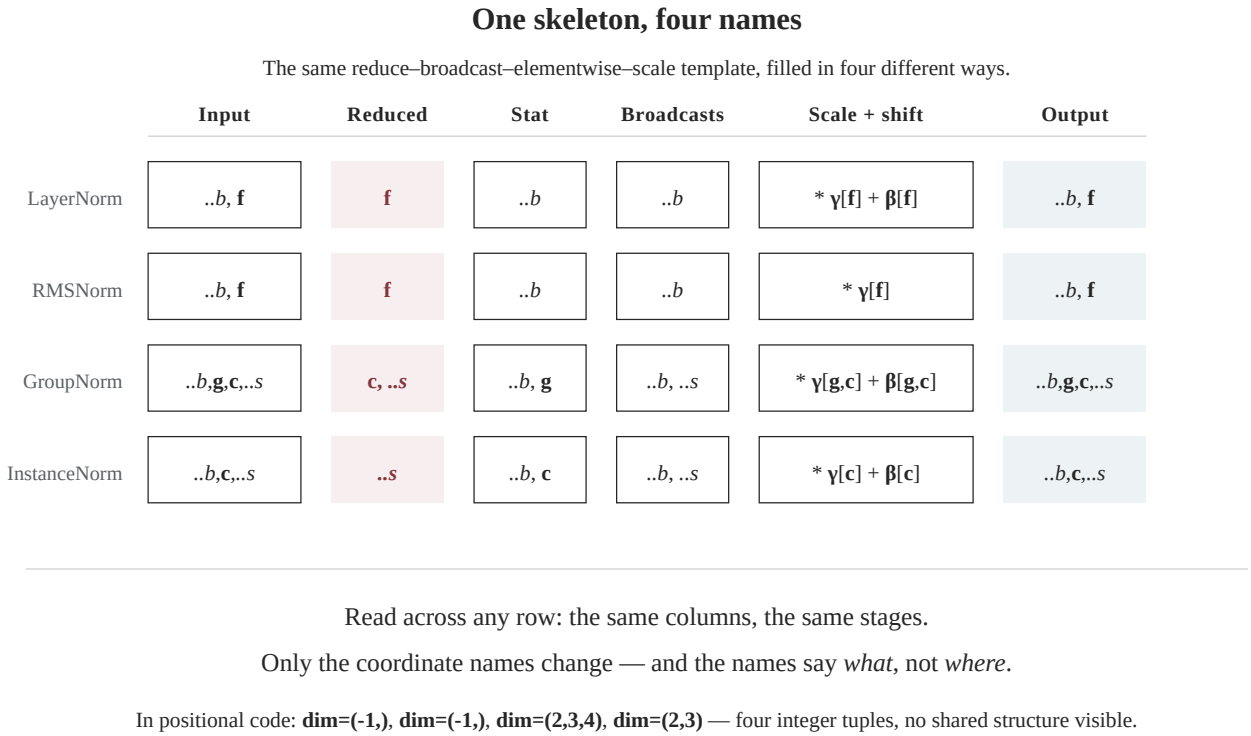


Figure 7: Four normalization variants, five columns each. The pipeline is identical; only the coordinate names change.

```
def group_norm(x, num_groups, gamma, beta, eps=1e-5):
    N, C, H, W = x.shape
    x = x.reshape(N, num_groups, C // num_groups, H, W)
    mean = x.mean(dim=(2, 3, 4), keepdim=True)
    var = x.var(dim=(2, 3, 4), keepdim=True)
    x = (x - mean) / (var + eps).sqrt()
    x = x.reshape(N, C, H, W)
    return x * gamma + beta
```

Stop and read this carefully. Ask yourself: which dimensions are being reduced by `dim=(2, 3, 4)`? What do positions 2, 3, and 4 correspond to? You need to trace backward through the `reshape`—position 2 is `C // num_groups` (channels per group), position 3 is `H`, position 4 is `W`. But this reasoning depends on the reshape chain. If the reshape changes, the `dim` tuple must change with it. If someone adds a temporal dimension before the spatial ones, the tuple shifts silently.

Now compare the Einlang version:

```
fn group_norm[g, c_in_group, ..s](x: [f32; ..b, g, c_in_group, ..s],
    gamma: [f32; g, c_in_group], beta: [f32; g, c_in_group])
    -> [f32; ..b, g, c_in_group, ..s]
{
    let m[..b, g] = mean[c_in_group, ..s](x[..b, g, c_in_group, ..s]);
    let v[..b, g] = mean[c_in_group, ..s](
        (x[..b, g, c_in_group, ..s] - m[..b, g]) ** 2.0
    );
    let y[..b, g, c_in_group, ..s] =
        (x[..b, g, c_in_group, ..s] - m[..b, g])
        / (v[..b, g] + 1e-5) ** 0.5;
    y[..b, g, c_in_group, ..s] * gamma[g, c_in_group] + beta[g, c_in_group]
}
```

The reduced coordinates are named: `c_in_group` and `..s`. No reshape needed. No positional arithmetic needed. If a temporal dimension is added, `..s` absorbs it—the reduction bracket stays the same. If `num_groups` changes, the coordinate `g` handles it—its domain just has a different size.

The deeper point: in a positional API, "feature" is `dim=-1`—the last axis. That works when the tensor is 2D. When it becomes 4D, "feature" no longer fits in a single integer. It spans three positions: channels per group, height, width. Positional code handles this with a reshape-permute-reshape dance that groups and ungroups those dimensions. Named coordinates handle it without the dance: one semantic coordinate (`c_in_group`) plus a spatial pack (`..s`) cover what `dim=(2,3,4)` covers in the positional version. The names don't change when the layout changes.

This is what packs buy you. `..s` absorbs however many spatial dimensions exist. The same `GroupNorm` skeleton works whether spatial covers one axis or three. `mean[c_in_group, ..s]` says exactly what is consumed—no reshape chain to reverse-engineer.

Now one more question. Suppose you encounter a new normalization variant—say, normalize only over the spatial dimensions, keeping the channel-group dimension intact. What would you change?

Think about it. In the Einlang version, you change one thing: remove `c_in_group` from the reduction bracket. `mean[..s](...)` instead of `mean[c_in_group, ..s](...)`. The skeleton is unchanged. The coordinate names carry the design decision.

In the PyTorch version, you'd change `dim=(2, 3, 4)` to `dim=(3, 4)`—but only if the reshape hasn't

changed the position of the spatial dimensions. If someone added a temporal axis between `c_in_group` and `H`, the tuple would need to shift to `dim=(4, 5)`. The fragility is not in the concept—it is in the notation's inability to record *which* dimensions are spatial.

Skeletons Compose

The normalization skeleton and the attention skeleton compose. A Transformer block is LayerNorm, then attention, then another LayerNorm, then a feedforward. In a positional implementation, the norm dimensions and attention dimensions share the `dim=-1` convention—until one of them shouldn't.

Before reading the code, ask: which coordinates does a Transformer block consume and reconstruct? The answer should be visible in the function signature alone.

Here is a complete Transformer block skeleton in Einlang:

```
fn transformer_block[head, seq, d, d_ff](
    x: [f32; ..b, seq, d],
    W_q: [f32; head, d, d_k],
    W_k: [f32; head, d, d_k],
    W_v: [f32; head, d, d_v],
    W_o: [f32; head, d_v, d],
    W_1: [f32; d, d_ff],
    W_2: [f32; d_ff, d],
    gamma1: [f32; d], beta1: [f32; d],
    gamma2: [f32; d], beta2: [f32; d]
) -> [f32; ..b, seq, d]
{
    // LayerNorm 1
    let norm1[..b, seq, d] = layer_norm[d](x[..b, seq, d], gamma1[d], beta1[d]);

    // Multi-head attention
    let attn_out[..b, seq, d] = attention[head, seq, seq, d](
        norm1[..b, seq, d], norm1[..b, seq, d], norm1[..b, seq, d],
        W_q[head, d, d_k], W_k[head, d, d_k], W_v[head, d, d_v], W_o[head, d_v, d]
    );

    // Residual connection
    let res1[..b, seq, d] = x[..b, seq, d] + attn_out[..b, seq, d];

    // LayerNorm 2
    let norm2[..b, seq, d] = layer_norm[d](res1[..b, seq, d], gamma2[d], beta2[d]);

    // Feedforward
    let ff[..b, seq, d_ff] = relu(sum[d](norm2[..b, seq, d] * W_1[d, d_ff]));
    let ff_out[..b, seq, d] = sum[d_ff](ff[..b, seq, d_ff] * W_2[d_ff, d]);

    // Residual connection
    res1[..b, seq, d] + ff_out[..b, seq, d]
}
```

Every coordinate that is consumed is named in a bracket. Every coordinate that survives is named in the output pattern. The `d` coordinate is consumed in two reductions (`layer_norm[d]` and `attention[..., d]`) and reconstructed each time. The `head` coordinate appears on the attention weights but not on the input `x`—it splits the feature dimension without changing the data layout.

Now ask: if you wanted to change this to a cross-attention block where queries come from one sequence and keys/values from another, what would you change? In a positional implementation, the code wouldn't change at all—the same `attention(Q, K, V)` call works for both. The difference is only in which tensors you pass. In the Einlang version, you'd change the signature: the first `norm1` gets coordinate `seq_q`, the second and third get coordinate `seq_k`. The code change is a coordinate name swap. The reader sees the architectural decision in the type signature.

Skeletons compose because coordinate contracts compose. The output coordinates of one function become the input coordinates of the next. The compiler traces the flow. You trace the meaning.

Pause here. Look back at the Transformer block on the previous page. Find every bracket. For each one, ask: is this coordinate being consumed, reconstructed, or passed through? The answer tells you whether the line is a reduction (consume), a normalization (consume then reconstruct), or a passthrough (neither). Three categories. The entire block—attention, layer norm, feedforward, residual—is built from them.

Spot the Skeleton

Here are four Einlang function signatures. Three implement normalization variants. One doesn't. Can you spot the odd one out?

```
fn A[j](x: [f32; ..b, j]) -> [f32; ..b, j]
fn B[coord](x: [f32; ..b, coord]) -> [f32; ..b]
fn C[f](x: [f32; ..b, f]) -> [f32; ..b, f]
fn D[t](x: [f32; ..b, t]) -> [f32; ..b, t]
```

Look at the return types and notice the pattern yourself before reading on.

Here is the pattern: Function B is the odd one out. Its return type is `[f32; ..b]`—the coordinate `coord` is missing. It was consumed and not reconstructed. Functions A, C, and D all return `[f32; ..b, <coordinate>]`—the coordinate survives. B is a reduction function (like `sum[coord]`). A, C, and D are normalization functions that preserve the coordinate.

Now the deeper question: **why** is this distinction visible in the type signature? Because the skeleton is more than "reduce then broadcast." The skeleton is "reduce, then broadcast back to **reconstruct the consumed coordinate in the output.**" A pure reduction consumes and doesn't reconstruct—the coordinate disappears from the return type. A normalization consumes and reconstructs—the coordinate reappears. The difference between "gone forever" and "gone and returned" is the difference between a reduction and a normalization. The return type records it.

The skeleton is visible in the type signature. The reduction bracket in the body (`max[coord]`, `mean[f]`, `sum[j]`) tells you what is consumed. The return type tells you whether it was reconstructed. A reader can distinguish a normalization from a reduction without reading the body—the coordinate flow is in the signature.

This is the abstraction layer. The function signature says: "I operate on coordinate `j`. I preserve `j` in the output. Everything else passes through." The body fills in the specific computation. The signature is the contract. The body is the implementation. And the contract is checkable.

Derive InstanceNorm

You've seen the table. InstanceNorm normalizes each sample's each channel independently over the spatial dimensions. In 2D: for each `(N, C)`, compute mean and variance over `(H, W)`. The coordinates it reduces over and the coordinates that survive are visible in the operation's signature. Which coordinates does it consume? Which survive? What parameters broadcast?

Here is the answer:

```
fn instance_norm[c, ..s](x: [f32; ..b, c, ..s],
                                gamma: [f32; c],
                                beta: [f32; c])
    -> [f32; ..b, c, ..s]
{
    let m[..b, c] = mean[..s](x[..b, c, ..s]);
    let v[..b, c] = mean[..s]((x[..b, c, ..s] - m[..b, c]) ** 2.0);
    let y[..b, c, ..s] =
        (x[..b, c, ..s] - m[..b, c]) / (v[..b, c] + 1e-5) ** 0.5;
    y[..b, c, ..s] * gamma[c] + beta[c]
}
```

The reduced coordinates are `..s`. The surviving coordinates are `..b`, `c`, and `..s`—the spatial coordinates are consumed for the statistics but preserved in the output. The broadcast parameters are `gamma[c]` and `beta[c]`, which broadcast over `..b` and `..s`.

Three things to observe: - `..s` absorbs however many spatial dimensions there are. - `..s` is placed in the reduction bracket because it's consumed for the statistics. - `..s` is kept in the return type because the output is not a scalar—it's a tensor with spatial dimensions.

These three observations together capture the skeleton. The coordinate names carry the design: `mean[..s]` says "I am consuming the spatial dimensions." The return type `[f32; ..b, c, ..s]` says "the spatial dimensions survive." The contradiction resolves: `..s` is consumed in the reduction but reconstructed in the output—the signature guarantees it.

Now consider: what if InstanceNorm should normalize over `c` as well? You'd change the reduction bracket to `[c, ..s]`. One change. The skeleton is the same. The coordinate name carries the design decision.

Coordinate Facts Flow

In PyTorch, a tensor leaves the data loader as `(32, 64)`. By the time it reaches the loss function, it has passed through five layers, three reshapes, and a transpose. The numbers have changed. The identities — `batch`, `feature` — were never recorded. They are gone. Every function that receives the tensor must re-discover what the dimensions mean from their positions.

In Einlang, the identities survive. Not because the compiler is clever. Because the rule is brutally simple: coordinates only change when an operation explicitly changes them.

Square a tensor. Coordinates unchanged. Add a constant. Coordinates unchanged. Pass through a pointwise function. Coordinates unchanged. Coordinate facts survive every operation that does not explicitly manipulate them. Reductions consume coordinates. Declarations introduce them. Coordinate-aware function calls thread them through signatures. Everything else — arithmetic, function calls that return tensors, `if` expressions — preserves them. You declare coordinates once. They propagate from that point forward.

This is stronger than type inference. Type inference says `x ** 2.0` has the same type as `x`. Coordinate flow says it has the same *identity* as `x`. The identities are not inferred from runtime shapes — they are propagated from declarations. A PyTorch tensor carries `(32, 64)` at runtime. It does not carry `(batch, channel)`. The identities are lost the moment the tensor leaves the data loader. In Einlang, they survive every intermediate binding, every arithmetic expression, every function return. The source is the declaration. The flow is forward.

Three cases capture the entire system:

1. `let y = x + 1.0;` — `y` has the same coordinates as `x`. Addition doesn't touch identity.
2. `let y = sum[j](x[i, j]);` — `j` is consumed. `y` has `[i]`. Reduction is the only operation that removes a coordinate.
3. `let y[i, j] = x[j, i];` — same names, swapped positions. `y` has `[i, j]`. This is a transpose. Not `x.transpose(0, 1)`. Not `x.permute(1, 0)`. Just `y[i, j] = x[j, i]`. The names carry the permutation. No position counting needed.

That's it. Three cases. Every coordinate flow in every Einlang program reduces to these three. The compiler traces them mechanically. You can trace them by hand. The names are the thread.

Reduction is the only consumer. Declaration is the only producer. Everything else is a pipe. Coordinates flow forward from declaration to consumption. Names are not inferred from runtime shapes. They are propagated from source declarations. A name, once declared, survives every operation that does not explicitly consume it.

Pause here. Open a file you wrote last week—any file with tensor operations. Pick five lines that move data: a reshape, a reduction, a broadcast, a permute, an elementwise operation. For each: which of the three cases applies? Coordinate unchanged? Coordinate consumed? Coordinates swapped?

If the answer is "I can't tell without printing shapes," you've found a place where a coordinate name would have helped your future self. If you can classify all five, you already think in names—you just haven't been writing them down. The three cases aren't the language's rules. They're the rules the coordinate structure already follows, whether or not your notation records it. The notation just makes them visible.

Coordinates also have a fourth role: time. A coordinate that doesn't just sit there, but flows.

The Skeleton Pattern, Seen in Signatures

The four-normalizations table revealed that LayerNorm, RMSNorm, GroupNorm, and InstanceNorm share a skeleton. But how do you discover the skeleton in the first place? Not by reading a table. By writing the functions and noticing what changes.

The skeletons are already visible in these signatures—no body needed, just the coordinate names and reduction brackets:

```
fn softmax[j](x: [f32; ..b, j]) -> [f32; ..b, j];
fn layer_norm[f](x: [f32; ..b, f], gamma: [f32; f], beta: [f32; f]) -> [f32; ..b, f];
fn instance_norm[..s, c](x: [f32; ..b, c, ..s], gamma: [f32; c], beta: [f32; c]) -> [f32; ..b, c,
```

Look at only the signatures. Can you tell which one reduces over which coordinate? In `softmax[j]`, the coordinate parameter `j` tells you: normalize over `j`. In `layer_norm[f]`, `f` tells you: normalize over `f`. In `instance_norm[..s, c]`, both `..s` and `c` are in the signature—but which one is reduced? The answer depends on which coordinate is placed in the reduction bracket in the body. The signature alone can't tell you—it can only tell you which coordinates exist. The body tells you which ones are consumed.

Now compare to the positional versions:

```
def softmax(logits, dim=-1): ...
def layer_norm(x, normalized_shape, ...): ...
def instance_norm(x, ...): ...
```

The positional signatures tell you even less. `softmax` has `dim`—a positional hint. `layer_norm` has `normalized_shape`—a shape hint, not a coordinate hint. `instance_norm` has no hint at all—you must read the body to know which dimensions are reduced.

What is happening when you read these signatures: you are asking "which coordinates are needed for this computation?" The ones in the brackets are the answer. The ones not in the brackets pass through. The signature is the skeleton's outline.

The next time you write a function that takes a tensor and returns a tensor, write its coordinate signature in a comment before the body. Which coordinates survive? Which are consumed? Which broadcast? If you can't answer from the code alone, the signature is the place to start.

Discovering a Coordinate Contract

The sections you just read show finished products. `layer_norm[f]` with its correct skeleton. `group_norm[g, c_in_group, ..s]` with its correct skeleton. The skeletons are clean. The coordinate choices look obvious.

They were not obvious when first written. The finished signature is the last thing you arrive at, not the first. This section walks through the process. You are going to do the walking.

Start Concrete

Standard LayerNorm computes statistics uniformly over the feature dimension. Weighted LayerNorm relaxes this: each feature position has a learned weight that scales its contribution to the mean and variance.

Here is your first attempt. It compiles.

```
fn weighted_layer_norm[f](x: [f32; ..b, f],
                          w: [f32; f],
                          gamma: [f32; f],
                          beta: [f32; f])
    -> [f32; ..b, f]
{
    let w_sum[..b] = sum[f](w[f]);
    let mean[..b] = sum[f](x[..b, f] * w[f]) / w_sum[..b];
    let centered[..b, f] = x[..b, f] - mean[..b];
    let var[..b] = sum[f](w[f] * centered[..b, f] ** 2.0) / w_sum[..b];
```

```
    let y[..b, f] = centered[..b, f] / (var[..b] + 1e-5) ** 0.5;
    y[..b, f] * gamma[f] + beta[f]
}
```

Before reading on, look at the contract for `w`. What coordinate set does `w` claim? The annotation says `w: [f32; f]`. That means `w` has exactly one coordinate: `f`. The weight cannot depend on batch. The weight cannot depend on a group. The weight is per-feature, period.

This works for standard Weighted LayerNorm. The code compiles. The coordinate flow is clean. You run the five checks from Chapter 3 and every one passes. You commit.

A week later, a colleague asks: "Can I use this for group-weighted normalization? I need the weights to vary per group and per channel-in-group."

You open the file. You stare at `w: [f32; f]`.

Find the baked-in assumption. Before reading on, write it down.

`w: [f32; f]` hardcodes the weight's coordinate set to `{f}`. The skeleton—weighted mean, weighted variance, normalize, scale—does not require `{f}`. The skeleton requires only that the weight live in whatever coordinate set the reduction consumes. The assumption was not wrong for the problem you solved. It was wrong for the skeleton.

The colleague's question did not ask for a new skeleton. It asked the existing skeleton to accept a different weight coordinate set.

Parameterize

Write the contract for the group-weighted version. What changes?

You change the coordinate set of `w`. Instead of `w: [f32; f]`, you need `w: [f32; g, c_in_g]`. The reduction bracket changes from `sum[f]` to `sum[c_in_g, ..s]`. The parameters `gamma` and `beta` also move from `{f}` to `{g, c_in_g}`.

Here is the result:

```
fn group_weighted_norm[g, c_in_g, ..s](
    x: [f32; ..b, g, c_in_g, ..s],
    w: [f32; g, c_in_g],
    gamma: [f32; g, c_in_g],
    beta: [f32; g, c_in_g]
) -> [f32; ..b, g, c_in_g, ..s]
{
    let w_sum[..b, g] = sum[c_in_g, ..s](w[g, c_in_g]);
    let mean[..b, g] = sum[c_in_g, ..s](
        x[..b, g, c_in_g, ..s] * w[g, c_in_g]
    ) / w_sum[..b, g];
    let centered[..b, g, c_in_g, ..s] =
        x[..b, g, c_in_g, ..s] - mean[..b, g];
    let var[..b, g] = sum[c_in_g, ..s](
        w[g, c_in_g] * centered[..b, g, c_in_g, ..s] ** 2.0
    ) / w_sum[..b, g];
    let y[..b, g, c_in_g, ..s] =
        centered[..b, g, c_in_g, ..s] / (var[..b, g] + 1e-5) ** 0.5;
```

```
    y[..b, g, c_in_g, ..s] * gamma[g, c_in_g] + beta[g, c_in_g]
}
```

The skeleton is identical to the first attempt. Compare the two, line by line. The only difference is the coordinate set of `w`, `gamma`, and `beta`, and the corresponding reduction brackets. The computation—weighted sum, centered, variance, normalize—is the same sequence in the same order.

Now compare the two contracts. In the first: `w` claims `{f}`. In the second: `w` claims `{g, c_in_g}`. The skeleton does not care which set. It cares only that the weight's coordinate set matches the reduction bracket and that the reduction bracket is a subset of the input's coordinate set.

Here is the general skeleton:

```
fn weighted_norm[stat_coords, ..survivors](
    x: [f32; ..b, stat_coords, ..survivors],
    w: [f32; stat_coords],
    gamma: [f32; stat_coords, ..survivors],
    beta: [f32; stat_coords, ..survivors]
) -> [f32; ..b, stat_coords, ..survivors]
```

`stat_coords` names the coordinate set the weight lives in and the reduction consumes. `..survivors` names additional coordinates that exist in the input and survive in the output—the reduction bracket includes them, and the return type reconstructs them. Standard Weighted LayerNorm: `stat_coords = f`, `..survivors` empty. Group-Weighted LayerNorm: `stat_coords = {g, c_in_g}`, `..survivors = ..s`. One signature, every variant.

The Loop

You started with a concrete contract—`w: [f32; f]`—that compiled and passed every check. A question from outside the code exposed the assumption you did not notice you made. You parameterized the assumption. The skeleton now covers every weighted normalization variant, present and future.

You did not arrive at the general skeleton by starting general. You arrived by starting concrete, finding the assumption, and widening it. The concrete is not a stepping stone you discard. The concrete is the only way to find the assumption.

This is the design loop:

1. Write the concrete version. Make it compile.
2. Find the assumption baked into the contract.
3. Parameterize it.
4. Verify the parameterized version still covers the concrete case.

You just ran it.

What Skeletons Reveal

Softmax, LayerNorm, RMSNorm, GroupNorm, InstanceNorm—five different functions, one skeleton. The skeleton is not limited to normalization. Every operation that follows a reduce-broadcast-elementwise pattern is a skeleton.

Consider the `mean(`, `sum(`, or `max(` calls in your codebase. Each one followed by a broadcast is a reduction-statistic-broadcast pattern. The skeleton is there, whether the code names it or not. When the `dim` argument is an integer, the question "which coordinate is reduced?" has no answer in the code—the

reduction's identity is a position, and the position depends on layout. If the layout changes—a dimension inserted, a transpose applied—the integer now points at a different coordinate, and the skeleton silently shifts shape.

Consider every `* gamma + beta` or `* scale + shift`. It is a broadcast-parameter suffix of a normalization skeleton. Which coordinates do `gamma` and `beta` broadcast over? In a positional API, the answer is "all dimensions except the one gamma was defined on," which requires knowing gamma's rank and the convention for parameter placement. The broadcast is correct by convention, not by construction.

Any two functions that share a skeleton have Einlang signatures. Even without using Einlang, writing `fn name[consumed](x: [..b, consumed]) -> [..b, consumed]` as a comment above the function reveals the skeleton. Two functions with matching signatures share a skeleton. Two functions with different signatures serve different purposes and should have different signatures. The comment costs one line of typing. The check costs zero.

A normalization variant that normalizes over `batch` instead of `feature` changes `layer_norm[f]` to `layer_norm[batch]`. The signature change documents the architectural decision. In a positional API, changing `dim=-1` to `dim=0` silently shifts the coordinate—and hopes no other code depends on the output shape.

Every skeleton's forward pass is a reduce-broadcast-elementwise pattern. Every skeleton's backward pass is the Inversion Rule applied to that pattern. The coordinate names are the thread connecting the two directions.

Skeletons are spatial—they describe which coordinates are reduced and which survive at a single point in the computation graph. But coordinates can also flow through time: coordinates indexed by `t`, `t-1`, `t+1`, carrying a causality constraint that the compiler can check. The skeleton's forward/backward symmetry extends into a forward/backward recurrence—the Inversion Rule, applied to time.

Chapter 6 · The Arrow in the Bracket

> "Time is what keeps everything from happening at once."
>
> — Attributed to John Archibald Wheeler

Combinations · The minus sign makes the arrow

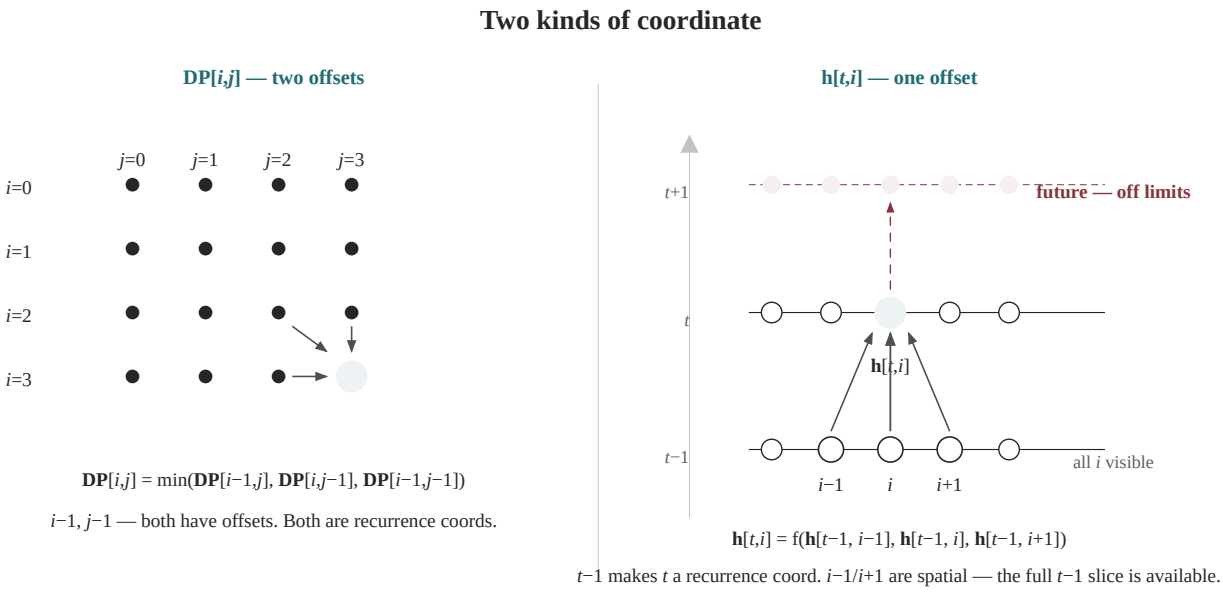
A coordinate you never see with an offset is a bookshelf. Pull any book off any shelf in any order — shelf 3 doesn't wait for shelf 2.

Now look at two programs that break that rule:

```
DP[i,j] = min( DP[i-1,j],  DP[i,j-1],  DP[i-1,j-1] )
```

```
h[t,i]  =   f( h[t-1,i-1], h[t-1,i],  h[t-1,i+1] )
```

Both have minus signs. Both read from earlier positions. But not every minus sign names a recurrence coordinate.



Left: both offsets on **DP** — i and j are recurrence. Right: only $t-1$ is on **h** — t is recurrence, i is a bookshelf.

Figure 8: Left: DP[i,j] with offsets on both i and j. Right: h[t,i] with offset only on t.

Look at the figure. Before reading on, answer this: on the left, which coordinates carry the dependency chain? On the right, which coordinate carries the arrow, and which are just spatial reads from a slice that already exists?

You saw it. On the left, both `i-1` and `j-1` appear on `DP` — both `i` and `j` are recurrence coords. The dependency flows in two directions. On the right, only `t-1` appears on `h`. The `i-1` and `i+1` read from `h` at `t-1` — where every `i` is already computed. `t` is recurrence; `i` is a bookshelf.

The compiler reads the minus sign on the declared variable. If the offset is measured against a coordinate of the variable being defined — `DP[i-1,j]` on `DP[i,j]` — that coordinate carries the dependency. If the offset is measured against a coordinate of a fully-materialized slice — `h[t-1,i+1]` on `h[t,i]` — it's a spatial read. The distinction is not the name. It is which coordinate the minus sign moves.

Not All Axes Are the Same

Look at this declaration:

```
u[t, i] = u[t-1, i] + f(u[t-1, i])
```

`t` and `i` are both written in brackets. The difference is not the name — it's the offset. `t` appears as `t-1` on the right-hand side. `i` never does.

In a spatial expression like `sum[i](A[i, k])`, `i` is just an index. You never write `i-1`. A coordinate that only appears as itself is concurrent — you sum over it, reduce along it, permute it — but you don't *recur* along it.

A coordinate that appears with an offset is different. `t-1` means step one depends on step zero, step two depends on step one. The offset makes `t` the direction of recurrence. Call it `t` or call it `step` — the minus sign is what matters.

Recurrence Declarations

In Einlang, time is just another coordinate—but one that appears in index arithmetic. You declare it with a range:

```
let u[t in 0..T, i] = init_temp(i);
let u[t in 1..T, i] = u[t-1, i] + alpha * (
    u[t-1, i+1] - 2.0 * u[t-1, i] + u[t-1, i-1]
);
```

The first clause defines `u` at `t=0`—the initial condition. The second clause defines `u` at every subsequent time step in terms of the previous step. `t-1` is a backward reference: the value at time `t` depends on the value at time `t-1`.

This is a recurrence. The coordinate `t` carries time's directional structure into the notation. You cannot write `u[t+1, i]` to define `u[t, i]`—that would be a forward reference, and it is rejected as a static error. Causality is not a comment. It is a syntactic constraint. If the index expression references an index greater than or equal to the declared index, it is a static error. This is not philosophy. It is subtraction: `t-1 < t`, valid; `t+1 > t`, rejected.

How does the compiler know? It does a mechanical scan. For every read of the declared variable in the body, at every coordinate position, it asks one question: **is this index expression exactly the loop variable?** If yes — no dependency, the coordinate is a bookshelf at this read. If no — the coordinate carries a dependency, it is a recurrence coordinate at this read. One question per position per read. That's it.

Walk through `u[t-1, i] + alpha * (...)` with the declaration `u[t in 0..T, i]`. The compiler finds one read of `u` in the body: `u[t-1, i]`. At position 0, the expression is `t-1`. Is `t-1` exactly the loop variable `t`? No. Recurrence on dim 0. At position 1, the expression is `i`. Is `i` exactly the loop variable `i`?

Yes. Not recurrence on dim 1. Result: dim 0 is recurrence, dim 1 is concurrent. The minus sign triggers it. No annotation. No `@recurrence`. The structural fact is in the code. The compiler derives it.

Now walk through the stencil from the opening: `u[t-1, i+1] - 2.0 * u[t-1, i] + u[t-1, i-1]` with the same declaration `u[t in 0..T, i]`. The compiler finds three reads of `u` in the body. All three are at `t-1` on dim 0 — each gives `t-1  t`, so dim 0 is recurrence. At position 1, the expressions are `i+1`, `i`, `i-1`. Two of the three differ from the loop variable `i`. So by the scan rule, dim 1 also has non-matching expressions. And yet — dim 1 is not a recurrence coordinate. Why?

Because every read is at `t-1`. The step `t-1` is fully materialized — every `i` exists. You can reach left to `i-1`, right to `i+1`. It's a bookshelf. The compiler knows this: if a read of the declared variable is from a *different* recurrence step, then only the offset that creates the cross-step dependency matters. The cross-step offset is `t-1`. The `i±1` offsets are within a completed step. The rule has two tiers:

Tier 1 — any offset on the declared variable? For every read of the declared variable, at every coordinate position: is the expression exactly the loop variable? If any read differs, the dimension is *candidate recurrence*. This catches all dependencies, including `i±1`.

Tier 2 — does the offset cross steps, or stay within one? If the read is at a different recurrence step (`t-1`), the spatial offsets from that read do not create recurrence. Only the dimension whose offset points across steps (`t-1`) is the *output recurrence dimension*. The spatial offsets on other dims are reads from a bookshelf — the previous step's full slice.

These two tiers separate *which dimensions have offsets* from *which dimension carries the dependency chain*. The first tier is a scan. The second tier is a filter. Together they produce the lowered IR's `recurrence_output_dim` — the one dimension whose history must be stored across steps. All other dimensions are concurrent within each step.

The declaration bracket names *what* is being defined and the domain of the recurrence coordinate. The body says *how*. The separation keeps the declaration side simple and declarative, while the body can use arbitrary index arithmetic:

```
let fib[0] = 0;
let fib[1] = 1;
let fib[n in 2..8] = fib[n-1] + fib[n-2];
```

The recurrence index range `n in 2..8` goes in the declaration bracket—it defines the domain. The expressions `n-1` and `n-2` go in the body—they compute the value by referencing earlier elements. Every reference must point strictly backward.

Now look at this line:

```
let h[t in 0..T] = step(h[t+1], x[t])
```

What should happen? The declaration says `t in 0..T`—the statement defines `h` at time `t`. The body references `h[t+1]`—a value at time `t+1`. At the moment `h[t]` is being computed, `h[t+1]` has not been computed yet. `t+1` is strictly greater than `t`. The rule: **every index reference to the declared variable must be strictly less than the declared index.** `t+1 < t` is false. Error.

The check does not need to know that `t` is "time." It does not need to know what "causality" means. It does exactly one thing: compare the reference index against the declared index, for every reference to the declared variable in the body. Reference index `<` declared index? Valid. Otherwise? Rejected. The coordinate can be called `t`, `x`, or `spatial_index`—the check is the same. Causality is not a name-declared property. It is subtraction.

This has a consequence that coordinates without offsets don't require. Only the steps that are actually

referenced backward need to be kept. If every step references only `t-1`, the storage needed is a rolling window of size 2, regardless of whether `T` is 100 or 100,000. This follows mechanically from the backward references—no annotation needed.

Be the Compiler

You now know what the compiler knows. For each fragment below, run the same mechanical scan. Find every read of the declared variable. Check each index expression against the loop variable. Then apply the causality rule: reference index < declared index.

Here are five fragments:

Fragment A:

```
let u[t in 0..T, i] = u[t-1, i] + u[t-2, i];
```

Fragment B:

```
let h[t in 0..T, i] = h[t-1, i] + x[t, i-1];
```

Fragment C:

```
let v[t in 0..T] = v[t+1] + v[t-1];
```

Fragment D:

```
let x[t in 0..T] = f(x[t-1]) + g(x[t]);
```

Fragment E:

```
let A[i in 0..M, j in 0..N] = A[i, j-1] + A[i-1, j];
```

Here is what the compiler decides:

Fragment A: passes. `t-1 < t` and `t-2 < t`. Both references are strictly backward. Window size: 2 (references `t-1` and `t-2`).

Fragment B: passes. `t-1 < t` — the RHS reads `h` at `t-1`, so `t` is a recurrence coordinate. But `x[t, i-1]` reads a different variable `x` — the offset `i-1` does not make `i` a recurrence coordinate because the check only applies to the variable being defined. `t-1 < t` is true. Fragment B passes.

Yes. The causality check applies to the recurrence coordinate `t`. `x[t, i-1]` is just a spatial read of an input — `x` was fully computed before this clause runs. `i-1` is a spatial offset, not a recurrence. Fragment B passes.

But notice: `x[t, i-1]` means the computation at position `i` reads position `i-1` from `x` at the same time `t`. If the spatial iteration goes left to right, this is fine—`x[i-1]` is already available. If the spatial iteration goes right to left, `x[i-1]` hasn't been loaded yet. The compiler doesn't check spatial order by default because spatial coordinates don't carry a direction constraint. If you want spatial causality, you declare the coordinate with an offset on the declared variable.

Fragment C: REJECTED. `t+1 > t`. This is a forward reference on the recurrence coordinate. The body references a value at `t+1`, which hasn't been computed yet (assuming forward iteration). Error.

Fragment D: REJECTED. `x[t]` references the same step being defined. `t < t` is false. A value at time `t` cannot depend on itself—that would be a circular definition. The reference index must be strictly less than the declared index. `t < t` is not strict.

Fragment E: passes, with both dims candidate recurrence. `A[i, j-1]` gives an offset at dim 1 — `j-1  j`. `A[i-1, j]` gives an offset at dim 0 — `i-1  i`. Both dims have offsets on the declared variable `A`. Tier 1 marks both. Tier 2 asks: which offset creates the cross-step dependency? `j-1` is satisfied by iteration order within the same `i` step — walk left to right, and `j-1` is already computed. `i-1` requires the previous row — the full `j` slice from the previous `i`. Only `i` is the output recurrence dim. The causality check: `i-1 < i` and `j-1 < j` — both are backward, both pass.

Five fragments, one rule: reference index `<` declared index. The spatial offset in Fragment B doesn't trigger the rule because the offset is on a different variable — `x[i-1]`, not `h[i-1]`. In Fragment E, both dims have offsets on the declared variable — but only one is the recurrence output dim, because only one offset creates a cross-step dependency. Only offsets on the declared variable create recurrence. The rule is simple. The check is mechanical.

The Optimizer as a Recurrence

Training a model is a recurrence over time:

```
let w[t in 0..T, out, feature] = init_random(out, feature);
let w[t in 1..T, out, feature] = w[t-1, out, feature] - lr * grad[t-1, out, feature];
```

At `t=0`, `w` is the random initialization. At each subsequent step, `w` is the previous `w` minus a gradient step. The recurrence reads backward in time (`t-1`). The time coordinate `t` makes the training trajectory explicit. You can inspect `w[10, out, in]` to see the weights after 10 steps. You can compute `w[T-1, out, in] - w[0, out, in]` to see the total change. The time dimension is not hidden inside a mutable variable—it is a coordinate like any other.

A full training step:

```
let logits[t, b, class] = model(x[t, b, feature], w[t, out, feature]);
let loss[t, b] = cross_entropy[class](logits[t, b, class], labels[t, b]);
let grad[t, out, feature] = @loss[t, b] / @w[t, out, feature];
let w[t+1, out, feature] = w[t, out, feature] - lr * grad[t, out, feature];
```

The time coordinate `t` threads through forward, loss, gradient, and update. Every tensor knows its temporal position. The gradient `@loss[t] / @w[t, out, feature]` is explicitly anchored to time step `t`. The optimizer step defines `w[t+1]` in terms of `w[t]` and `grad[t]`.

There is a quieter distinction at work here that deserves explicit attention: **parameters versus hyperparameters.** Both are `let` bindings. Both are immutable values in scope for subsequent code. But they have different roles in the optimization story:

```
let weight: [f32; out, feature] = init_random(out, feature);
let learning_rate: f32 = 0.001;
```

`weight` is a parameter—its value changes during training, driven by gradients. Each coordinate on `weight` tells the optimizer something: `out` names the output neurons, `in` names the input connections. A weight decay regularizer that treats all elements uniformly can apply without coordinate awareness, but a per-neuron regularization policy needs to know which axis is `out` and which is `in`.

`learning_rate` is a hyperparameter—it controls the optimizer's behavior but is not itself updated by gradients. It carries no coordinate names because it has no coordinate structure.

This distinction is not a language feature. It is a naming discipline. But the discipline is only possible because the language provides a place to put the coordinate names. Without named axes, the optimizer sees `(128, 64)` and cannot distinguish `out` from `in`.

An Axis with an Offset Has a Direction

A coordinate that only appears as itself has no dependencies — all positions exist concurrently. A coordinate that appears with an offset on the RHS has a dependency. `t-1` means `t` depends on the previous step. Not concurrency. Dependency.

This distinction has consequences. A coordinate with an offset carries two properties that a coordinate without one doesn't:

1. **Causality**: every offset reference to the declared variable must be strictly less than the declared index. `t-1` is valid. `t+1` is a compile error.
2. **Memory**: only the steps that are actually referenced backward need to be kept in memory. If every step references only `t-1`, the storage needed is a rolling window of size 2, regardless of whether T is 100 or 100,000. This follows mechanically from the backward references—no annotation needed.

Bidirectional Recurrence

Not all recurrences look only to the past. A bidirectional RNN reads the sequence both ways:

```
let h_forward[t in 1..T, i] = step(h_forward[t-1, i], x[t, i]);
let h_backward[t in 0..T-1, i] = step_back(h_backward[t+1, i], x[t, i]);
```

The forward recurrence reads `t-1`—standard. The backward recurrence reads `t+1`—the future from the perspective of `t`. This is still valid because the backward recurrence iterates from right to left: `t` runs from `T-1` down to `0`, so `t+1` is always already computed. The direction of iteration determines which references are "backward."

The same coordinate domain, two different iteration directions, one linguistic mechanism. The declaration bracket names the domain and direction. The body states the dependency. Consistency is checked.

Notice when someone writes this:

```
let h[t in 0..T, i] = step(h[t+1, i], x[t, i]);
```

No iteration direction declared. Just `t in 0..T`. The body references `t+1`. Is this valid or not?

It depends on whether the compiler infers the iteration direction from the reference pattern. If `t+1` references a time step that hasn't been computed yet (because iteration goes left to right), this is a forward reference and should be rejected. But if the compiler can infer that `t` should iterate from `T` down to `0`—making `t+1` already computed—it could be valid.

The Einlang rule is conservative: without an explicit reverse-direction declaration, forward references are rejected. `t+1` with `t in 0..T` is an error. The programmer must write `t in T..0` (or equivalent syntax) to declare the reverse iteration. The tool prevents the ambiguous case by default.

This is the same design choice as Chapter 3's Coordinate Contract and Chapter 5's Pack Resolution: when a reference pattern is ambiguous, the language requires the programmer to disambiguate. Default deny. Explicit allow.

The Rolling Window: What Causality Buys

Causality is not just a correctness check. It is a memory optimization.

When a recurrence body only references `t-1`, the compiler knows that only one previous time step is needed. It can allocate a rolling window of size 2 rather than storing the entire `(T, ...)` tensor in memory. When `T` is 100,000, this is the difference between allocating gigabytes and allocating megabytes.

This optimization follows mechanically from the backward references. The compiler scans the body for time-indexed references. Every reference to `t - k` (positive `k`) requires storing `k` previous steps. The rolling window size is `max(k)`. No annotation needed. The coordinate names and index arithmetic carry enough information for the compiler to derive the memory plan.

The same principle—coordinate set subtraction—is at work. The output coordinate set includes `t`. The body references `t - k`. The difference `t - (t - k) = k` tells the compiler how many previous steps to store. Set subtraction, introduced in Chapter 2 for broadcast detection, applied here to memory planning. The operation is the same. The application is different.

Consider a second-order recurrence:

```
let u[t in 2..T, i] = u[t-1, i] + 0.5 * (u[t-1, i] - u[t-2, i]);
```

References: `t-1` and `t-2`. Maximum offset: 2. Rolling window size: 3 (current, t-1, t-2). The compiler derives this from the index expressions. No `@roll_window(3)` annotation. The information is in the code, not in a compiler directive.

The programmer writes `t-2`. The compiler derives window size 3. The programmer writes `sum[class]`. The compiler derives `axis=1`. The programmer writes `bias[j]` omitting `i`. The compiler derives the backward-pass sum over `i`. Source records intent. Compiler derives execution.

Three Declarations, Three Storage Schemes

The same recurrence declaration leads to different storage strategies depending on what the programmer references. The compiler reads the references and derives the strategy. Three scenarios:

Scenario 1: Rolling window. `u[t] = f(u[t-1])`. References: `t-1` only. Window size: 2 (one past step plus current). Storage: two arrays. The compiler allocates `u_prev` and `u_curr`, swaps them after each step. No allocation proportional to `T`.

Scenario 2: Full materialization. `u[t] = f(u[t-1])` followed by `mean[t](u[t, i])`. The entire time trajectory is needed for the final mean over `t`. The compiler sees that `t` is consumed by a reduction after the recurrence, so all time steps must be kept. Storage: full `(T, ...)` tensor.

Scenario 3: Strided observation. `u[t] = f(u[t-1])` followed by `u[0], u[10], u[20], ...` (every 10th step). The compiler sees that only `u[k*10]` is used downstream. Storage: rolling window of size 10, with the current and last 9 steps buffered. At each multiple of 10, the current value is written to persistent storage and the buffer recycles.

In all three scenarios, the declaration is the same: `let u[t in 0..T, i] = f(u[t-1, i])`. The difference is in what downstream code does with `u`. The compiler reads the downstream uses and derives the storage strategy. No annotation. The structural fact is in the code. The compiler derives the engineering consequence.

The recurrence body records the dependency — the compiler found it by scanning for offsets. The downstream uses record the observation pattern. The compiler connects them. One scan, two purposes: correctness (causality check) and memory (window size). Both from the same minus sign.

The compiler reads the code the way a reader reads a story: forward to understand what each step depends on, backward to determine what must be kept. The same declaration can compile to two arrays or a full trajectory tensor. The difference is not in the declaration. It is in what the code goes on to demand.

From Recurrence Dims to Execution Strategy

The two-tier scan you ran in Fragment E does more than classify dimensions. It picks the execution strategy. But before reading the answer, ask yourself the question the compiler faces:

Fragment E has two recurrence dims — `i` and `j`. Neither alone satisfies Tier 2. The heat equation has one recurrence dim (`t`), and it does satisfy Tier 2. Should these two cases compile to the same loop structure? Or different ones?

Take a minute. What would *you* emit?

The heat equation (`u[t-1, i]`, `u[t-1, i±1]`) reads only `t-1` on the declared variable. Tier 2 is satisfied on `t`. The compiler picks `t` as the **recurrence output dim** — step `t` forward one at a time, compute all `i` positions in parallel at each step. It allocates a rolling history buffer (two rows for lookback, plus tail steps for downstream reads), iterates the `t` loop, and runs the spatial computation as a single tensor operation per step. Fibonacci, the optimizer recurrence, and the bidirectional RNN all use this path. One recurrence output dim, vectorized spatial.

Fragment E (`A[i,j] = A[i-1, j] + A[i, j-1]`) has no dim that satisfies Tier 2. `A[i-1, j]` is backward on dim 0, but `A[i, j-1]` is not — on dim 0, `i` appears without offset. Dim 1 has the symmetric problem: `j-1` is backward, but `A[i-1, j]` uses `j` without offset. Neither dim earns the strict-backward guarantee. The compiler falls back to **partition/step**: nested loops over `i` and `j`, compute one position at a time, buffer the full previous row.

The same recurrence detection that checks causality also picks the execution path. Tier 1 alone → partition/step. Tier 2 satisfied → vectorized with rolling window. The compiler doesn't guess. It reads the offsets.

The programmer writes `i-1`. The compiler determines whether that minus sign means "store one row" or "iterate one position at a time." No annotation. The minus sign is the annotation.

Time in the Training Loop

The optimizer recurrence from earlier is worth tracing step by step:

```
// Step 0: random initialization
let w[0, out, feature] = init_random(out, feature);

// Step 1: forward pass
let logits[1, b, class] = model(x[1, b, feature], w[0, out, feature]);
let loss[1, b] = cross_entropy[class](logits[1, b, class], labels[1, b]);

// Step 1: backward pass
let grad[1, out, feature] = @loss[1, b] / @w[0, out, feature];

// Step 1: update
let w[1, out, feature] = w[0, out, feature] - lr * grad[1, out, feature];

// Step 2: forward pass
let logits[2, b, class] = model(x[2, b, feature], w[1, out, feature]);
// ... and so on
```

At each time step, three things happen: forward (model produces output), backward (gradient is computed), update (weights move against the gradient). The time index `t` is explicit on every tensor. You can read the value of `w` after any step. You can read the loss at any step. The training trajectory is a tensor, not a sequence of in-place mutations.

Now compare to the PyTorch equivalent:

```
w = init_random(out, in)
for t in range(1, T):
    logits = model(x[t], w)
    loss = cross_entropy(logits, labels[t])
    loss.backward()
    with torch.no_grad():
        w -= lr * w.grad
```

`w` is a single mutable tensor. `loss` is a scalar. The time dimension is the loop variable `t`—visible in the Python control flow but absent from the tensor structure. You cannot inspect `w[10]` without checkpointing the value at step 10 yourself. The training trajectory exists in execution time, not in the type system.

The Einlang version makes the training trajectory a data structure. The PyTorch version makes it a side effect. The difference is whether you can query the past.

Diffusion Models

Diffusion models add noise over `T` timesteps and learn to reverse it. The time coordinate appears in two roles: recurrence index for the sampling chain, and conditioning signal for the denoising network.

```
let x[t in 0..T, b, c, h, w] = ...;
let x[t in 1..T, b, c, h, w] = sqrt(1 - beta[t]) * x[t-1, ...] + sqrt(beta[t]) * eps[t, ...];
```

The schedule `beta[t]` is indexed by `t`, making the dependency visible. In the backward pass:

```
let x_hat[t in T..1, b, c, h, w] = denoise(x[t, ...], t, model(x[t, ...], t));
```

The iteration runs backward. The model receives `t` as conditioning. This is the same mechanism that carried `class` through `softmax[class]` in Chapter 3, applied to a coordinate with an offset. The direction—forward or backward—is the only difference.

A coordinate with an offset carries a direction constraint. The constraint is checked. The coordinate flows through functions. The training loop is a recurrence. The diffusion process is a recurrence. The optimizer is a recurrence. Three domains, one mechanism.

Return to the Recurrence: Kalman Filter

You've seen recurrences over scalar fields (the heat equation), over parameters (the optimizer), and over noisy samples (diffusion). Here is a recurrence over matrix-valued state — a Kalman filter tracking position and velocity from noisy measurements:

```
let dt = 0.1;
let F = [[1.0, dt], [0.0, 1.0]];       // state transition
let H = [1.0, 0.0];                     // observation matrix
let Q = [[0.01, 0.0], [0.0, 0.1]];      // process noise
let R = 1.0;                            // measurement noise

// State: x[t, i] where i in 0..2 (position, velocity)
// Covariance: P[t, i, j] where i, j in 0..2
let x[0, i in 0..2] = [0.0, 1.0][i];
let P[0, i in 0..2, j in 0..2] = [[1.0, 0.0], [0.0, 1.0]][i, j];

// Predict
let x_pred[t in 1..T, i in 0..2] =
    F[i, 0] * x[t-1, 0] + F[i, 1] * x[t-1, 1];
let P_pred[t in 1..T, i in 0..2, j in 0..2] =
    sum[k in 0..2](F[i, k] * sum[l in 0..2](P[t-1, k, l] * F[j, l])) + Q[i, j];

// Update
let y[t in 1..T] = z[t-1] - (H[0] * x_pred[t, 0] + H[1] * x_pred[t, 1]);
let S[t in 1..T] = H[0] * P_pred[t, 0, 0] * H[0] + R;
let K[t in 1..T, i in 0..2] = (P_pred[t, i, 0] * H[0]) / S[t];
let x[t in 1..T, i in 0..2] = x_pred[t, i] + K[t, i] * y[t];
```

Before reading on, answer two questions. First: how many times does `t-1` appear in this code, and on which variables? Second: `i` ranges over the state (position, velocity) and `j` ranges over the same domain in the covariance. Are `i` and `j` recurrence coords or bookshelves? How do you know?

`t-1` appears four times — on `x`, twice on `P`, and on `P_pred`. Four backward references. All at the same time step. The dependency chain flows through a 2×2 covariance matrix, not a scalar. Same mechanism as the heat equation. Same check. Only `t` carries the arrow.

`i` and `j` are bookshelves. Every position at `t-1` is materialized — you can read `i=0` or `i=1`, `j=0` or `j=1`, in any order. The minus sign on `t-1` is the only signal that matters. Matrix-valued state doesn't change the rule.

In PyTorch, the shapes are `(T, 2)` and `(T, 2, 2)`. Position 0 is probably time. But "probably" is not

a check. Here, `t-1` is the check.

The Gradient of a Recurrence

Recurrences have gradients. And because recurrences are self-referential—each step depends on the previous step—the gradient must flow backward through time. This is Backpropagation Through Time (BPTT), and its coordinate structure is the same recurrence, read in reverse.

Forward: `h[t] = step(h[t-1], x[t])`. The output `h[t]` depends on `h[t-1]`, which depends on `h[t-2]`, and so on back to `h[0]`.

Backward: the gradient `d_loss/d_h[t]` must propagate to `d_loss/d_h[t-1]`, then to `d_loss/d_h[t-2]`, and so on. At each step, the gradient flows through the `step` function's Jacobian with respect to `h[t-1]`. The backward recurrence:

```
let d_h[t in T..0] = @loss[t] / @h[t] + @step(h[t], h[t-1], x[t]) / @h[t] * d_h[t+1];
```

The backward recurrence runs from `T` down to `0`, referencing `t+1` (the future in the backward direction, which has already been computed). This is the same bidirectional mechanism from Section 6, applied to the gradient. The coordinate `t` still carries the causality constraint, but the iteration direction has reversed.

In Einlang, the backward recurrence is generated from the forward recurrence by the same Inversion Rule that governs reductions and broadcasts: `t in 1..T` forward becomes `t in T..0` backward. The coordinate names stay the same; the compiler generates the backward loop from the forward declaration.

Time was one coordinate with a direction. Next: terrain where one coordinate splits into two roles—`point` becomes `point_i` and `point_j` in a distance matrix, `sample` becomes `anchor` and `positive` in contrastive learning. Convolution adds index arithmetic (`oh + kh`). The split is the operation. The names record it.

Chapter 7 · Complex Terrain

> "The shortest path between two truths in the real domain passes through the complex domain."
>
> — Jacques Hadamard

Combinations · Distance matrices, convolution, and fancy indexing

You have a distance matrix. Shape `(N, N)`. Axis 0 and axis 1 both index over the same set of points. Which one is the source point, and which is the target? The shape does not tell you. The positions do not tell you. If you wrote a comment, the comment tells you—for now, until the comment rots.

The matrix is square because the same coordinate appears twice, playing two different roles. The coordinate has split. `point` becomes `point_i` and `point_j`. The split is invisible to every positional tool you have. It is visible to the reader only if the notation records which copy is which.

This chapter is about the split—one phenomenon seen through several lenses. Distance matrices, where one coordinate turns into two. Convolution index arithmetic, where coordinates carry formulas (`oh + kh`). Fancy indexing, where coordinate collisions are bugs or features depending on intent. Every section asks the same question: when a coordinate's role is more complex than "this axis exists," does your notation record the complexity—or bury it in shape arithmetic?

Distance Matrix: When One Coordinate Becomes Two

Given a set of points, compute the pairwise distance between every pair. The same coordinate—`point`—appears twice, playing two roles: once as the source, once as the target.

```
let dist[point_i, point_j] =
    sum[dim]( (points[point_i, dim] - points[point_j, dim]) ** 2.0 ) ** 0.5;
```

`point` has been split into `point_i` and `point_j`. Both index the same underlying domain—the set of points. But `point_i` walks the rows of the distance matrix and `point_j` walks the columns. The naming makes the split visible. That both index into the same coordinate domain is checked. The reader sees that `point` was duplicated, not two unrelated coordinates that happen to share a prefix.

In a positional framework, this split is invisible:

```
dist = ((points[:, None, :] - points[None, :, :]) ** 2).sum(-1) ** 0.5
```

`None` inserts a dimension. `:` slices everything. Which dimension is `point_i` and which is `point_j`? You have to count positions. If `points` changes from `(N, D)` to `(D, N)`, the positional code silently computes the wrong matrix.

The coordinate-split pattern appears in many domains. Once you recognize it, you see it everywhere:

Graph neural networks. `source_node` and `target_node` split from the same node domain. Edges connect `source_node` to `target_node`. The adjacency matrix has `(source_node, target_node)` coordinates. A message-passing step gathers from `target_node` and scatters to `source_node`. In positional code, both are axis 0 and axis 1. In named code, they carry different identities even when they index the same domain.

Contrastive learning. `anchor` and `positive` index the same set of samples. The similarity matrix has `(anchor, positive)` coordinates. The loss pulls `anchor[i]` close to `positive[i]` (pairwise) and pushes

`anchor[i]` away from `positive[j]` for `i != j`. The coordinate split `anchor`/`positive` distinguishes "same sample, different view" from "different sample." In positional code, the distinction is in the mask tensor, not in the coordinate structure.

Collaborative filtering. `user` and `item` share a latent factor `k` through a matrix factorization: `ratings[user, item] = sum[k](U[user, k] * V[item, k])`. The inner dimension `k` is shared and consumed. `user` and `item` are different coordinates indexing different domains. In positional code, `U @ V.T`—all three identities (`user`, `item`, `k`) collapsed into positions.

The names record what was split. The positional code records only the mechanic of `unsqueeze`.

The split was spatial—one coordinate name, two roles, same domain. Next: what happens when coordinates carry arithmetic, and a coordinate from the output combines with a coordinate from the kernel to produce an index?

Convolution: Coordinates with Arithmetic

A convolution is a sum of products with index arithmetic:

```
let conv[b, oc, oh, ow] = sum[ic, kh, kw](
    input[b, ic, oh + kh, ow + kw] * weight[oc, ic, kh, kw]
);
```

The novelty is `oh + kh` and `ow + kw`. These are index expressions—arithmetic on coordinate variables. `oh + kh` says: to read from the input at spatial position `oh + kh`, take the output position `oh` and add the kernel offset `kh`. Bounds of this arithmetic are not verified (that's a runtime check), but every coordinate in the index expression is verified to be in scope and to have a known domain.

Notice where the expressions live: in the body index positions, not in the declaration bracket. The declaration bracket names the output coordinates `b, oc, oh, ow`—simple identifiers. The body index expressions `oh + kh` describe how to compute the input indices from the output indices and the kernel coordinates. Left side: definition. Right side: computation.

The gradient with respect to `weight` follows mechanically from the coordinate sets:

```
let dW[oc, ic, kh, kw] = sum[b, oh, ow](
    dConv[b, oc, oh, ow] * input[b, ic, oh + kh, ow + kw]
);
```

The coordinates `b`, `oh`, `ow` are summed away because they appear in the output but not in `weight`. The surviving coordinates `oc`, `ic`, `kh`, `kw` are exactly `weight`'s coordinates. Set subtraction, applied to coordinate names, derives the formula. No memorization of transpose rules needed.

Depth-to-Space: One Line Instead of Three

A depth-to-space operation reshapes a tensor by moving pixels from the channel dimension into the spatial dimensions, increasing resolution. In a positional framework:

```
b, c, h, w = x.shape
x = x.reshape(b, c // 4, 2, 2, h, w)
x = x.permute(0, 1, 4, 2, 5, 3)
x = x.reshape(b, c // 4, h * 2, w * 2)
```

Three operations. Six position numbers to track. The semantic claim—"channel pixels become spatial neighbors"—is invisible.

In Einlang:

```
let y[b, c_out, h * 2 + dy, w * 2 + dx] =
    x[b, c_out * 4 + dy * 2 + dx, h, w];
```

One line. The index arithmetic says exactly what moved where. The names `dy` and `dx` declare the sub-pixel offsets. The multiplication `h * 2 + dy` expresses the output spatial coordinate in terms of the input. The coordinate names carry the story.

Index arithmetic reshapes coordinates. Next: when coordinates determine *which* elements you read from a tensor, and the difference between "read together" and "read independently" depends entirely on whether the coordinate name is shared.

Fancy Indexing: Names Disambiguate

Fancy indexing—using arrays of indices rather than slices—is where positional notation becomes most ambiguous. Consider gathering elements from a matrix:

```
# Positional: what does this gather?
result = matrix[indices_i, indices_j]
```

In NumPy, this does *pairwise* indexing: `result[k] = matrix[indices_i[k], indices_j[k]]`. But if `indices_i` is a row vector and `indices_j` is a column vector, broadcasting produces *outer-product* indexing instead. The behavior depends on the shapes of the index arrays, not on anything written in the code.

In Einlang, the coordinate names disambiguate:

```
// Pairwise gather: same coordinate name appears in both index positions
let gathered[k] = matrix[indices_i[k], indices_j[k]];

// Outer-product gather: different coordinate names → Cartesian product
let outer[i, j] = matrix[indices_i[i], indices_j[j]];
```

When `k` appears in both index positions, pairwise indexing is inferred: `indices_i` and `indices_j` are traversed together. When `i` and `j` are different coordinates, outer-product indexing is inferred: every combination is produced. The distinction is visible in the coordinate names, not hidden in the shapes of the index arrays.

In NumPy, you need `np.ix_` or NEP 21's `oindex`/`vindex` to explicitly declare which behavior you want. In Einlang, the names do it.

Consider the indexing distinction in NumPy. Suppose you have two index arrays: `i = np.array([0, 1, 2])` and `j = np.array([0, 1, 3])`, and a 2D array `A`. To get elements at `A[0,0]`, `A[1,1]`, `A[2,3]` (pairwise), you write `A[i, j]`. To get all 3×3 combinations (outer-product), you write `A[i[:, None], j[None, :]]`. Compare the two expressions. Which one required you to create a dummy axis? The outer-product version did—you had to broadcast `i` and `j` into mutually orthogonal shapes with `None`/`np.newaxis`.

The pattern is visible: NumPy uses shape manipulation to disambiguate pairwise from outer-product indexing. The coordinate-sharing approach does it with names. Coordinate-sharing (`k` in both index po-

sitions) means pairwise; different names mean outer-product. The coordinate name carries the distinction. No shape manipulation needed.

When One Coordinate Becomes Two: Gather vs. Scatter

Fancy indexing isn't the only place where coordinates split. Consider gathering rows from a matrix by index:

```
# Positional: gather rows 0, 3, 2 from a matrix
rows = matrix[[0, 3, 2], :]
```

The index list `[0, 3, 2]` selects specific positions along axis 0. But what if you also want to select specific columns? And what if those column indices depend on the row?

```
# Positional: for each selected row, pick a different column
idx = np.array([0, 3, 2])   # row indices
col_idx = np.array([1, 0, 2])  # column index for each row
result = matrix[idx, col_idx]  # pairwise: result[k] = matrix[idx[k], col_idx[k]]
```

This is pairwise indexing—`k` walks through both index arrays together. But what if you wanted outer-product: every combination of `idx` and `col_idx`? You'd need:

```
result = matrix[idx[:, None], col_idx[None, :]]  # outer-product via broadcast
```

Two different behaviors, two different indexing patterns, one API. The difference is in the shapes of the index arrays (1D vs 2D after broadcasting), not in any semantic marker. If `idx` and `col_idx` happen to have the same length, the pairwise version runs and produces a result—just not the one you wanted if you intended outer-product.

Einlang:

```
// Pairwise: same coordinate name in both index positions
let gathered[k] = matrix[idx[k], col_idx[k]];

// Outer-product: different coordinate names → Cartesian product
let gathered[i, j] = matrix[idx[i], col_idx[j]];
```

When `k` appears in both index positions, the compiler infers pairwise indexing. When `i` and `j` are different, outer-product is inferred. The coordinate name *is* the disambiguation. No shape-dependent behavior. No manual broadcasting.

The named notation records the mental model ("these index together" vs "these index independently"). The positional notation buries it in shapes.

The Coordinate Collision Test

There is a simple test for whether your notation disambiguates well. Consider two operations that have the same shape but different coordinate semantics. Can a colleague tell which is which just by looking at the code?

For fancy indexing:

```
# Operation A
result = matrix[idx, col_idx]

# Operation B
result = matrix[idx[:, None], col_idx[None, :]]
```

If `idx` has length 5 and `col_idx` has length 5, both operations produce a `(5, 5)` result. But Operation A produces the diagonal (pairwise), while Operation B produces the full Cartesian product. From the code alone—without printing shapes or reading comments—can your colleague tell which is which?

Now the Einlang versions:

```
// Operation A: pairwise
let result[k] = matrix[idx[k], col_idx[k]];

// Operation B: outer-product
let result[i, j] = matrix[idx[i], col_idx[j]];
```

The difference is in the coordinate names. `k` vs `(i, j)`. One coordinate means pairwise. Two means outer-product. Your colleague reads the code and sees the difference. The code records the intent.

This is the Coordinate Collision Test: when two operations produce the same shape but different semantics, does your notation distinguish them? If the answer is no, the semantics are in your head. If the answer is yes, they are in the code. The distance between those two places is the distance between a bug found in review and a bug found in production.

Derive it yourself: Collision-proof your notation. Here are three index-gathering operations. For each one, write both the pairwise and outer-product versions by choosing coordinates. Then check: would a reader see the difference?

1. A lookup table `L` maps `(token, feature)` pairs to values. Gather from it using a list of indices. Write the pairwise version (index `k` selects tokens and features together) and the outer-product version (indices `i` and `j` select tokens and features independently).
2. A routing operation sends each item `item[i]` to a destination `dest[j]` using a routing table `route`. Write the pairwise version and the outer-product version.
3. A graph convolution gathers features from neighbors: `adj[node, neighbor]` selects which neighbors to aggregate. Write the pairwise version (each `edge` selects a specific `(node, neighbor)` pair) and the outer-product version.

The answer in each case: one coordinate means pairwise. Two means outer-product. The names carry the semantics. The shape alone—`(N,)` in both cases—carries none.

Every section so far has been about the forward pass. Now: what happens when differentiation runs backward through these same patterns? Index arithmetic in the forward pass becomes inverted index arithmetic in the backward pass. Coordinate splits in the forward pass determine which coordinates get summed in the gradient. The names don't change. The direction does.

Convolution Backward: Gradient as Index Arithmetic

The input gradient—the one that backpropagates through the network—shows how index arithmetic survives differentiation.

Forward: `conv[b, oc, oh, ow] = sum[ic, kh, kw](input[b, ic, oh + kh, ow + kw] * weight[oc, ic, kh, kw])`.

To find `d_input[b, ic, ih, iw]`: hold one input cell, list every output cell that reads it, invert the index relationship, sum over the path coordinates.

1. **Hold one cell** of `input`: `input[b0, ic0, ih0, iw0]`.
2. **List every output cell that reads it.** The held cell is read by every `conv[b0, oc, oh, ow]` where `oh + kh = ih0` and `ow + kw = iw0`, for all `oc`, all `kh`, all `kw`. That means `oh = ih0 - kh` and `ow = iw0 - kw`. For each `(kh, kw)`, the output at position `(oh, ow) = (ih0 - kh, iw0 - kw)` receives a contribution.
3. **Attach the incoming gradient.** Each output cell `conv[b0, oc, oh, ow]` carries `d_conv[b0, oc, oh, ow]`. The contribution through the held input cell is `d_conv[b0, oc, ih0 - kh, iw0 - kw] * weight[oc, ic0, kh, kw]`.
4. **Multiply by the local derivative.** The derivative of `input * weight` with respect to `input` is `weight`.
5. **Sum over the path coordinates.** The output has `{b, oc, oh, ow}`. The input has `{b, ic, ih, iw}`. The path coordinates are `{oc, kh, kw}`—they appear in the output (via `oh, ow`) but are absorbed into `ih, iw` through the index arithmetic. But `oh, ow` are not independent—they are coupled to `ih, iw` through `kh, kw`. The sum is over `{oc, kh, kw}`, with the index relationship inverted: `oh → ih - kh`, `ow → iw - kw`.

Result:

```
let d_input[b, ic, ih, iw] = sum[oc, kh, kw](
    d_conv[b, oc, ih - kh, iw - kw] * weight[oc, ic, kh, kw]
);
```

This is the convolution transpose—a transposed convolution with flipped kernel indices. You derived it without memorizing "the gradient of convolution is a transposed convolution." You did coordinate accounting: which coordinates is the output indexed by that the input is not? The output uses `oh, ow` where the input uses `ih, iw`. The kernel indices `kh, kw` are shared—they appear in both the forward and the gradient. The output channel `oc` is in the output but not the input—sum over it.

The index arithmetic `ih - kh` and `iw - kw` comes from inverting the forward relationship `oh + kh → ih`. The gradient "reads" from the output at the position where the input contributed. The inversion is mechanical: solve `oh + kh = ih` for `oh`, giving `oh = ih - kh`.

Step back. You just derived a transposed convolution—the gradient of convolution—without memorizing a formula. The steps were: hold one input cell, list every output cell that reads it, invert the index relationship, sum over the path coordinates. The names `ih`, `iw`, `kh`, `kw`, `oc` stayed constant. The operation changed: forward addition (`oh + kh`) became backward subtraction (`ih - kh`). Same coordinate accounting, different arithmetic.

This is the central claim of Part II: coordinate names survive composition. They survive function boundaries (Chapter 3), broadcast (Chapter 4), normalization skeletons (Chapter 5), recurrence (Chapter 6), complex index arithmetic (this chapter), and differentiation (Chapter 8). The name is the invariant. The operation around it changes. The name remains.

The Boundary

Index arithmetic like `oh + kh` is syntactically checked—`oh` and `kh` must be in scope. Whether `oh + kh < input_height` can be proved depends on what the compiler knows.

When the domain sizes are known at compile time—`oh` ranges over `0..output_height`, `kh` over `0..kernel_height`, both static—the constraint solver can prove the bound. `(output_height - 1) + (kernel_height - 1) < input_height` is a single comparison. The proof is mechanical. Chapters 9 and 10 show how.

When the domain sizes are runtime values—image dimensions read from a file, sequence lengths that vary per batch—the compiler cannot prove the bound. The names are still there. The error message names the coordinate: `IndexError: oh + kh = 67 exceeds input width 64`. The positional equivalent names a position: `IndexError: dimension 3 out of bounds`. The difference is the distance between the two.

What names can and cannot check is the subject of Chapter 14. For now: the compiler checks every fact derivable from declarations and proves every bound that static ranges permit. What it cannot prove—runtime-dependent bounds, data-dependent shapes, semantic correctness—it leaves visible, with names attached.

When Index Arithmetic Meets Gradients

The general pattern: index arithmetic in the forward pass produces inverted index arithmetic in the backward pass. Forward `input[..., oh + kh, ...]` becomes backward `d_conv[..., ih - kh, ...]`. The subtraction inverts the addition. The coordinates involved in the arithmetic—`kh`, `kw`—become reduction axes in the backward sum. The coordinates that were loop axes—`oh`, `ow`—become the path coordinates that get summed.

In a positional autodiff engine like PyTorch's, this inversion is computed by the engine's internal graph. The programmer never writes `ih - kh`. The engine traces the forward operations, records the index arithmetic as node dependencies, and derives the backward computation automatically. The programmer writes `conv2d(input, weight)` and the gradient is handled by autograd.

The difference is not correctness—both derive the same result. The difference is auditability. When the backward pass is written explicitly with coordinate names, a reader can trace `ih - kh` back to the forward `oh + kh` and verify that the inversion is correct. When the backward pass is generated by the autodiff engine, the inversion is a black box. It is correct until it isn't—and when it isn't (because a custom kernel was written with the wrong index arithmetic, or because a `@tf.custom_gradient` rule has a bug), the reader has no source-level path from the forward expression to the backward expression.

Named coordinates don't replace autodiff. They give autodiff's output a form that a human can read, verify, and debug. The index arithmetic is in the source. The coordinate names tell you what is being inverted. The reader can trace the thread.

Ranges Are Expressions

Every reduction so far has used a fixed range—`sum[i](data[i])` sums over all of `i`. But the range can be an expression. And the expression can reference another coordinate:

```
let cumsum[i] = sum[k in 0..i+1](data[k]);
```

The coordinate `i` appears in two places: on the left-hand side (the output index) and in the reduction range `k in 0..i+1`. Each output position `cumsum[0]`, `cumsum[1]`, `cumsum[2]` sums a different prefix of `data`.

This is not a primitive `scan` operation. It is a reduction. The only difference from `sum[k](data[k])` is that the range depends on `i`. The coordinate `i` does double duty—it tells you which output cell you are computing, and it sets the boundary for how many inputs that cell sees.

The same pattern produces every cumulative operation:

```
let cumprod[i] = prod[k in 0..i+1](data[k]);      // cumulative product
let cummax[i] = max[k in 0..i+1](data[k]);         // cumulative maximum
let running_avg[i] = sum[k in 0..i+1](data[k]) / (i + 1) as f32;  // running average
```

And it composes with other coordinates:

```
// Cumulative energy along time, per batch element
let energy[b, t] = sum[k in 0..t+1](signal[b, k] * signal[b, k]);
```

`b` is a survivor—it passes through untouched. `t` is the scan axis—it sets the range for `k`. `k` is consumed by the sum. Three coordinates, three roles, one line.

In a positional framework, `np.cumsum(data, axis=0)` gives you the cumulative sum. But `axis=0` tells you nothing about which coordinate is being scanned. If the layout changes—a temporal dimension is inserted, or the batch and time axes swap—`axis=0` silently scans the wrong coordinate. The named version says `cumsum[t]`. The scan is over `t`, regardless of where `t` sits in the layout.

Ranges are expressions. Expressions can reference coordinates. Coordinates can appear in both the output declaration and the reduction range. The three roles—output index, range parameter, consumed variable—are all visible in the notation, all carried by names.

Ranges that Shape a Neighborhood

The ranges so far have been one-sided: `k in 0..i+1` says "stop at the current position." But ranges can also exclude boundaries, and—combined with offset index expressions—they can define a spatial neighborhood that moves across a grid.

Consider the 1D heat equation, discretized on a grid of 41 spatial points evolved over 80 time steps. The continuous equation is $\partial u/\partial t = \partial^2 u/\partial x^2$. The discrete version uses a three-point centered difference:

```
let r = 0.2;
let n = 41;

let u[0, i in 0..41] = 0.0;
let u[0, 20] = 10.0;                              // initial bump

let u[t in 1..80, 0] = 0.0;                       // left boundary - fixed
let u[t in 1..80, 40] = 0.0;                      // right boundary - fixed
let u[t in 1..80, i in 1..40] =                   // interior - stencil
    u[t-1, i] + r * (u[t-1, i-1] - 2.0 * u[t-1, i] + u[t-1, i+1]);
```

Read the interior line: at time `t` and spatial position `i`, the new value is the old value plus a diffusion term that reads three neighbors from the previous time step — `i-1`, `i`, and `i+1`. The range `i in 1..40`

excludes the boundaries (0 and 40 are handled separately). The range `t in 1..80` starts at 1 because the initial condition occupies `t=0`.

The range expressions on the left side declare *which cells are computed*. The offsets on the right side declare *which already-computed cells are read*. `i in 1..40` says: compute every interior cell. `u[t-1, i-1]` says: read the cell one spatial step left, from the previous time step. The two ranges are different expressions, referencing the same coordinate `i` in two roles. The left range says "compute positions 1 through 40." The right offsets say "read positions 0 through 41." The overlap is exact — the interior reads both boundaries without computing them.

The complete program — initial condition, boundary conditions, interior evolution — is four `let` statements. No loop construct. No time-stepping primitive. The ranges are the loops. The offsets are the stencil. The recurrence across `t` is the same mechanism from Chapter 6, applied to a spatial grid. The coordinate `t` carries time. The coordinate `i` carries space. The names distinguish them.

In a positional framework, the equivalent is nested loops with manual index arithmetic:

```
u = np.zeros((81, 41))
u[0, 20] = 10.0
for t in range(1, 81):
    u[t, 0] = 0.0
    u[t, 40] = 0.0
    for i in range(1, 40):
        u[t, i] = u[t-1, i] + r * (u[t-1, i-1] - 2*u[t-1, i] + u[t-1, i+1])
```

The loops are explicit. The ranges are numbers. The stencil pattern is visible — but it is visible in the arithmetic, not in the structure. If the grid expands to 2D (heat on a plate) or 3D (heat in a volume), the positional version adds nested loops. The named version adds coordinates:

```
let u[t in 1..80, i in 1..40, j in 1..40] =  // 2D interior
    u[t-1, i, j] + r * (u[t-1, i-1, j] + u[t-1, i+1, j]
                      + u[t-1, i, j-1] + u[t-1, i, j+1]
                      - 4.0 * u[t-1, i, j]);
```

Two coordinates added. The stencil grew from three neighbors to five. The range `i in 1..40, j in 1..40` excludes the 2D boundary. The structure — ranges on the left, offsets on the right — is identical in 1D and 2D. The notation scales. The loops scale too, but they scale by nesting. The nests grow. The names don't.

This is the third form of range expression. The first was a fixed range (`k in 0..3`). The second was a range that references a coordinate (`k in 0..i+1` for cumulative operations). The third is a range that excludes boundaries, combined with offsets, to define a moving stencil over a grid. All three use the same mechanism: a coordinate appears in a range expression on the left, and in index expressions (possibly with offsets) on the right. The range says what to compute. The offsets say what to read. The coordinate name ties them together.

Ranges are expressions. Stencils are ranges with offsets. The PDE is the program.

Every section in this chapter was a variation on one operation: a single coordinate splitting into two roles.

Distance matrix: `point` becomes `point_i` and `point_j`. Convolution: `oh` and `kh` together index the spatial input — one coordinate from the output, one from the kernel. Depth-to-space: `c_out * 4 + dy * 2 + dx` splits the channel index into a channel group and two sub-pixel offsets. Fancy indexing: `k`

appears in both index positions for pairwise, or `i` and `j` are different for outer-product — coordinate-sharing IS the disambiguation. Gather vs. scatter: same split, different direction.

In every case, the split is the operation. The names record it. The positional notation records the mechanic — `unsqueeze`, `permute`, `reshape`, `None`, `:` — and buries the identity. When you read `points[:, None, :] - points[None, :, :]`, you see three dimension manipulations. When you read `points[point_i, dim] - points[point_j, dim]`, you see one split. The difference is the distance between a notation that records what happened and a notation that records why.

A notation that records the split gives the split a name. A notation that records only the mechanic gives the mechanic a position. Positions shift. Names don't.

Index arithmetic was the most complex coordinate manipulation in Part II. Next: what happens when we differentiate through every operation we have built—applying the Inversion Rule systematically to reductions, broadcasts, index arithmetic, and recurrence. The gradient is coordinate set subtraction, applied in reverse. The five-step pullback is the procedure. The coordinate names are the bridge between the forward and backward directions.

Chapter 8 · Names Through Differentiation

"In the forward pass, you eliminate information. In the backward pass, you guess."

Combinations · Automatic differentiation and the pullback

The last chapter ended with index arithmetic — `oh + kh` in the forward pass became `ih - kh` in the backward pass. It was the Inversion Rule, applied to coordinates that carry arithmetic.

You've been computing gradients all week. `loss.backward()` handles them. You don't think about them. Then you write a custom backward pass — a `torch.autograd.Function` or a manual gradient check — and suddenly you're staring at a sum over the wrong axis, wondering which coordinate you missed.

The autograd engine computes gradients by tracing the forward pass and inverting each operation. When you write the backward pass by hand, you are the engine. And the engine's hardest question is: *over which coordinates do I sum?*

The answer is always the same. It's the Inversion Rule from Chapter 2, applied to every operation in the forward graph: forward broadcast becomes backward reduction, forward reduction becomes backward broadcast. This chapter shows that the five-step pullback — the procedure for deriving any gradient by hand — is coordinate set subtraction, applied in reverse.

The Shopping Cart and the Restocking Run

An intuition model:

You are writing the backward pass for a custom layer. The forward code is in front of you:

```
let C[i, j] = sum[k](A[i, k] * B[k, j]);
```

You have `dC[i, j]`—the gradient of the loss with respect to `C`. You need `dA[i, k]`. You don't want to look up the formula. You don't want to memorize the transpose rule. You just want to derive it from what the forward code says.

Start from first principles. `A[i, k]` appears exactly where? Inside the `sum[k]`: it is multiplied by `B[k, j]` and then summed over `k`. For a specific element `A[i0, k0]`, which output cells does it contribute to? Every output cell where `i = i0` AND the sum includes `k = k0`. That means `C[i0, j]` for *all* `j`.

So `A[i0, k0]` sends a contribution `A[i0, k0] * B[k0, j]` to `C[i0, j]`. The gradient signal `dC[i0, j]` must flow back along that same path. The local derivative of `A[i, k] * B[k, j]` with respect to `A[i, k]` is `B[k, j]`. So the contribution from output cell `C[i0, j]` back to `A[i0, k0]` is `dC[i0, j] * B[k0, j]`.

Now sum over all the output cells that received a contribution. Those are all `j` positions. The coordinate `j` is in `C` but NOT in `A`. Sum over it:

```
let dA[i, k] = sum[j](dC[i, j] * B[k, j]);
```

You just derived the matmul pullback. You didn't memorize it. You didn't look it up. You did coordinate accounting: which coordinates does the output have that the operand doesn't? Sum over those.

What just happened: tracing which coordinates were "paths" from `A` to `C`. The coordinate `j` was a path—every output position along `j` received input from `A[i0, k0]`. To send the gradient back, you had to sum over that path. The coordinate `k` was consumed by the forward sum—it existed in `A` and was eliminated.

The backward pass doesn't need to sum over `k` because `A[i, k]` only contributed to `C[i, j]` through the specific `k` value in the sum. Each `k` position's gradient is independent.

Here is the pattern. In the forward pass, some coordinates are *consumed*—a reduction (`sum[k]`) eliminates them. Some coordinates are *silent*—a broadcast copies a value along them without the value depending on them. In the backward pass, every consumption becomes a broadcast (the gradient must be spread back over what was consumed) and every silence becomes a reduction (all the copies must be collected).

The forward pass is shopping; the backward pass is restocking the same receipt in reverse.

The Gradient as Coordinate Subtraction

The general rule:

`@loss / @W` has the same shape as `W`. The coordinates on the gradient result are exactly the coordinates on the denominator.

Apply this to matrix multiplication. Here is `C[i,j] = sum[k](A[i,k] * B[k,j])` and its gradient, forward and backward, side by side:

On the left, `sum[k]` consumes `k` in the forward pass. On the right, the gradient `dA[i,k]` sums over `j` — the coordinate in `C` that is not in `A`. The figure reads the same in both directions: forward consumption becomes backward broadcast, forward broadcast becomes backward reduction.

The pullback rule in one sentence: **the gradient sums over the set-difference of coordinates between the output and the operand.** Output has `{i, j}`. Operand `A` has `{i, k}`. Difference: `{j}`. Sum over `j`. The missing coordinate `k` is provided by the other operand `B[k, j]`.

Coordinate accounting. No transpose rules. No memorization.

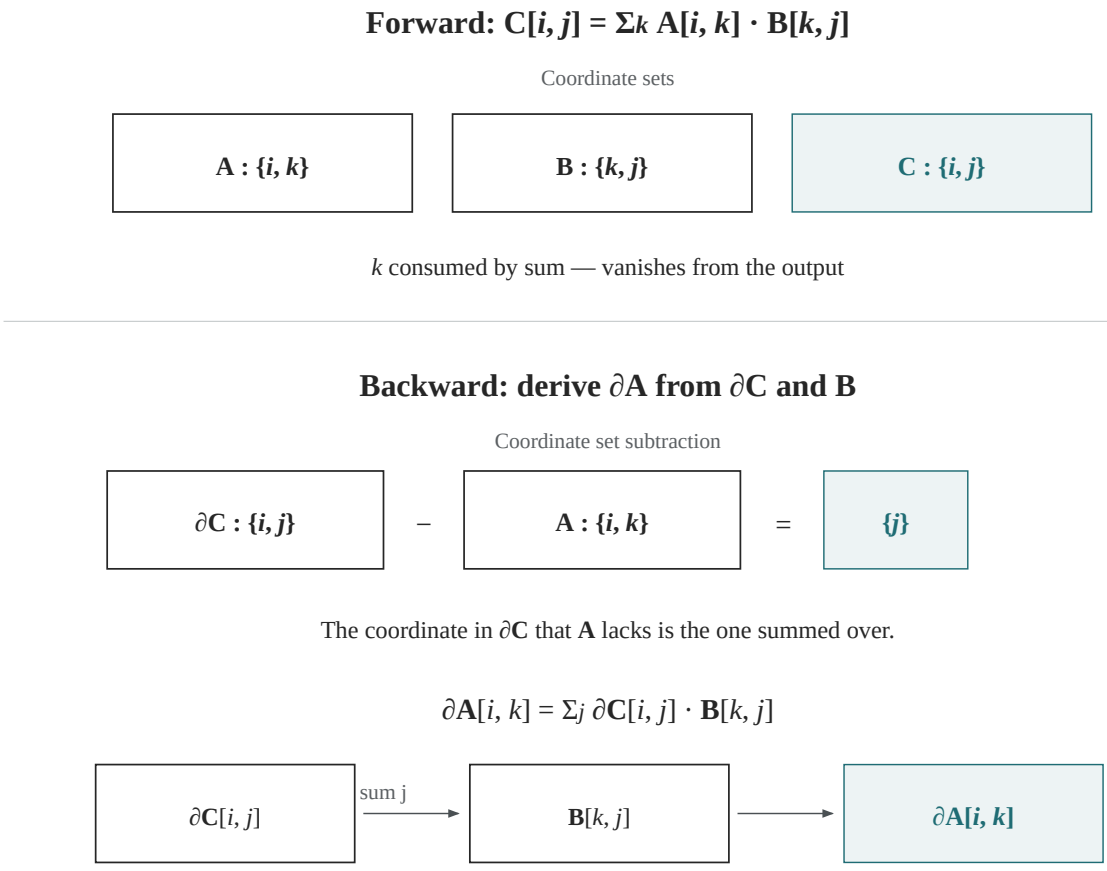
The Five-Step Pullback Procedure

Before reading the procedure, try it yourself. Forward: `C[i, j] = sum[k](A[i, k] * B[k, j])`. You have `dC[i, j]` — the gradient of the loss with respect to `C`. You want `dA[i, k]`. Which output cells does `A[i0, k0]` contribute to? What's the local derivative? Which coordinate must you sum over?

Given a forward expression and a target operand, derive the gradient:

1. **Hold one cell** of the target operand. Choose a specific element—say `A[i0, k0]`.
2. **List every output cell that reads it.** For `C[i, j] = sum[k](A[i, k] * B[k, j])`, the held cell is read by every output where `i = i0` and the sum includes `k = k0`. That means `C[i0, j]` for *all* `j`.
3. **Attach the incoming gradient.** Each output cell `C[i0, j]` carries a gradient signal `dC[i0, j]`. The contribution from the path through `A[i0, k0]` is `dC[i0, j] * B[k0, j]`.
4. **Multiply by the local derivative.** For elementwise multiplication inside the sum, the local derivative of `A[i, k] * B[k, j]` with respect to `A[i, k]` is `B[k, j]`.
5. **Sum the routes.** The path coordinate—the coordinate in `C` but not in `A`—is `j`. Sum over it.

   Path coordinate. A coordinate is a *path coordinate* for an operand if it appears in the output of a forward computation but does not appear in that operand's index pattern. The gradient with respect to that operand sums over all of its path coordinates. Formally: `paths(operand, output) = {coordinates in output} \ {coordinates in operand}`.



ıe pullback rule: sum over every coordinate the output has that the operand lacks. The other operand provides the re

Figure 9: Forward and backward: the gradient sums over coordinates in C but not in A

The result: `dA[i, k] = sum[j](dC[i, j] * B[k, j])`. No calculus memorization. No transpose rules. Just coordinate accounting.

The Pullback in One Example

Observe the five steps on a broadcast-add. Forward:

```
let out[i, j] = A[i, j] + bias[j];
```

Given `d_out[i, j]`, find `d_bias[j]`. What coordinates does the output have? `{i, j}`. What coordinates does `bias` have? `{j}`. The path coordinate (in output but not in bias) is `{i}`. Sum over `{i}`.

1. Hold one cell: `bias[j0]`.
2. Every output cell reads it: `out[i, j0]` for *all* `i`. The held `j0` value is copied to every `i` position.
3. Attach the incoming gradient: each output cell carries `d_out[i, j0]`. The contribution from the path through `bias[j0]` is `d_out[i, j0] * 1` (the local derivative of `x + bias` wrt `bias` is 1).
4. Local derivative: 1.
5. Sum over the path coordinates: output has `{i, j}`, bias has `{j}`. Difference: `{i}`. Sum over `i`.

Result: `d_bias[j] = sum[i](d_out[i, j])`. The broadcast coordinate `i` becomes the reduction coordinate. The Inversion Rule, mechanically applied.

Verify with coordinate set subtraction alone. Forward: `out[i, j] = A[i, j] + bias[j]`. `out` has `{i, j}`, `bias` has `{j}`. Set difference: `{i}`. Sum over `{i}`. `d_bias[j] = sum[i](d_out[i, j])`.

The pattern: the gradient sums over whatever is in the output but not in the operand. Five steps. No calculus memorization. The coordinate sets tell you what to sum over. The forward expression tells you what to multiply by.

Convolution Gradients

A convolution is a sum of products with index arithmetic:

```
let conv[b, oc, oh, ow] = sum[ic, kh, kw](
    input[b, ic, oh + kh, ow + kw] * weight[oc, ic, kh, kw]
);
```

The gradient with respect to `weight` sums over everything that `weight` does not own:

```
let dW[oc, ic, kh, kw] = sum[b, oh, ow](
    dConv[b, oc, oh, ow] * input[b, ic, oh + kh, ow + kw]
);
```

The coordinates `b`, `oh`, `ow` are summed away because they appear in the output but not in `weight`. The coordinates `oc`, `ic`, `kh`, `kw` survive because they *are* `weight`'s coordinates.

Again: set subtraction. The formula is mechanically derivable from the coordinate sets. The same five steps produce the weight gradient with the correct index arithmetic.

Derive it yourself: The weight gradient. The linear layer is `z[b, out] = sum[in](x[b, in] * W[out, in]) + bias[out]`. You derived `d_bias[j]` above by coordinate set subtraction. Now derive `dW[out, in]`.

Apply the five steps. The output `z[b, out]` has coordinates `{b, out}`. The operand `W[out, in]` has coordinates `{out, in}`. Path coordinates in the output but not in `W`: `{b}`. But `W` also has `in`, which the output does not have—because `in` was consumed by the forward `sum[in]`. Consumed coordinates survive in the weight gradient. The path coordinates to sum over are `{b}`—the coordinate `W` was silent on in the forward pass.

Before reading further, write the answer. Then verify: does your `dW` have coordinates `{out, in}`? Does the sum go over `{b}`?

```
dW[out, in] = sum[b](dZ[b, out] * x[b, in])
```

The sum is over `b`—the coordinate `W` broadcasts over. `in` survives because each `in` position gets an independent gradient contribution through the forward sum. `out` survives because it is shared with the output. Two coordinates survive. One is summed away. The five steps, mechanically applied. You didn't memorize the formula. You read it off the coordinate sets.

The Broadcast Self-Audit

A forward broadcast makes a promise. A backward gradient collects dependence. If the two disagree, the gradient is silently wrong.

Consider a linear layer with a bias:

```
let z[b, out] = sum[in](x[b, in] * W[out, in]) + bias[out];
```

The bias term omits `b`—it promises that the bias does not depend on the batch element. Now compute the gradient:

```
let d_bias[out] = sum[b](d_loss[b, out]);
```

The gradient sums over `b`. Why? Because in the forward pass, `bias[out]` was replicated across every batch element. Every batch element carries a piece of the gradient signal. To update `bias`, collect all those pieces. The omitted coordinate becomes the reduced coordinate in the backward pass.

Three questions for any broadcast:

1. **What coordinate am I broadcasting over?** Is the name visible in the code, or is it inferred from position?
2. **Is independence truly justified?** Does the broadcast value genuinely not depend on that coordinate?
3. **What will the gradient do?** Does the backward reduction produce the right shape for the parameter update?

These three questions are the broadcast self-audit. They catch the class of bugs where a broadcast is shape-correct but semantically wrong.

Custom Differentiation with `@fn`

Some functions have derivatives that are better expressed directly:

```
fn relu(x) { if x > 0.0 { x } else { 0.0 } }

@fn relu(x) {
```

```
    if x > 0.0 { @x } else { 0.0 }
}
```

The `@fn` declaration shares the function's name and parameter list. Inside the body, `@x` refers to the tangent flowing through parameter `x`. Custom rules can be coordinate-aware:

```
@fn softmax[j](x: [f32; ..left, j, ..right]) {
    softmax_tangent[j](x, @x)
}
```

The coordinate parameter `j` appears in both the primal function and its derivative rule. The tangent computation follows the same coordinate contract as the primal.

Now the case where `@fn` is not an optimization but a necessity. `argmax` is not differentiable — its true derivative is zero almost everywhere. The autodiff pass has no branch for `ReductionOp.ARGMAX`. If you want a gradient through a selection, you need a Straight-Through Estimator:

```
fn ste_top1[j](p: [f32; ..left, j, ..right]) -> [i32; ..left, ..right] {
    argmax[j](p[..left, j, ..right])
}

@fn ste_top1[j](p: [f32; ..left, j, ..right]) {
    soft_surrogate_tangent[j](p, @p)
}
```

Before reading on, answer this: what connects the forward pass to the backward pass? Which name appears in both, and what does it guarantee?

The coordinate `j`. It appears in `argmax[j]` (hard selection along `j`) and in `soft_surrogate_tangent[j]` (soft surrogate along `j`). The compiler verifies the coordinate match — the tangent rule must consume the same coordinate the primal consumes. Forward and backward are locked to the same axis by name.

PyTorch can do this — `torch.autograd.Function` with custom `forward`/`backward`. But `dim` is an integer. If the upstream tensor changes layout, `dim=1` silently points to the wrong axis. The STE still runs. The gradient still flows. It just flows to the wrong coordinate. The integer doesn't know.

Here, `j` is a name. The coordinate contract extends from the forward pass into the tangent rule. Same name, checked in both directions.

Where Clauses in the Backward Pass

A where clause acts as a gate in both directions:

```
let pos_sum = sum[i](data[i]) where data[i] > 0;
```

In the forward pass, only positive elements are summed. In the backward pass, the gradient signal is distributed only to the positive elements. Elements that were filtered out receive zero gradient. You don't write a separate backward filter. The where clause defines the domain of the operation, and the domain applies in both directions.

The where clause is the gate. Items that pass the condition in the forward pass receive gradient in the backward pass. Items that don't, don't. The condition is written once.

The most common where clause in practice: masked softmax. You have a sequence with padding. The logits for padding positions should be ignored—both in the forward pass (they should not contribute to the sum) and in the backward pass (they should receive zero gradient):

```
let probs[b, j] = softmax_over_valid[class](logits[b, class])
    where valid[b, class];
```

The `valid[b, class]` tensor is a boolean mask—`true` for real tokens, `false` for padding. The where clause gates every operation inside `softmax_over_valid`: the max reduction, the subtraction, the exponentiation, the sum reduction, and the division. Five operations. One gate.

In PyTorch, you'd write a custom masked softmax, or add `float('-inf')` to masked positions before the softmax, or use `torch.where` after the softmax to zero out padding. Each approach has its own backward behavior. The `-inf` trick works for softmax but not for mean. The `torch.where` approach leaves non-zero gradients on padding positions (they were computed, then zeroed—waste). The custom function is correct but requires writing a custom backward pass.

The where clause avoids all three. The gate is part of the operation's domain. The compiler reads it once, applies it to every operation inside the reduction, and generates both the forward gating and the backward gating. No custom backward code. No `-inf` hack. No wasted computation. The gate is written where it belongs—in the domain specification of the operation it gates.

Now a harder one. What if the where clause references a coordinate that is consumed by the operation?

```
let top_sum[b] = sum[class](logits[b, class]) where logits[b, class] > threshold[b];
```

The where clause references `class`—the coordinate being consumed by `sum`. The condition `logits[b, class] > threshold[b]` is a tensor of shape `(batch, class)`. Each `(b, class)` position has a boolean gate.

In the backward pass, the gradient signal `d_top_sum[b]` is distributed back to `logits[b, class]`—but only to positions where the gate was open. For each `b`, elements of `class` that were below `threshold[b]` receive zero gradient. The where clause is evaluated during the forward pass and its boolean mask is stored. The backward pass reads the mask and gates the gradient accordingly.

No separate backward code. No custom `@fn` rule. The where clause is part of the reduction's domain specification, and the compiler generates both the forward gating and the backward gating from it. The coordinate names in the condition—`class` in this case—tell the compiler which dimension the gate applies to.

LayerNorm: A Complete Gradient Walkthrough

Tracing the gradient of LayerNorm end to end:

Forward pass:

```
fn layer_norm[feature](x: [f32; ..b, feature],
                       gamma: [f32; feature],
                       beta: [f32; feature])
    -> [f32; ..b, feature]
{
    let mean_val[..b] = mean[feature](x[..b, feature]);
    let centered[..b, feature] = x[..b, feature] - mean_val[..b];
    let var[..b] = mean[feature](centered[..b, feature] ** 2.0);
```

```
    (centered[..b, feature] / (var[..b] ** 0.5 + 1e-5)) * gamma[feature] + beta[feature]
}
```

`d_out[..b, feature]` is the gradient of the loss with respect to the LayerNorm output. Compute `dx[..b, feature]`, `d_gamma[feature]`, and `d_beta[feature]`.

Step 1: Gradient of beta. The output expression ends with `+ beta[feature]`. This is an elementwise addition. `beta` omits `..b` in its index pattern—it broadcasts over all batch dimensions. By the Inversion Rule, the gradient sums over the broadcast set:

```
let d_beta[feature] = sum[..b](d_out[..b, feature]);
```

`d_out` has `{..b, feature}`. `beta` has `{feature}`. Difference: `{..b}`. Sum over it. Result: `{feature}`, matching `beta`'s shape.

Step 2: Gradient of gamma. Same pattern. `gamma[feature]` broadcasts over `..b`. The gradient sums over `..b`:

```
let d_gamma[feature] = sum[..b](
    d_out[..b, feature] * centered[..b, feature] / (var[..b] ** 0.5 + 1e-5)
);
```

The local derivative of `gamma * x_norm` with respect to `gamma` is `x_norm`. Multiply by the incoming gradient. Sum over the broadcast set. Done.

Step 3: Gradient of x. This is the complex one—`x` contributes to the output through three paths: directly through `centered` (which contains `x`), and indirectly through `mean_val` and `var` (which were computed from `x`). But the coordinate accounting still works: `x` has `{..b, feature}`. The output has `{..b, feature}`. No coordinate difference. The gradient must have the same shape. No sum—the contributions from all three paths accumulate at each `(..b, feature)` position.

Coordinate set subtraction tells you *which* coordinates get summed. The local computation tells you *what* to sum. Together they produce the gradient without shape memorization.

The Recurrence Gradient: Time as a Path Coordinate

Every gradient so far summed over spatial coordinates — `j` in matmul, `b` in bias, `oh, ow` in convolution. Recurrence introduces a new question: what happens when the forward pass steps through time?

Consider the simplest recurrence — a scalar accumulator:

```
let h[0] = x;
let h[t in 1..T] = h[t-1] + 1.0;
```

Before reading the answer, derive the gradient yourself. Forward: `h[0] = x`, `h[1] = h[0] + 1`, `h[2] = h[1] + 1`, `h[3] = h[2] + 1`. Given `dh[t]` (the gradient of the loss with respect to each `h[t]`), what is `dh[0]`? What is `dx`?

Trace the dependence. `h[0]` contributes to `h[1]`. `h[1]` contributes to `h[2]`. `h[2]` contributes to `h[3]`. The gradient signal must flow backward along every link in this chain.

Take a minute. Draw the dependence graph. Then read on.

`h[3]` depends on `h[2]`. By the chain rule, `dh[2]` receives `dh[3] * 1` (the local derivative of `h[t-1] +`

1 with respect to h[t-1] is 1). But h[2] may also receive gradient directly if the loss reads h[2]. So dh[2] = dh[2]_direct + dh[3] * 1.

h[2] contributes to h[3]. h[1] contributes to h[2]. All the way back to h[0]. The gradient at step t receives contributions from two sources: the direct loss signal at step t, and the backpropagated signal from step t+1. The second source chains backward through every subsequent time step.

Unrolled for T=4:

```
dh[3] = dh[3]_direct
dh[2] = dh[2]_direct + dh[3]
dh[1] = dh[1]_direct + dh[2]
dh[0] = dh[0]_direct + dh[1]
dx    = dh[0]
```

Now a recurrence with a parameter — the kind that appears in every optimizer and RNN:

```
let h[0] = x;
let h[t in 1..T] = h[t-1] * W + b;
```

Given dh[t] for all t, find dW. Apply the five-step pullback.

Step 1: Hold one cell of W. W is a scalar (no coordinates). The held cell is W.

Step 2: List every output cell that reads it. The output has {t}. The parameter W appears at every time step. W is read by every h[t] for t in 1..T — the recurrence body multiplies h[t-1] by W at each step.

Step 3: Attach the incoming gradient. At step t, the contribution through W to h[t] is dh[t] * h[t-1] (local derivative of h[t-1] * W with respect to W is h[t-1]).

Step 4: Multiply by the local derivative. h[t-1] at each step.

Step 5: Sum over the path coordinates. The output has {t}. The operand W has {} (scalar). Path coordinate: {t}. Sum over t.

```
let dW = sum[t in 1..T](dh[t] * h[t-1]);
```

The coordinate set subtraction works exactly as before. t is the path coordinate — it appears in the output h[t] but not in the parameter W. Sum over t. The recurrence unrolling is the mechanism that makes the sum possible; the coordinate accounting tells you *what* to sum over.

The same pattern applies to db:

```
let db = sum[t in 1..T](dh[t] * 1.0);
```

And to dx (the gradient of the initial state). x has no coordinates where h has {t}. The path coordinate is {t}. But x only appears at h[0] — not at every t. The sum over t collapses to a single term: dh[0], which incorporates the backpropagated signal from all future steps through the chain rule.

Now the general form. Forward:

```
let h[t in 0..T, i] = initial[i];
let h[t in 1..T, i] = f(h[t-1, i], W, i);
```

The parameter W omits t. The gradient sums over t:

```
let dW = sum[t in 1..T, i](dh[t, i] * f/ W);
```

The recurrence dimension `t` becomes a path coordinate — summed over in the gradient, exactly like `j` in matmul or `b` in bias. The unrolling is the mechanism. The coordinate set subtraction picks the summation axes. No new rules. No special BPTT formula to memorize. `t` is just another coordinate that the output has and the parameter doesn't.

The recurrence backward pass is coordinate set subtraction where one of the coordinates happens to carry a dependence chain. The chain rule unrolls the dependence. The coordinate sets determine the summation. Two mechanisms. One derivation.

The compiler's recurrence analysis matters for differentiation. When the compiler detects `history_lookback_steps = 1` and `downstream_tail_steps = 1`, it allocates a rolling buffer of size 2. The same analysis tells the backward pass how many steps to unroll and which coordinates must be summed. The structural fact (`t-1` in the forward pass) determines both the memory strategy AND the gradient summation. One minus sign. Two compiler passes. Same coordinate name.

Be the Compiler: Derive the RNN Gradient

An RNN cell with a hidden state and an output projection:

```
let h[t in 0..T, hidden] = initial[hidden];
let h[t in 1..T, hidden] = tanh(
    h[t-1, hidden] * W_h[hidden, hidden] + x[t, in] * W_x[in, hidden] + b[hidden]
);
let y[t, out] = h[t, hidden] * W_y[hidden, out];
```

Three parameter gradients to derive: `dW_h`, `dW_x`, `dW_y`, `db`. For each one, ask: what coordinate does the parameter omit that the output has? That coordinate is the path coordinate — sum over it.

Before reading the answer, write each gradient. Use the five steps. The coordinate sets tell you what to sum. The forward expression tells you what to multiply.

The pattern from recurrent gradients is the same pattern from every pullback: hold one cell, trace its contribution to every output cell, sum over the path coordinates. The recurrence adds a dependence chain through time. The coordinate accounting does not change. `t` is a coordinate. The output has it. The parameter doesn't. Sum over `t`.

The pullback procedure was always coordinate set subtraction. Recurrence proved it survives the hardest test.

Fancy Indexing: Where the Gradient Scatters

Chapter 7 distinguished pairwise indexing from outer-product indexing by coordinate name: `k` in both positions means pairwise; `i` and `j` different means outer-product. The gradient behavior differs in exactly the way the names predict.

Pairwise indexing. Forward:

```
let gathered[k] = matrix[idx[k], col_idx[k]];
```

The output has `{k}`. The operand `matrix` has `{i, j}`. Path coordinate: `{k}`. The gradient must sum over `k` and restore the contribution to the `{i, j}` shape of `matrix`. But each `k` contributes to exactly one `(i,`

`j)` position — `(idx[k], col_idx[k])`. The sum over `k` is a **scatter-add**: accumulate `d_gathered[k]` into `d_matrix[idx[k], col_idx[k]]` for each `k`.

No dense sum. No broadcast. A sparse scatter, indexed by the same index arrays as the forward pass, accumulating at the gathered positions. The coordinate name `k` tells you the path coordinate. The index arrays tell you where each `k` maps.

Outer-product indexing. Forward:

```
let outer[i, j] = matrix[idx[i], col_idx[j]];
```

The output has `{i, j}`. The operand `matrix` has the same `{i, j}` in its index pattern (via `idx` and `col_idx`). No coordinate to sum — the gradient flows directly: `d_matrix[idx[i], col_idx[j]] += d_outer[i, j]`.

But here is the difference from pairwise: if `idx[i] = 3` and `col_idx[j] = 3` for multiple `(i, j)` pairs, the gradient at `matrix[3, 3]` accumulates contributions from every such pair. The outer-product scatter can have collisions. The pairwise scatter cannot — each `k` maps to a distinct `(idx[k], col_idx[k])` by construction.

The coordinate names distinguish the two indexing modes. The same names determine the gradient accumulation pattern. Pairwise: sum over `k`, no collisions. Outer-product: no sum, collisions possible. One rule. Two behaviors. The names tell you which.

The compiler does not have a special scatter-add pass for indexing gradients. It inlines the gather body into Einstein notation (`result[i,j] = data[indices[i], j]`) and differentiates through the `RectangularAccessIR` node: the differential of the array is wrapped in the same index access pattern. This general autodiff produces the correct gradient — `d_data[indices[i], j] += d_result[i, j]` — without recognizing it as a scatter. The coordinate accounting is a conceptual framework that predicts what the autodiff will produce. The autodiff is the mechanism that produces it.

Where Set Subtraction Stops

The coordinate set subtraction rule — sum over `{output coords} \ {operand coords}` — derives the gradient for most tensor operations: reductions, broadcasts, matmul, convolutions, recurrences, where clauses. Three operations require a different treatment.

Non-differentiable operations. `argmax[j](x[i, j])` returns an index, not a continuous value. The output is an integer coordinate position. Perturbing `x` by does not change the argmax except at boundaries where two values are exactly equal. The gradient is zero almost everywhere, undefined at ties. The autodiff pass has no branch for `ReductionOp.ARGMAX` — the operation is not differentiable. The compiler leaves a zero gradient, but this is a silent default, not a checked error. A correct compiler should report `error: gradient of argmax is not defined` at the point where `@loss / @argmax_result` is written. Downstream code that needs a gradient through a selection must use a soft approximation (softmax with temperature) or a straight-through estimator wrapped in `@fn`. The `max` reduction *is* differentiable — the compiler uses `SelectAtArgmaxIR` to route the cotangent to the winning index. `argmax` returning an integer index is not. The distinction between `max` (differentiable, gradient routes to argmax position) and `argmax` (non-differentiable, returns integer) is a compiler-semantic distinction, not a syntactic one.

`prod`. `result = prod[i](data[i])`. The gradient of `data[k]` is `d_result * result / data[k]` — the product of all elements EXCEPT `data[k]`. This does not fit the pure coordinate-set-subtraction pattern because the local derivative of `data[k]` depends on every other element's value. The compiler handles `prod` with a built-in gradient rule: the autodiff pass recognizes `ReductionOp.PROD` and

emits `sum((total_product / body) * d_body)`. Unlike `sum` (which distributes the tangent as-is) or `max` (which routes to the winning index via `SelectAtArgmaxIR`), `prod` requires the full product to compute the local derivative. The coordinate accounting still works — `i` is the consumed coordinate, the gradient restores it — but the *value* of the local derivative requires knowing the total product, not just the coordinate structure.

Reshape and permute. These operations change coordinate layout without changing values. The gradient is the inverse operation: a forward `permute` of `(batch, feature)` to `(feature, batch)` has a backward `permute` of `(feature, batch)` to `(batch, feature)`. The coordinate accounting is trivial — no sum, no path coordinate — because every output cell reads exactly one input cell. The gradient flows through the inverse layout change.

For everything else — reductions, broadcasts, matmuls, convolutions, recurrences, where clauses, pack polymorphism, cumulative scans, normalization, attention — the five-step pullback is coordinate set subtraction. The names tell you what to sum. The forward expression tells you what to multiply. The rest is accounting.

The Pullback as Coordinate Set Subtraction: A Summary

Every pullback you have seen follows the same rule. Given a forward expression and the operand whose gradient you want:

1. Write the operand's coordinate set. `A` has `{i, k}`.
2. Write the output's coordinate set. `C` has `{i, j}`.
3. The path coordinates = output set minus operand set. `{j}`.
4. Sum over the path coordinates. `dA[i, k] = sum[j](...)`.

The other operand (`B[k, j]`) provides the missing coordinate `k`. The local derivative comes from differentiating the forward operation with respect to the operand. The path-sum structure comes from the coordinate sets.

This is the pullback rule as coordinate accounting. No transpose rules memorized. No Jacobian dimensions counted. Just set subtraction, applied to coordinate names.

In a single equation—carry this with you:

```
dA = Σ_{paths(A, C)}  dC · d(forward)/dA

where paths(A, C) = {coordinates in C} \ {coordinates in A}
```

`paths(A, C)` is the set of coordinates in the output `C` that are NOT in the operand `A`. Sum over them. Multiply by the local derivative. Done. It is the formula behind every gradient derived from a tensor expression. The rest is accounting.

The gradient is not a separate computation from the forward pass. It is the forward pass, read backward. The coordinate names that organize the forward pass—which survive, which are consumed, which are omitted—organize the backward pass in exactly the same way. A coordinate eliminated by a forward sum becomes a coordinate introduced by a backward broadcast. A coordinate omitted by a forward broadcast becomes a coordinate summed by a backward reduction.

The names are the same. The direction is reversed. The principle is symmetric.

The Gradient of a Coordinate-Aware Function Call

When a function call appears in the forward pass, its backward pass is determined by the function's coordinate contract. The gradient does not need to re-verify the contract—the forward call already did. The gradient only needs to apply the Inversion Rule to each primitive operation inside the function body.

Consider `softmax[class](logits)`. Forward: the function body contains `max[class]`, `sum[class]`, elementwise operations, and a division. Backward: `max[class]` becomes a broadcast (only the argmax position receives the gradient), `sum[class]` becomes a broadcast (every summed element receives the gradient), the division's gradient is elementwise.

The coordinate `class` is the reduction axis in both `max` and `sum`. The backward pass broadcasts over `class` for both. The gradient flows through the function's coordinate parameter—`class` in, `class` out. The caller never sees the internal backward operations. The coordinate contract at the call site guarantees that `class` exists on the input, so the gradient exists on the same coordinate.

A custom `@fn` rule for softmax overrides this automatic derivation—but the coordinate parameter in the custom rule must match the primal. The compiler checks this:

```
fn softmax[j](x: [f32; ..left, j, ..right]) -> [f32; ..left, j, ..right] { ... }

@fn softmax[j](x: [f32; ..left, j, ..right]) {
    // Custom backward: softmax_tangent[j](x, @x)
    // j must match the primal's coordinate parameter
}
```

If the custom rule declares `@fn softmax[k]` (different coordinate parameter name), the compiler reports a mismatch. If the custom rule returns a gradient with different coordinates than the primal's input, the compiler reports a mismatch. The coordinate contract extends from the forward pass into the backward pass. The same names, checked in both directions.

Revisit the last gradient you debugged. Write the forward expression with coordinate names. Derive the backward expression using coordinate set subtraction. Does the backward sum match the forward broadcast? The answer to that question is the answer to whether your gradient was correct.

The Pullback in Practice

The five-step pullback procedure applies to matmul, broadcast-add, and sum-to-scalar. It also applies to convolution, normalization, and attention. The procedure is always the same: hold one cell, list every output cell that reads it, attach the incoming gradient, multiply by the local derivative, sum over the path coordinates.

A gradient written or debugged by hand—a custom backward pass, a `torch.autograd.Function`, a manual gradient check—is a forward expression read backward. Writing the forward expression with coordinate names, even if the original code is in PyTorch, makes the backward derivation mechanical. What coordinates does the output have? What coordinates does the operand have? What's the difference? Sum over the difference. No insight required. Coordinate set subtraction, applied to the forward expression, produces the backward sum.

When the derived gradient matches the original, the coordinate accounting confirms the manual derivation—both arrived at the same result through different paths. When they differ, either the

coordinate accounting was wrong, or the original gradient was. Both possibilities are worth checking. The same procedure works for convolution, normalization, and attention. The coordinate sets are larger. The procedure doesn't change.

The pullback is not a separate computation from the forward pass. It is the forward pass, read backward, through the lens of the Inversion Rule. Every forward reduction becomes a backward broadcast. Every forward broadcast becomes a backward reduction. The coordinate names are the bridge between the two directions. The five steps are the procedure for crossing it.

Return to the Transformer

You derived matmul gradients in Chapter 8. Now derive one for the transformer.

```
// forward
let attn_out[head, seq_q, d] = sum[seq_k](weights[head, seq_q, seq_k] * V[head, seq_k, d]);
```

The backward pass needs `d_V` — the gradient of the loss with respect to the value tensor. Use the five-step pullback.

Step one: hold one cell of `V` — a specific `(head, seq_k, d)`. Step two: list every cell of `attn_out` that reads it. The multiplication means every `seq_q` position reads this cell. Step three: attach the incoming gradient `d_attn_out[head, seq_q, d]`. Step four: the local derivative of `weights[...] * V[head, seq_k, d]` with respect to `V[head, seq_k, d]` is `weights[head, seq_q, seq_k]`. Step five: sum over the path coordinates. The path coordinate is `seq_q` — the coordinate that varies while `(head, seq_k, d)` is held fixed.

What are the path coordinates? What does `d_V` sum over?

```
let d_V[head, seq_k, d] = sum[seq_q](d_attn_out[head, seq_q, d] * weights[head, seq_q, seq_k]);
```

In the forward pass, `seq_k` was consumed and `seq_q` survived. In the backward pass, `seq_q` is consumed and `seq_k` survives. Reduction and broadcast swapped places — the Inversion Rule, at the scale of attention.

Part II gave you the tools. Part III builds the machine that runs them.

Part II asked one question: when operations compose, does the coordinate story survive? You tested it against every operation in tensor programming. Functions: yes, the contract is checked at the call site. Broadcasts: yes, the three-question audit reveals unwarranted independence claims. Normalization: yes, four variants share one skeleton. Recurrence: yes, the compiler detects the direction of time from a minus sign. Complex terrain: yes, the split of one coordinate into two roles is the operation, and the names record it. Differentiation: yes, the gradient is coordinate set subtraction, read in reverse.

The five-step pullback is coordinate accounting done by hand. Can a machine do it? The answer turns out to be the same question you have been asking since Chapter 2: *which coordinates survive?*

Part III · Construction

Chapter 9 · The Shape of Thought

> "A compiler is a program that reads a program written in one language—and says it again in another."
>
> — Adapted from SICP

Construction · How the compiler reads names

Part II showed that coordinate names survive composition—through functions, through time, through differentiation. Part III asks: can this be automated? This chapter constructs a compiler frontend that reads named coordinates, checks them against five rules, and lowers them to integers. It is the proof that the checks you have been doing by hand—tracing `batch` and `class` through softmax, subtracting coordinate sets at broadcast sites, verifying that function contracts match—can be done by a machine.

You just wrote a short Einlang program:

```
let x[batch, class] = softmax[class](logits[batch, class]);
let y[batch] = sum[class](x[batch, class] * labels[batch, class]);
```

Someone asks: what is the shape of `y`? You cannot run this code—there is no data. But you can still answer.

Trace the coordinates. `logits` has `[batch, class]`. `softmax[class]` preserves the shape—output is `[batch, class]`, bound to `x`. Then `sum[class]` consumes `class`. The surviving coordinate is `batch`. `y` has shape `[batch]`.

You just performed coordinate propagation in your head. You tracked each name—where it was introduced, how it flowed through the function call, whether it survived the reduction. You did not need data. You did not need to run the program. You needed only the names.

This—exactly this—is what a compiler must do. For every line, without data, without execution, it must answer: what is the shape of every tensor? Which coordinates survive each operation? Does the coordinate contract at each call site match the function's declaration?

Before we see the machinery that answers these questions, let's watch the questions themselves catch bugs.

The Wall

Imagine a detective's wall. Five slots, each labeled with one rule. When a bug is found, it gets pinned to the slot of the rule that caught it.

Here is a broken program. It was written by a programmer who intended a linear layer with bias followed by softmax over classes. It compiles. It runs. It produces wrong results.

```
fn predict[class](x: [f32; batch, in], W: [f32; out, in], bias: [f32; out])
    -> [f32; batch, out]
{
    let logits[batch, class] = sum[k](x[batch, k] * W[out, k]) + bias[out];
    softmax[class](logits[batch, class])
}
```

Look at the program. Which names don't match?

Rule 1. `k` is referenced on `x` and `W`, but both declare `in`, not `k`. The reduction introduces a ghost coordinate. Rule 1 pins: `k` is not a declared coordinate of `x` or `W`. The writer meant `in`.

Rule 2. Syntactically, `k` appears in all operands. Rule 2 passes—but the wrong coordinate is being consumed. Rules 1 and 2 together cover both "coordinate doesn't exist" and "coordinate exists but isn't used consistently."

Shape analysis. The output declaration says `(batch, class)` but the reduction body produces `(batch, out)`. `class` appears from nowhere. `out` is produced but not declared. The declared coordinates don't match the coordinates that actually flow through the expression.

Rule 5. Return type says `[f32; batch, out]` but the body's `softmax[class](logits[batch, class])` returns `[f32; batch, class]`. The coordinate `out` in the return type doesn't match `class` in the return value. Pinned.

Rule 3. `bias[out]` omits `batch`—a correct broadcast (bias is independent of batch). Recorded for the gradient. No bug.

Rule 4. No recurrence. Silent.

The corrected program:

```
fn predict[class](x: [f32; batch, in], W: [f32; class, in], bias: [f32; class])
    -> [f32; batch, class]
{
    let logits[batch, class] = sum[in](x[batch, in] * W[class, in]) + bias[class];
    softmax[class](logits[batch, class])
}
```

Three changes. `sum[k]` → `sum[in]`. `W[out, k]` → `W[class, in]`. Return type `out` → `class`. Every coordinate flows from declaration to use. Every name matches.

Not one of these bugs was a shape error. In a positional framework, `sum(axis=1)` on `(batch, in)` and `(out, in)` would produce `(batch, out)`—a valid shape. The code would run. The loss would descend. And the model would be computing a meaningless function.

Every one of these checks reads the same fact from every tensor: its **coordinate layout**—the ordered list of names the tensor carries. When you write `x: [f32; batch, in]`, the compiler records `layout(x) = [batch, in]`. Rule 1 checks whether `k` appears in that list. Rule 3 subtracts lists to find broadcast axes. Rule 5 compares the call-site list against the function's declared parameter list. The five rules are five queries against one data structure. The coordinate layout is the one fact every rule reads.

The five rules are the five ways a name can be wrong: it can refer to a non-existent coordinate (Rule 1), it can fail to appear where the operation requires it (Rule 2), it can broadcast silently without the record the backward pass needs (Rule 3), it can reference the future in a recurrence (Rule 4), or it can violate the contract of a function call (Rule 5). That's it. Those are all the ways.

Derive it yourself. Here is a buggy program. Apply the five rules. Which rules fire? What does each rule catch?

```
fn forward[batch, out](x: [f32; batch, in], W: [f32; out, in], b: [f32; out])
    -> [f32; batch, out]
{
```

```
    sum[k](x[batch, k] * W[out, k]) + b[out]
}
```

Rules 1–5. Go.

But to apply these rules mechanically, the compiler needs a representation where names are preserved and operations are explicit. It needs an **intermediate representation**—a tree where every name is visible.

Before we build the tree, try this: write `sum[k](A[i, k] * B[k, j])` on a piece of paper. Circle every name. Draw an arrow from each name to where it appears. Now erase the brackets and the `sum`. What's left? The structure of the computation—two indices multiplied, one coordinate reduced away, two coordinates surviving. That structure is the IR. The names are its bones.

The IR Tree

Conceptually, the compiler's IR can be understood as a tree of expressions where names are first-class:

```
A[i, j] + B[i, j]
```

```
(+ (index A (i j)) (index B (i j)))
```

Add a reduction:

```
sum[k](A[i, k] * B[k, j])
```

```
(reduction sum (k)
  (* (index A (i k)) (index B (k j))))
```

The reduction coordinate `k` is named, not numbered. A full declaration:

```
let C[i, j] = sum[k](A[i, k] * B[k, j])
```

```
(let-decl (output C (i j))
  (reduction sum (k)
    (* (index A (i k)) (index B (k j)))))
```

`(output C (i j))` declares the surviving coordinates: `i` and `j` survive, `k` is consumed.

Three things the IR preserves: **names** (`i` and `k` remain names, never become `axis=0`), **reduction targets** (`(reduction sum (k) ...)` operates on `k`), and **index patterns** (`(index A (i k))` matches what the source wrote). The IR has not *translated* your program. It has *said it again*, as a tree.

But the IR preserves more than the tree shape. Each node carries its **coordinate set**—the names that survive at that point. `(index A (i k))` carries `{i, k}`. `(reduction sum (k) ...)` carries `{i}`—`k` was consumed. The compiler computes these sets by walking the tree: index nodes introduce coordinates, reductions subtract them, additions merge them. This is exactly what you did by hand at the start of this chapter—tracing `batch` and `class` through softmax and sum, without data, using only the names. The compiler does it mechanically, node by node, for every expression in the program.

Lowering: Names Become Numbers

The tree passed every check. But it cannot be handed to a numerical backend. NumPy does not understand `class`. It needs `axis=1`.

Translating the analyzed tree into executable instructions is **lowering**. The mapping is deterministic: the compiler reads the coordinate layout stored on each tensor's IR node—the same layout that the five rules consulted during checking. `logits` was declared `[f32; batch, class]`, so its layout is `[batch, class]`. At lowering, the compiler walks the layout and assigns positions: `batch → 0`, `class → 1`. Then every index expression in the tree is rewritten: `(index logits (batch class))` touches both axes; `(reduction max (class) ...)` becomes a reduction over axis 1. The layout is the map. Lowering is the lookup. The name is burned.

After lowering, the softmax conceptually becomes:

```
def softmax(logits):
    m = np.max(logits, axis=1, keepdims=True)
    e = np.exp(logits - m)
    return e / np.sum(e, axis=1, keepdims=True)
```

`keepdims=True` was not in the source. The compiler inferred it. Here is the principle, visible in the core loop at the end of this chapter:

The subtraction `logits - max_result` has two operands. `logits` carries `{batch, class}`. `max_result`—the output of `max[class](logits)`—carries `{batch}` because `class` was consumed. The coordinate sets differ: left has `class`, right does not. Set difference: `{batch, class} - {batch} = {class}`. The missing coordinate is `class`. The compiler records the broadcast—not as a flag the programmer writes, but as a requirement the coordinate structure demands. This is not a heuristic. It is the same set subtraction from Chapter 2, applied to the operand coordinate sets at every binary operation.

In the actual compiler, this check lives in the `CoordinateGroundingPass`: at every `BinaryOpIR` node, the left and right operand layouts are compared. When they are equal, the shared layout is stamped on the result. When they differ, no layout can be stamped—the coordinate structures are incompatible, and the mismatch is recorded as a broadcast that the lowering pass must resolve. The core loop at the end of this chapter reduces this logic to four lines: compare the sets, record the differences, return the union. The principle is the same. The implementation is more careful.

The same set-difference logic applies to reductions. `sum[k](A[i, k] * B[k, j])` — `k` appears in both operands but not in the output `C[i, j]`. The compiler subtracts `{k}` from the body's coordinate set to produce the output layout `{i, j}`. The coordinate `k` is a contracting dimension—shared by both operands, consumed by the reduction, absent from the output. The pattern `(i, k) × (k, j) → (i, j)` follows from the coordinate structure alone. No annotation needed. The compiler lowers this to a nested loop with a reduction over `k`. The names told it which axis to contract.

The Panorama: One Name, Five Forms

Here is softmax, in five simultaneous forms:

```
max[class](logits[i, class])                          ← what you wrote
(reduction max (class) (index logits (i class)))     ← what the compiler sees
class: (range 0 n_class), reduction axis, Rule 2      ← what the compiler derives
class → axis=1, reduction                                ← how the name becomes a number
```

```
np.max(logits, axis=1, keepdims=True)                ← what executes (conceptually)
```

Five forms. One name. The name `class` traveled through all five without changing its identity. It was written as `class`, preserved as `class`, verified as `class`, mapped from `class` to `axis=1`. At no point was it guesswork.

Now ask: if the positional version had a bug—if `dim=-1` was normalizing over the wrong axis—at which of the five stages would that bug be caught?

Source: not caught. `dim=-1` is a valid integer. IR: not caught. No names to verify. Analysis: not caught. No coordinate contract to check. Lowering: not caught. `-1` maps correctly—it's the *choice* of `-1` that is wrong. Generated code: not caught. `np.max(logits, axis=-1)` is valid NumPy.

The answer: **none of them.** The positional bug is invisible to all five stages because the information that would expose it—the name of the coordinate—was never written down.

The Einlang bug is caught at Form 3. Analysis checks: does `class` appear in every operand of the reduction? Does it exist on the tensor? The bug surfaces before the program runs, at the stage where names are still names and the compiler can still reason about them.

Range Inference: Where Domains Come From

The constraint solver needs ranges. `oh` ranges over `0..output_height`. Where does that range come from?

Sometimes the user declares it: `oh in 0..output_height`. But in the common case, they don't. When you write `let result[i] = data[i] * 2`, the range of `i` is never stated. The compiler infers it.

The algorithm: find every array access where `i` appears as an index. For each access, look at the array's declared shape at the position where `i` appears. If `data` has shape `[N]`, then `i < N`. If `i` also appears in `arr[i+1]` and `arr` has shape `[M]`, then `i+1 < M`, so `i < M-1`. Collect all inferred upper bounds. Take the minimum — the most restrictive one. That is `i`'s range.

When you write `let result[i] = arr[i] + arr[i+1]`, the compiler finds two accesses: - `arr[i]` → `i < len(arr)` → `i in [0, N)` - `arr[i+1]` → `i+1 < len(arr)` → `i in [0, N-1)`

Intersection: `i in [0, N-1)`. The compiler inferred that `i` cannot reach `N-1` because `arr[i+1]` would be out of bounds. No annotation needed. The name `i`, appearing in two positions, carries enough information.

The entire algorithm is: sort accesses by expression complexity (direct `i` before `i+1`), back-compute a range from each, intersect. The coordinate-name philosophy makes this possible — every index variable appears literally in array accesses, so every range is inferrable from the shapes those arrays were declared with.

Index Arithmetic: The Constraint Solver

With ranges in hand, the compiler can check a harder problem: index arithmetic. `input[b, ic, oh + kh, ow + kw]` — the expression `oh + kh` must not exceed the input's spatial extent. Given the ranges for `oh` and `kh`, is this checkable at compile time?

When the domain sizes are known statically, the answer is yes. The compiler's constraint solver reads the index expression and the declared bounds, then solves for each index variable. The algorithm is pattern-matching on the expression tree.

Take the convolution index `oh + kh`. The compiler knows: - `oh` ranges over `0..output_height` (from the output declaration) - `kh` ranges over `0..kernel_height` (from the weight declaration) - The input has `input_height` (from the input declaration) - Constraint: `oh + kh < input_height`

The solver must verify that `oh + kh < input_height` for all valid `oh` and `kh`. It does this by solving for each variable in the worst case. For `oh + kh`, the maximum value is `(output_height - 1) + (kernel_height - 1)`. If this maximum is less than `input_height`, the constraint holds. If the domains are known, the check is a single comparison.

But the solver handles more complex expressions. Consider `(i * 2 + offset) < N`. The solver pattern-matches the expression tree:

1. **Top level is addition** `(i * 2) + offset < N`. Isolate the target: `i * 2 < N - offset`.
2. **Recurse into multiplication** `i * 2 < N - offset`. Isolate: `i < (N - offset) / 2`.
3. **Base case** `i < bound`. The range is `[0, ceil(bound))`.

The solver chains these rewrites automatically. Addition adjusts the bound by subtraction. Multiplication adjusts by division (with ceiling, so `(a + b - 1) / b` for safety). Division by a constant adjusts by multiplication. When the target appears with a negative coefficient—`k - target < bound`—the solver returns nothing: negative coefficients require a lower bound, and the solver only computes upper bounds. The pattern is recognized. It is declared unsolvable by the current pass.

The solver operates entirely on IR nodes, not Python integers. The bound `N` can be a dynamic expression like `image.shape[0]` rather than a compile-time constant. The ceiling division `ceil(a / b)` is implemented as `(a + b - 1) / b` at the IR level, so it works for any input size. This means the bounds inference produces symbolic ranges that are safe for any runtime shape.

The constraint solver is not a general-purpose theorem prover. It handles the patterns that appear in real tensor index expressions: linear combinations with positive coefficients, simple arithmetic (`i*2`, `i/2`, `i+1`, `i-1`), and the common stencil patterns from Chapter 7. For patterns it cannot solve—modulo, negative coefficients, non-linear expressions—the solver returns no range, and lowering fails with a compile error. The failure is a compile error, not a runtime surprise. The name is attached to the error, as it is to every error in this compiler.

What the solver proves: that a coordinate arithmetic expression, evaluated over its declared ranges, stays within the declared bounds of the tensor it indexes. What it cannot prove—correctness of the arithmetic itself—is not a failure of the solver. It is the boundary from Chapter 14. The solver narrows that boundary. Every expression it proves safe is one less degree of freedom for silent errors. Every expression it cannot prove is flagged before the program runs.

The Core Loop

The Wall presented five rules. Before reading the implementation: Rules 1 and 2 (undeclared coordinates, missing operands) require checking every index expression against its tensor's declaration. Rule 3 (broadcast recording) requires comparing operand coordinate sets at every binary operation. Rule 4 (causality) applies only to recurrences. Rule 5 (contract matching) applies only at function call boundaries. Which of these does a tree-walking checker handle naturally? Which require additional machinery?

The entire compiler frontend fits in fifteen lines:

```
check(expr, env, errors):
  match expr:
    case Index(T, coords):
      for c in coords:
        if not declared(c, T, env):
          errors.push("undeclared", c, "in", T)
      return coords

    case Reduction(op, c, body):
      out = check(body, env + [c], errors)
      if c not in out:
        errors.push(c, "not consumed")
      return out - {c}

    case Add(left, right):
      L = check(left, env, errors)
      R = check(right, env, errors)
      for c in R:
        if c not in L:
          errors.push("broadcast", c, "into left")
      for c in L:
        if c not in R:
          errors.push("broadcast", c, "into right")
      return L | R

    case LetDecl(output, T, coords, body):
      check(body, env + coords, errors)
      return coords

    default:
      errors.push("unknown node", expr)
      return {}
```

Walk the tree. At each node, ask one question. If wrong, record it. If right, return the coordinate set.

Rules 1 and 2 are the `Index` case: check every coordinate against its tensor's declaration. Rule 3 is the `Add` case: compare left and right coordinate sets, record the differences as broadcasts. Rule 5 is the `LetDecl` case: verify the function body's coordinate set against the declared output. Rule 4—causality—is not in this loop; recurrence checking is a separate pass with its own fifteen lines. The five rules from the Wall map to five cases in the tree walk. Four of them are above.

Lines 11–16 are the broadcast merge. When `Add(left, right)` is checked, every coordinate on the right absent on the left means the left operand must broadcast into it—and vice versa. The `Add` node doesn't need to know what operation it's checking—only that both sides contribute coordinate sets and the broadcast relationship must be recorded.

The complexity is in the details—type inference, pack resolution, error message formatting. The structure is fifteen lines. You can hold the entire thing in your head.

Fifteen lines. Read them again—slowly this time. The `Index` case checks that every coordinate in a bracket exists in its tensor's declaration. The `Add` case compares two coordinate sets and records the broadcast. The `LetDecl` case verifies the body against the declared output. Four of the five rules fit in

ten of those fifteen lines. The Wall—the five rules from Chapter 4 that felt like a detective story—is a tree walk with five cases. A first-year CS student could type it during a lecture.

The gap between "five abstract rules" and "fifteen lines of code" is the gap between notation and implementation. The notation made the rules visible. The implementation makes them mechanical. The names are the bridge.

The fifteen lines are the skeleton. Three additions make them a working compiler. Here are the first two. (Pack resolution — the third — is Chapter 5's subject; you've already seen how the compiler resolves `..left` and `..right` around a named anchor.)

Type Inference: The Simplest Pass

Type inference is the simplest pass in the compiler. You already know how to do it — you just haven't named it.

Look at `3.14 + 0.5`. What is the type? `f32`. You didn't run the program. You didn't consult a type environment. You looked at two float literals, added them in your head, and concluded the result is a float. The compiler does exactly what you did. It just does it systematically, node by node, from leaves to root.

The leaves are the easy part. A literal carries its type — `3.14` is `f32`, `5` is `i32`, `true` is `bool`. You see it instantly. The compiler sees it the same way: the literal node in the IR has a type field. Done.

A variable inherits its type from its declaration. You wrote `let x: [f32; batch, in] = input()`. The `f32` is right there in the source. When the compiler encounters `x` in an expression, it looks up `x`'s declaration and reads the type. No inference. No guessing. The answer was written down.

Now the interior nodes. A binary operation: `left + right`. You know the rule without being taught it: both sides must be the same type, and the result is that type. Float plus float is float. Integer plus float is an error — not because the hardware can't do it, but because the programmer probably didn't mean to mix them. The compiler enforces this. If `left` is `f32` and `right` is `i32`, it reports a type mismatch rather than silently promoting. No implicit coercion. No `1.0 + 2 = 3.0` surprises.

A reduction: `sum[k](body)`. The sum iterates over `k` and accumulates values. The element type doesn't change — summing `f32` values produces an `f32`. Reducing `i32` produces `i32`. The reduction changes the shape (consuming `k`). It doesn't touch the type.

A conditional: `if cond { a } else { b }`. `cond` must be `bool`. `a` and `b` must have the same type. The result is that type.

That's the entire algorithm. Five cases. Walk the tree bottom-up. At each node, look at the children's types. Apply one rule. Propagate upward. When you reach the root, every node has a type.

Walk a real expression to see it in action:

```
let z[b, out] = sum[in](x[b, in] * W[out, in]) + b[out] * 0.5;
```

Start at the leaves. `x` is `f32` (from its declaration). `W` is `f32`. `b` is `f32`. `0.5` is `f32` (literal).

Move up. `x * W`: both `f32`, result `f32`. `sum[in](...)`: element type unchanged, `f32`. On the other branch: `b * 0.5`: both `f32`, result `f32`.

Move up again. `sum_result + scaled_bias`: both `f32`, result `f32`. `z` is `f32`.

Every type in this program is known before a single value is computed. The declarations at the leaves — `x: [f32; ...]`, `W: [f32; ...]` — determine every type in the tree. No `dtype` annotations on operations. No `astype` calls in the generated code. Twelve lines of code. Zero ambiguity.

Error Formatting: The Name in the Message

The checker found an error. What does the programmer see?

A positional error says: `IndexError: dimension 3 out of bounds`. The programmer counts dimensions. Guesses which one. Guesses why.

A named error says:

```
error[E004]: coordinate not declared
  → src/model.eln:42:18
   |
42 |     let result = sum[class](logits[batch, class]);
   |                      ^^^^^ coordinate `class` not found in `logits`
   |
   = logits declared with coordinates: {batch, feature}
   = did you mean `feature`?
```

Four parts. Each has a job.

The error code (`E004`). Assign each of the five rules a number. The programmer who has seen `E001` (undeclared coordinate) three times recognizes it on the fourth. The code is for muscle memory.

The location (`src/model.eln:42:18`). File, line, column. The caret (`^^^^^`) points to the exact span — `sum[class]`, five characters, underlined. The programmer's eye lands where the compiler's eye landed.

The message. One sentence. Names the coordinate (`class`), names the tensor (`logits`), states the mismatch. Not "coordinate contract violation." Not "broadcast inconsistency." Plain words: "coordinate `class` not found in `logits`."

The context. `logits declared with coordinates: {batch, feature}`. The compiler shows what it expected and what it found. The difference is visible: `class` vs `feature`. The programmer sees the typo instantly.

Now a broadcast error:

```
error[E003]: broadcast inconsistency
  → src/model.eln:67:10
   |
67 |     let out[b, c, h, w] = x[b, c, h, w] + scale[c];
   |                                           ^^^^^^^^ coordinate `h` required by output but m
   |
   = output coordinates: {b, c, h, w}
   = scale coordinates: {c}
   = missing: {b, h, w}
   = scale broadcasts over {b, h, w} - is this intended?
```

The missing coordinates are enumerated. The question at the end — "is this intended?" — is not rhetorical. It is the broadcast self-audit from Chapter 4, framed by the compiler. The programmer reads the missing set and decides: yes, `scale` should be silent on `b`, `h`, and `w`, or no, `scale` should also depend on `h`. The compiler cannot answer the semantic question. It can only ask it. But the question has a place to land — the missing coordinate set, computed by set subtraction, printed with names.

The error formatter is forty lines of code. A third of it is the caret pointing. The rest is formatting the coordinate sets, looking up the declaration, and suggesting the nearest name. The principle: every error

names the coordinate, shows the context, and asks a question the programmer can answer. The compiler is not the authority. It is the messenger. The names are the message.

If you implement these fifteen lines and add the three additions — type inference (twelve lines), pack resolution (Chapter 5's algorithm), and error formatting (forty lines) — you have a working coordinate checker. Total: roughly three hundred lines. The fifteen lines above are the skeleton. The three additions are the flesh. The names are the firewood.

The Checker in Action: A Chained Reduction

The fifteen-line checker above is abstract. Here it is on a concrete expression — a nested reduction drawn from the Einlang standard library of examples:

```
let W[i in 0..4, k in 0..2] = i as f32 + k as f32;
let X[k in 0..2, h in 0..3] = k as f32 * 10.0 + h as f32;
let nested = max[h](sum[k](W[i, k] * X[k, h]));
```

`W` has coordinate set `{i, k}`. `X` has `{k, h}`. The expression `max[h](sum[k](W[i, k] * X[k, h]))` has two reductions, one inside the other. Watch the 15-line checker walk it.

Step into the inner reduction: `sum[k](W[i, k] * X[k, h])`.

The body is the product `W[i, k] * X[k, h]`. The checker reaches the `*` node — it's an `Add`-like binary operation (the checker treats `*` the same as `+`: compare sets, record broadcasts, return union). Left operand `W[i, k]` has set `{i, k}`. Right operand `X[k, h]` has set `{k, h}`. The checker compares: `k` is in both — shared, contracting. `i` is only on the left — broadcast recorded. `h` is only on the right — broadcast recorded. Union: `{i, k, h}`.

The `Reduction` case fires for `sum[k]`. The body returned `{i, k, h}`. The reduction consumes `k`: `{i, k, h} - {k} = {i, h}`. This is the coordinate set of the inner sum.

Step into the outer reduction: `max[h](inner_result)`.

The inner result carries `{i, h}`. The `Reduction` case fires again. The reduction consumes `h`: `{i, h} - {h} = {i}`. This is the coordinate set of the full expression.

Verification. The declared output `nested` has coordinate `i`. The `LetDecl` case compares: does the body's coordinate set `{i}` match the declared output? `{i} = {i}`. Yes. Four cases — Index, BinaryOp, Reduction, LetDecl — processed the entire nested expression. Three lines of code each. The program passes every check.

Walk the trace backward. `k` was consumed by the inner sum. `h` was consumed by the outer max. `i` is the sole survivor — it was present on `W` from the start, passed through both reductions untouched, and emerged as the output coordinate. The expression computes: for each `i`, take the maximum over `h` of the sum over `k` of `W[i, k] * X[k, h]`. The coordinate names recorded exactly what happened. The checker verified it mechanically. The names were both the specification and the proof.

Now ask: what would a positional checker verify? `W` has shape `(4, 2)`. `X` has shape `(2, 3)`. `W @ X` produces `(4, 3)`. `np.max(..., axis=1)` produces `(4,)`. The shapes match. The positional checker is silent. It has nothing to verify beyond shape consistency.

But the shapes are consistent *because* the coordinate names are consistent. `k` appears in both `W` and `X` — that's why `axis=1` of `(4, 2)` and `axis=0` of `(2, 3)` can contract. `h` is the coordinate to maximize over — that's why `axis=1` of `(4, 3)` is the right axis. The positional code is correct because the named

structure is correct. The names were the proof. The positions were the consequence of the proof. But only the proof survives audit.

A second pattern: chained independent reductions. Consider a different expression from the same example file — two reductions over the same coordinate name, on different tensors, combined element-wise:

```
let A[k in 0..2, n in 0..3] = (10 * k + n) as f32;
let B[n in 0..3, j in 0..4] = (n * 100 + j) as f32;
let chained = argmax[n](A[k, n]) as f32 * max[n](B[n, j]);
```

Walk the `*` node. Left: `argmax[n](A[k, n])` — body returns `{k, n}`, reduction subtracts `n`, result `{k}`. Right: `max[n](B[n, j])` — body returns `{n, j}`, reduction subtracts `n`, result `{j}`. BinaryOp merge: `{k}  {j} = {k, j}`. Broadcast recorded for both directions — `k` absent from right, `j` absent from left.

The same coordinate `n` is consumed by two different reductions on two different tensors. The checker verifies that `n` exists on both `A` and `B`. It records that the argmax result broadcasts over `j` and the max result broadcasts over `k`. The element-wise multiplication of a `(2,)` tensor and a `(4,)` tensor produces a `(2, 4)` tensor. The broadcast relationship is explicit — computed by set subtraction, not inferred from shapes.

In a positional framework: `argmax(A, axis=1)` produces `(2,)`. `max(B, axis=0)` produces `(4,)`. Multiply them and NumPy broadcasts automatically. But nothing in the positional code records *why* the broadcast works — that `n` names the same domain in both tensors, that the domain sizes match, that the multiplication is semantically valid. The shapes happen to broadcast. The names guarantee that they *should*.

The fifteen-line checker traces both patterns — nested reductions and chained independent reductions — with the same four cases. Index introduces coordinates. Reduction subtracts. BinaryOp merges with broadcast recording. LetDecl verifies the output. The structure is identical. The complexity of "nested vs. chained" is not in the checker. It is in the expression. The checker doesn't care whether reductions are nested or chained — it subtracts one coordinate set at a time, in tree order, and the result falls out.

You wrote `class`. Five characters. They survived parsing, analysis, lowering—each stage asking a question that a number could not. At the end, they became `axis=1` and were burned. But the burn was correct because the name was verified.

The positional alternative is `dim=-1`: three keystrokes that enable zero checks. The ratio is the distance between correct-by-construction and correct-by-coincidence.

Consume—that word has appeared in every chapter since Chapter 2. A reduction consumes a coordinate. A broadcast consumes silence. A gradient consumes the broadcast set. And now the compiler consumes the name itself. `class` goes in. `axis=1` comes out. A good abstraction is good firewood. Its beauty is not in its surface—but in the light the flame casts when it burns.

The checker has one more pass to learn: range inference, shape analysis, and lowering. After that, the names face the real test.

Chapter 10 · The Name in the Mirror

> "A problem well stated is a problem half solved."
>
> — Charles Kettering

Construction · Range inference, shape analysis, type propagation, constraint solving, and execution strategies

Your program passed all five rules. You run it. It crashes.

```
index out of bounds: 32
```

The index `i` was declared. The checker saw it. But the checker only asks "is this name declared?"—not "what values can it take?" The declaration said `i in 0..N`, and the compiler never computed `N`. It had no machinery to compute `N`. The name was verified. The range was not.

Now the compiler has to actually answer the question it skipped: what does each name *imply*? Before it can lower `class` into `axis=1` or `batch` into a loop, it must answer three questions about every expression in the tree:

1. **Range.** `i` was declared — but what values can it take? 0 to 10? 0 to `batch_size`?
2. **Shape.** Which coordinates survive? What is the output shape?
3. **Type.** Is this expression `f32` or `f64`? Integer or float?

The 15-line checker asks "is this name declared?" These three questions ask "what does this name imply?" The checker verifies. The analyzer infers. Together they form the complete compiler frontend—one that can take a program without a single runtime value and determine the shape of every tensor in it, so `i in 0..32` is known before `data[i]` is ever evaluated.

Range Inference

You write:

```
let result[i] = data[i] * 2.0;
```

`data` has shape `[N]`. You never wrote `i`'s range. The compiler infers it.

Before reading how the compiler does it, do it by hand. `data` has shape `[N]`. What values can `i` take? You know instantly: `0` to `N-1`. You didn't run the program. You looked at `data[i]` and `data`'s shape. The compiler does the same thing.

The algorithm: find every array access where the variable appears, back-compute a bound from each array's declared shape, and intersect. Here, `data[i]` is the only access. `data` has shape `[N]`, so `i < N`. Range: `i in [0, N)`.

Now a two-access case:

```
let result[i] = arr[i] + arr[i + 1];
```

`arr` has shape `[N]`. Two accesses:

- `arr[i]` → `i < N` → `i in [0, N)`
- `arr[i + 1]` → `i + 1 < N` → `i in [0, N - 1)`

Intersection: `i in [0, N - 1)`. The compiler inferred that `i` cannot reach `N - 1` because `arr[i + 1]` would be out of bounds. No annotation. No `range` declaration. The name `i`, appearing in two positions, carries enough information.

Now a case with two different arrays:

```
let result[i] = x[i] + y[2 * i];
```

`x` has shape `[N]`. `y` has shape `[M]`.

- `x[i]` → `i < N` → `i in [0, N)`
- `y[2 * i]` → `2 * i < M` → `i < ceil(M / 2)` → `i in [0, ceil(M / 2))`

Intersection: `i in [0, min(N, ceil(M / 2)))`.

The compiler doesn't know `N` or `M` at compile time — they may be runtime values. But it knows the *relationship*: `i`'s range is bounded by both. The lowering pass will emit `for i in range(min(N, ceil(M / 2)))`. The bound is symbolic. The check is structural.

When Inference Fails

Inference needs at least one array access where the variable appears as a simple index (not inside a pack, not in an arithmetic expression that the constraint solver can't invert). If you write:

```
let result[i] = f(i);
```

where `f` is an opaque function, the compiler has no array shape to back-compute from. The variable `i` is declared but unconstrained. The compiler reports:

```
error: cannot infer range for index variable `i`
  → no array access found where `i` appears as a direct index
  → declare the range explicitly: `i in 0..N`
```

This is not a limitation of the inference algorithm. It is a boundary of the notation: index variables exist to index into arrays. If a variable never indexes into an array, the compiler asks what it *is* indexing into.

Packs and Ranges

Packs don't change the inference. When the compiler sees:

```
let result[..b, i] = x[..b, i] + 1.0;
```

it resolves `..b` first — anchoring on `i` to determine which dimensions `..b` absorbs. Then it infers `i`'s range from `x`'s shape at the position where `i` appears. The pack is a group of dimensions. The index variable is a single position. They don't interfere.

The Principle

Range inference works because of the coordinate-name philosophy. Every index variable appears literally in array index expressions. Every array has a declared shape. The compiler traces the variable through the accesses, reads the shapes, and intersects. No data needed. No execution needed. The names carry the constraints.

You can do this by hand. When you write `let result[i] = arr[i] + arr[i + 1]`, you know `i` can't reach `N - 1`. You don't run the program to know it. You look at `arr[i + 1]` and mentally subtract one from the bound. The compiler does the same thing — systematically, for every variable, without getting tired.

Shape Analysis

Once every index variable has a range, shapes follow mechanically. The shape of a tensor is the list of its surviving coordinates, each replaced by its range length.

```
let C[i, j] = sum[k](A[i, k] * B[k, j]);
```

`A` has shape `[N, K]`. `B` has shape `[K, M]`. The reduction `sum[k]` consumes `k`. Surviving coordinates: `{i, j}`. Ranges: `i in [0, N)`, `j in [0, M)`. Output shape: `[N, M]`.

The compiler traces this through the IR tree. Each node has a coordinate set. Index nodes introduce coordinates. Reductions subtract them. The output shape is the outermost node's coordinate set, with each coordinate replaced by its range.

A shape mismatch is caught here. If the user declares:

```
let C[i, j] = sum[k](A[i, k] * B[k, j]);  // C declared with [i, j]
```

but the body somehow loses `j` — say, through a mistaken reduction — the compiler reports:

```
error: declared output coordinates {i, j} do not match body coordinates {i}
  → missing coordinate: j
```

This is Rule 5 from Chapter 9, applied to shapes. The declaration says one thing. The body says another. The compiler reports the difference.

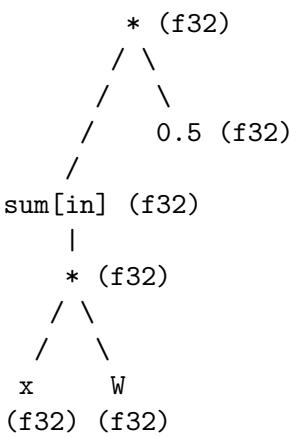
Type Propagation

Types flow from leaves to root. It is the simplest pass in the compiler — one traversal of the tree, bottom-up, no backtracking.

Consider a fragment of a linear layer:

```
let x: [f32; batch, in] = input();
let W: [f32; out, in] = param();
let scaled[batch, out] = sum[in](x[batch, in] * W[out, in]) * 0.5;
```

Walk the tree of `sum[in](x[batch, in] * W[out, in]) * 0.5`:



`x` is `f32` — declared. `W` is `f32` — declared. `x * W` — both `f32`, result `f32`. `sum[in](...)` — reduces over `in`, element type unchanged, `f32`. `0.5` — literal, `f32`. `sum_result * 0.5` — both `f32`, result `f32`. The entire expression is `f32`. Done.

Every literal has a type. `3.14` is `f32`. `5` is `i32`. Every variable inherits its type from its declaration. Every operation propagates the types of its operands upward. The only check: index variables used in arithmetic (`i + 1`) must be integer-typed. If a float variable appears as an index, the compiler reports it.

Types and shapes are orthogonal. A tensor's type says what the elements are. Its shape says how many elements there are and how they are arranged. The type propagates through the element values. The shape propagates through the coordinate structure. They meet only at the leaves — the array declarations that specify both.

The pass is twelve lines of code. But without it, the compiler cannot generate correct code — it wouldn't know whether to emit `np.float32` or `np.float64`. The simplest pass. Also the one that would silently corrupt every number in the program if it were wrong.

The Constraint Solver

You write a convolution:

```
let output[b, oc, oh, ow] = sum[ic, kh, kw](
    input[b, ic, oh + kh, ow + kw] * weight[oc, ic, kh, kw]
);
```

The ranges are known: `oh` up to `output_height`, `kh` up to `kernel_height`. But `oh + kh` indexes into `input` — and `input` has `input_height`. Is `oh + kh` always less than `input_height`?

You check by hand. The worst case: `oh = output_height - 1`, `kh = kernel_height - 1`. Sum: `output_height + kernel_height - 2`. If that's less than `input_height`, safe. If not, out of bounds. Simple arithmetic.

The compiler does the same check — automatically, for every index expression in the program. It is a miniature theorem prover, but it only knows one trick: pattern-matching on the expression tree.

Take `(i * 2 + offset) < N`. The solver walks the tree:

1. **Top level is addition.** `(i * 2) + offset < N`. Subtract offset: `i * 2 < N - offset`.
2. **Recurse into multiplication.** `i * 2 < N - offset`. Divide by 2 (with ceiling): `i < ceil((N - offset) / 2)`.
3. **Base case.** `i < bound`. Range is `[0, ceil(bound))`.

Three rewrites. The solver chains them mechanically. Addition → subtract. Multiplication → divide. Division → multiply. The ceiling is `(a + b - 1) / b` at the IR level, so it works for any input size.

Now ask: what can the solver *not* prove?

Negative coefficients. The solver reads `k - target < bound`. To isolate `target`, it would need to add `target` to both sides and subtract `bound` — producing a lower bound. The solver only computes upper bounds. `k - target < bound` is recognized as unsolvable by the current pass. The pattern is there. The solver sees it. It says: *I can't.*

Modulo. `i % 2`. The solver stares at it. Modulo is not addition, not multiplication, not division. The solver has no rewrite rule for it. The range of `i % 2` is `{0, 1}`, but plugging that into `data[i % 2]` requires reasoning about value sets, not arithmetic bounds. Different kind of solver. Different pass. Not this one.

Nonlinear. `i * j`, `i ** 2`. Products of two variables. The solver doesn't handle them — and almost nothing in a tensor program asks it to. When these appear, the programmer is doing something unusual, and the compiler's silence is the right response: *declare the range yourself.*

When the solver fails:

```
error: cannot prove index expression `oh + kh` stays within bounds of `input`
  → constraint: oh + kh < input_height
  → oh in [0, output_height), kh in [0, kernel_height)
  → try declaring the range explicitly or simplifying the index expression
```

The error names every variable. Shows every range. States the constraint. The programmer reads it and knows exactly what the compiler couldn't prove — and exactly what to fix.

This is the difference between a named error and a positional one. The positional equivalent is `IndexError: dimension 3 out of bounds`. Which dimension is 3? What was supposed to be in bounds? The number answers neither question. The names answer both.

The solver is not a general-purpose theorem prover. It is a pattern matcher for the expressions that appear in real tensor indices — linear combinations with positive coefficients, the arithmetic of stencils and convolutions and pooling windows. Every expression it proves safe is one less silent bug. Every expression it cannot prove is flagged, with names, before the program runs.

The Solver in the Wild: Strided Convolution in Whisper

The solver's pattern-matching is not a toy. It handles the expressions that appear in production models. Consider the Whisper speech recognition encoder — a real model that transcribes audio to text. Its second convolution layer downsamples the time axis by a factor of 2:

```
let c2[co in 0..384, t in 0..1500] =
    sum[ci in 0..384, k in 0..3](c1_pad[ci, t * 2 + k] * enc_conv2_w[co, ci, k])
    + enc_conv2_b[co];
```

The novelty is `t * 2 + k`. This is a strided convolution — stride 2, kernel size 3. The output position `t` maps to input position `2*t` (the stride), plus `k` offsets the kernel window (0, 1, 2). The constraint solver must verify that `t * 2 + k < input_length` for all valid `t` and `k`.

The solver walks the expression tree just as it did for `i * 2 + offset`:

1. **Top level is addition.** `t * 2 + k < input_length`. Subtract `k`: `t * 2 < input_length - k`.
2. **Recurse into multiplication.** `t * 2 < input_length - k`. Divide by 2 (with ceiling): `t < ceil((input_length - k) / 2)`.
3. **Base case.** `t < bound`. The worst case is `k = 0` — the largest upper bound on `t`. Range: `t in [0, ceil(input_length / 2))`.

The solver concludes: `t` ranges up to half of `input_length`. The output has 1500 time steps because the input (after padding) has 3002 time steps — `ceil(3002 / 2) = 1501`, and the kernel of size 3 subtracts one more from the valid range. The numbers check out. The compiler proved it without running a single value.

Now look at the compiler's work from the other direction — the input's perspective. `c1_pad` has shape `(384, 3002)`. The index expression `t * 2 + k` reads from it. The ranges are: `t` up to 1500, `k` up to 3. Worst case: `t = 1499`, `k = 3` → `1499 * 2 + 3 = 3001`. The input's last valid index is 3001. The

expression fits. The constraint is satisfied. The compiler checked it by solving for `t` given `k`, then verifying the worst-case bound against `input_length`. Two rewrites. One comparison. Zero runtime checks.

This is the constraint solver earning its keep. The expression `t * 2 + k` is the defining pattern of strided convolution. Every model that downsamples — every convolutional encoder, every U-Net bottleneck, every feature pyramid — uses this arithmetic. The solver handles it with the same rewrite rules it uses for `i * 2 + offset`. Addition subtracts. Multiplication divides. The pattern is the same. The context — a 384-channel convolution in a speech recognition model — is different. The solver doesn't care. It sees the expression tree. It applies its rules. The proof falls out.

And the names survive. When the solver reports that `t` is bounded by `ceil(input_length / 2)`, the bound is in terms of `input_length` — a name, not a number. The relationship between `t`'s range and `input_length` is visible in the error message. The programmer who changes the padding or the stride sees the constraint change. The name `t` carries the context. The name `input_length` carries the bound. The numbers in the generated code — `range(1500)` — are the last trace of the names that proved them.

Before moving on, trace the inference for three more cases. `data` has shape `[N]`.

```
let result[i] = data[i] + data[i + 2];
```

Two accesses. `data[i]` says `i < N`. `data[i + 2]` says `i + 2 < N`, so `i < N - 2`. Intersection: `i` stops at `N - 2`. The compiler subtracts two from the bound without being asked. You did the same thing in your head just now.

```
let result[i] = data[2 * i] + data[2 * i + 1];
```

`2*i < N` and `2*i + 1 < N`. The second is tighter. Intersection: `i < ceil((N-1)/2)`. The compiler divides by two, takes the ceiling, and infers the bound. You didn't run the program. You looked at the index expressions and the shape.

`let result[i] = data[i] + other[3 * i];` — `other` has shape `[M]`.

Two different arrays, two different shapes. `data[i]` gives `i < N`. `other[3*i]` gives `3*i < M` → `i < ceil(M/3)`. Intersection: `i < min(N, ceil(M/3))`. The bound is the tighter of two constraints from two different tensors. The compiler traces the variable through every access, reads every shape, intersects. No annotation. No execution. The names and shapes carry the constraints.

Runtime-Dependent Coordinates

Everything so far assumes ranges are known at compile time. Not all are.

You have a batch of variable-length sequences. `input[batch, seq]`. For sample 0, `seq` has 15 tokens. For sample 1, 23 tokens. For sample 2, 8. The coordinate `seq` means something different in every row.

Try handing this to `np.softmax(logits, axis=1)`. It runs — NumPy doesn't care that row 0 has 15 elements and row 2 has 8. But `axis=1` normalizes over the whole axis, including padding zeros. Your model now attends to nothing. The loss drops — it learned to predict padding. You find the bug three days later.

The compiler saw this coming. `seq` is *ragged* — its extent depends on `batch`. No single `axis=` integer captures that. The compiler cannot emit `np.softmax(logits, axis=1)`. It must emit a loop:

```
for b in range(batch):
    seq_len = lengths[b]
    row = logits[b, :seq_len]
    m = np.max(row)
    e = np.exp(row - m)
    result[b, :seq_len] = e / np.sum(e)
```

Same coordinate. Different strategy. The compiler chose it — not the programmer.

Three kinds of runtime-dependence trigger this:

Ragged dimensions. A coordinate's extent varies with another. Sequence lengths. Graph node counts. Image resolutions in a batch. The compiler verifies the coordinate exists and is used consistently. It cannot lower it to a constant. It emits a dynamic loop, per element, with the correct bound each time.

Sparse coordinates. Not every position carries data. A `(batch, class)` logit matrix where only a subset of classes are valid per sample. The `where` clause from Chapter 2 records the mask; the compiler emits it alongside the loop. The coordinate is named. The mask is named. The relationship between them is visible in the source.

Data-dependent shapes. The output shape depends on the data, not just the input shapes. `top_k` returns a variable number of elements. Beam search expands dynamically. The compiler cannot know the output shape at compile time because the output shape is the answer.

Every tensor compiler faces this boundary, named or positional. The positional compiler says: "axis 1 has dynamic extent." Axis 1 might be `seq`, `class`, `head`, or `feature`. The number doesn't know. The name does. `seq` is ragged. The error names the coordinate. The strategy names the coordinate. The loop bound reads the coordinate's runtime length. The name survives analysis and lowering — it is the thread connecting the static check to the dynamic execution.

Execution Strategies

The compiler has ranges. It has shapes. It knows which coordinates are static and which are ragged. Now it must decide: how to execute.

Three doors.

Door 1: Vectorized. Every coordinate has a static range. No ragged dimensions. No data-dependent shapes. The compiler lowers every name to an integer, emits a single NumPy call, and walks away.

```
m = np.max(logits, axis=1, keepdims=True)
e = np.exp(logits - m)
return e / np.sum(e, axis=1, keepdims=True)
```

`class` → `axis=1`. `keepdims=True` — not written by the programmer, deduced by the compiler from the broadcast recorded during analysis. The generated code is identical to what you would write by hand, if you were careful.

Door 2: Scalar. At least one coordinate is ragged or data-dependent. The compiler emits Python loops — one per dynamic coordinate, with the bound read at runtime.

```
for b in range(batch):
    seq_len = lengths[b]
    row = logits[b, :seq_len]
```

```
    m = np.max(row)
    e = np.exp(row - m)
    result[b, :seq_len] = e / np.sum(e)
```

`seq` is ragged. The compiler knew this — it recorded `seq` as runtime-dependent during analysis. It did not lower `seq` to an integer. It emitted a loop with `seq_len` as a dynamic bound. Same coordinate name. Different execution path.

Door 3: Hybrid. Three static coordinates and one ragged one, in the same operation. The compiler vectorizes over the static three and loops over the ragged one. The names tell it which is which. `batch`, `head`, and `d` get `axis=` integers. `seq` gets a loop.

The programmer did not choose the strategy. The compiler did. The programmer wrote the names. The compiler read them, classified each coordinate, and chose the door. Vectorized where possible. Scalar where necessary. The choice is per operation, not per program — a single function can mix static and dynamic coordinates, and the compiler handles each one correctly.

This is lowering. Names go in. Execution comes out.

Three doors, one compiler. The programmer never writes `axis=0` or `axis=-1`. The programmer never chooses between a vectorized loop and a scalar loop. The programmer writes coordinates. The compiler reads them, classifies them, and picks the door. Static coordinates become fused dimensions. Dynamic coordinates become guarded loops. Ragged coordinates become runtime bounds. The same name `seq` can be static in one function and ragged in another. The compiler handles each case per operation, not per program.

The three doors are not three compiler flags. They are three consequences of one fact: the coordinate `seq` has a range, and the range is known or unknown at compile time. The compiler checks that fact during range inference. The execution strategy falls out. Correctness first. Performance follows.

What the Mirror Shows

The 15-line checker from Chapter 9 verified names. The passes in this chapter infer what those names imply. Range. Shape. Type. Proof that the arithmetic stays in bounds.

Together, the checker and the analyzer form a complete frontend. Source goes in. Names are checked. Ranges are inferred. Shapes are derived. Types are propagated. Index arithmetic is proved safe. And then — only then — lowering burns the names into numbers.

The mirror metaphor is not decorative. Analysis is reflection. The compiler holds the program up to itself and asks: what do these names imply that the programmer did not state? The answer is already in the names. The compiler makes it explicit.

You can do every pass by hand. You did range inference at the start of Chapter 9 when you traced `batch` and `class` through softmax and sum. You did shape analysis when you determined that `y` has shape `[batch]`. You did type propagation without thinking about it — `logits` is `f32`, so everything downstream is `f32`. The compiler does what you did, systematically, for every line.

Follow the Name

Take one program. Watch every pass operate on it.

```
let x: [f32; batch, in] = input();
let W: [f32; out, in] = param();
let b: [f32; out] = param();
let logits[batch, out] = sum[in](x[batch, in] * W[out, in]) + b[out];
let activated[batch, out] = relu(logits[batch, out]);
```

Five lines. One linear layer. Now follow `batch` through the compiler.

Checker. Rule 1 — every coordinate in an index list exists on the tensor. `x` has `{batch, in}`, so `x[batch, in]` passes. `W` has `{out, in}`, so `W[out, in]` passes. `b[out]` passes. Rule 2 — the reduction coordinate `in` appears in both `x[batch, in]` and `W[out, in]`. Passes. Rule 3 — `b[out]` omits `batch`. Broadcast recorded: `b` broadcasts over `batch`. Rule 5 — `sum[in]` consumes `in`. Output `logits` has `{batch, out}`. `relu` preserves coordinates. Output `activated` has `{batch, out}`. Five checks. Zero errors. The program is consistent.

Range inference. `batch` appears only in index lists, never as a bound. Its range is inferred from the input — if `x` is declared with `batch in 0..32`, then `batch in [0, 32)`. `in` and `out` are inferred the same way from `W`'s declaration. No index arithmetic, so the constraint solver has nothing to prove. Trivial pass.

Shape analysis. `sum[in]` consumes `in`. Surviving coordinates: `{batch, out}`. Output shape: `[batch_size, out_size]`. `relu` preserves coordinates — output shape unchanged. `activated` has shape `[batch_size, out_size]`. Pass complete.

Type propagation. `x` is `f32`. `W` is `f32`. `b` is `f32`. `x * W` is `f32`. `sum[in](...)` is `f32`. `+ b` is `f32`. `relu(...)` is `f32`. Every tensor in this program is `f32`. Pass complete.

Lowering. Three named coordinates. Three integers: - `batch` → axis=0. Static range. Vectorized. - `in` → axis=1, consumed by sum. The sum becomes `np.sum(..., axis=1)`. - `out` → axis=1 after reduction (was axis=1 in W, becomes axis=1 in logits). Static range. Vectorized.

The broadcast `b[out] + ...` is lowered with `keepdims` logic: `b` has shape `(out_size,)`, the sum result has shape `(batch_size, out_size)`. NumPy broadcasts automatically. The compiler recorded the broadcast during analysis; the lowering pass trusts the recording.

Generated code:

```
logits = np.sum(x * W.T, axis=1) + b
activated = np.maximum(logits, 0)
```

No `batch`. No `out`. No `in`. The names are gone. Every one was checked before it burned.

Now add a ragged coordinate. Same program, but `batch` has dynamic extent — variable batch size per call. The compiler reclassifies `batch` as runtime-dependent. Lowering switches from vectorized to scalar:

```
activated = np.zeros((batch_size, out_size), dtype=np.float32)
for b_idx in range(batch_size):
    logits_b = np.sum(x[b_idx] * W.T, axis=0) + b
    activated[b_idx] = np.maximum(logits_b, 0)
```

Same names. Same checks. Different execution strategy. The programmer didn't choose the loop. The coordinate `batch` being runtime-dependent chose it.

This is the compiler's contract with the programmer: *name the coordinates, declare the shapes, write the computation. The compiler checks consistency, infers ranges, derives shapes, propagates types, proves bounds, and chooses the execution strategy. Names go in. Correct execution comes out.*

Lowering: The Algorithm

Lowering is the final pass. Names have been checked, ranges inferred, shapes derived, types propagated, bounds proved. What remains: translate every named operation into an operation on integers that a numerical backend can execute.

The algorithm walks the IR tree one more time. At each node, it asks: given the axis assignments for this tensor, what integer operation does this named operation become?

Step 1: Assign positions. Every tensor in the IR carries its layout — an ordered list of coordinate names, built during shape analysis. The lowering pass reads each layout and numbers the coordinates left to right:

```
logits layout: [batch, class]  →  batch→0, class→1
```

This mapping is local to each tensor. If a different tensor has layout `[class, batch]`, its mapping is `class→0, batch→1`. The name is the invariant. The position is per-tensor.

Step 2: Lower index expressions. `(index logits (batch class))` becomes: read `logits` at all positions — axis 0 and axis 1. In NumPy, the slice `[:, :]`. The compiler doesn't emit slices for full reads; it emits the tensor name. The index node confirms that every requested coordinate exists and records the access pattern for later steps.

Step 3: Lower reductions. `(reduction sum (class) body)` becomes `np.sum(body, axis=class_pos)`. The compiler looks up `class` in the body tensor's layout. If the body has layout `[batch, class]`, `class` is at position 1. The reduction becomes `axis=1`. If `keepdims` is needed — because a downstream operation expects `class` to still exist as a dimension of size 1 — the compiler adds it. The decision is not a flag in the source. It is a requirement of the coordinate structure: if a later operation reads `class` from the reduction's output, the reduction must preserve the dimension. If nothing downstream reads `class`, `keepdims` is omitted.

Step 4: Lower broadcasts. At a binary operation like `logits - max_result`, the compiler compares the operand layouts. `logits` has `[batch, class]`. `max_result` has `[batch]` — `class` was consumed by the reduction that produced it. The coordinate sets differ: `{class}` is missing. The compiler records the broadcast. In the generated code, the broadcast becomes NumPy's automatic shape broadcasting — `logits` has shape `(B, C)`, `max_result` has shape `(B,)`, NumPy broadcasts `(B,) → (B, C)` automatically. If the missing coordinate is not the last one, the compiler inserts a `reshape` or `expand_dims` to align the dimensions.

The compiler does not rely on NumPy's broadcasting rules. It relies on its own coordinate set subtraction — the same subtraction from Chapter 2. The coordinate sets tell it which axes are missing. NumPy's broadcasting is the implementation mechanism, not the specification. The specification is the coordinate sets.

Step 5: Lower index arithmetic. `input[b, ic, oh + kh, ow + kw]` becomes a slice with computed offsets. The compiler emits loops over `kh` and `kw` — the reduction coordinates — and computes the input indices as `oh + kh`, `ow + kw` within each loop iteration. If the ranges are static and small, the loop is unrolled. If they are dynamic, the loop is emitted as written. The index expression `oh + kh` survives lowering as an expression, not as a precomputed offset. The generated code computes it at runtime with the correct bounds.

Step 6: Emit. Walk the lowered tree. Emit NumPy operations for vectorized coordinates. Emit Python `for` loops for ragged coordinates. Emit `if` guards for `where` clauses. The emission is a recursive function that switches on the node type and writes the corresponding Python text.

The six steps together are:

```
lower(ir_node, layout_map):
  match ir_node:
    Index(tensor, coords)     → record access, return tensor name
    Reduction(op, c, body)    → "np.{op}({lower(body)}, axis={layout_map[c]})"
    BinaryOp(op, left, right) → record broadcast if layouts differ,
                                    return "{lower(left)} {op} {lower(right)}"
    LetDecl(name, body)       → "{name} = {lower(body)}"
```

Four cases. The actual implementation has a few more — `Where`, `If`, function calls — but the structure is the same. Walk the tree. Look up the coordinate's position. Emit the integer.

The lowering pass is the point where names die. After this pass, the IR contains no `class`, no `batch`, no `..s`. Every name has become an integer. The integers are correct because the names were checked. The fire is lit. The light is the only trace of the name that remains.

Return to the Transformer

The compiler takes the attention expression and lowers it. Here is what it produces:

```
// lowered IR (S-expression form)
(red sum seq_k
  (mul (index weights (head seq_q seq_k))
       (index V       (head seq_k  d))))
→ einsum string: "hqk,hkd->hqd"
→ axis map: {head: 0, seq_q: 1, seq_k: 2, d: 3}
```

The name `seq_k` appears in the source. It disappears in the lowered code — replaced by the integer 2. But the compiler knew, at every step, that axis 2 was `seq_k`. It derived the range of `seq_k` from the input tensor's shape. It type-checked the broadcast of `V` over `seq_q` by confirming that `seq_q` does not appear in `V`'s index list. It chose the reduction order — `seq_k` as axis 2, not axis 1 — because the layout map placed it there.

Trace the lowering of `seq_k`:

1. **Parse.** The source `sum[seq_k](...)` becomes a `Reduction` node with coordinate `seq_k`.
2. **Name resolution.** `seq_k` is looked up in the tensor layouts. `weights` declares it at position 2, `V` at position 1. The compiler records the mapping: `seq_k → layout_map["weights"][2], layout_map["V"][1]`.
3. **Range inference.** The compiler traces `weights` to its input and reads the shape. `seq_k` ranges from `0` to `sequence_length`.
4. **Shape analysis.** The reduction consumes `seq_k`, so the output shape drops it. The compiler confirms: output is `(head, seq_q, d)` — one fewer dimension than the operands.
5. **Lowering.** The `Reduction` node emits `axis=layout_map[seq_k]`. The integer `2` replaces the name. The einsum string records the consumption as the absence of `k` on the output side of `hqk,hkd->hqd`.

The names burn away during lowering. What remains is correct because they were there.

Part III asked: can a machine do what you have been doing by hand? Trace a coordinate name from source to integer, checking every contract along the way, never running the program. The answer is a compiler — fifteen lines for the core loop, five check rules pinned to a detective's wall, an S-expression IR

that preserves every name, a lowering pass that burns each name into an integer only after it has been verified. You built it.

But a compiler that checks names is an engineering artifact. The question that opened Chapter 1 was not "can we build this?" It was "when the notation has a place for the fact, does the bug survive?" The compiler is the proof that the checking is mechanical. The remaining question is whether the checking matters — placing Einlang side by side with PyTorch and NumPy on normalization, attention, and physical simulation. Not which notation is better. What each notation makes visible, and what each notation hides.

Part IV · Comparisons & Limits

Chapter 11 · Comparison: Normalization

> "A notation is a tool for thought."
>
> — Kenneth Iverson

Comparisons · LayerNorm, RMSNorm, GroupNorm in two notations

You are two weeks into a Transformer project. LayerNorm is working. You swap in RMSNorm for the memory-efficient run—same `dim=-1`, same shape, loss looks fine. Then you try GroupNorm on the convolutional front-end. `dim=-1` again. The shapes align. The loss descends. Three days later you notice the GroupNorm is normalizing over `channels_per_group` instead of `spatial`. The `dim=-1` that was `feature` in LayerNorm became `channel-group-index` in GroupNorm, silently.

Each normalization normalizes over different coordinates. Each uses a position number to say which one. Switch from one to another, and every `dim` must be audited—because `dim=-1` means `feature` in LayerNorm, `channel` in GroupNorm, and nothing at all in RMSNorm.

Part III built the compiler—the proof that names can be checked mechanically. Part IV asks the practical question: does any of this matter for real code? The next three chapters demonstrate the answer across three domains, each escalating the stakes.

Chapter 11 asks: **does the pattern hold?** Normalization — the simplest skeleton, the fewest coordinates. If names don't earn their keep here, they don't earn it anywhere.

Chapter 12 asks: **what does the pattern reveal?** Attention — where self-attention and cross-attention have identical positional code, and the distinction is in runtime shapes, not in source.

Chapter 13 asks: **what does the pattern prevent?** Physics — the oldest domain, where integer indices have been silently swapping coordinates since Fortran, and the bugs produce plausible-but-wrong results that only a physicist's eye catches.

This chapter takes the first question. Three normalization functions appear below in both PyTorch and Einlang, side by side.

The Normalization Skeleton

LayerNorm, RMSNorm, GroupNorm, and InstanceNorm share a single skeleton: reduce to get statistics, broadcast them back, apply elementwise. The four functions differ only in *which coordinates* the reduction consumes. The PyTorch and Einlang versions are laid side by side, and the coordinate names tell the story.

LayerNorm

Given an input of shape `(batch, seq, feature)`, LayerNorm normalizes across the `feature` dimension for each `(batch, seq)` position independently.

PyTorch:

```python
class LayerNorm(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5):
        super().__init__()
        self.gamma = nn.Parameter(torch.ones(normalized_shape))
        self.beta = nn.Parameter(torch.zeros(normalized_shape))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        return (x - mean) / torch.sqrt(var + self.eps) * self.gamma + self.beta
```

`dim=-1` is correct as long as `feature` is the last dimension. It always is—until a refactoring makes it not. `keepdim=True` is needed so the broadcast aligns; forgetting it produces a silent shape mismatch in the gradient.

Einlang:

```
fn layer_norm[feature](x: [f32; ..b, feature], gamma: [f32; feature], beta: [f32; feature])
    -> [f32; ..b, feature]
{
    let mean[..b] = mean[feature](x[..b, feature]);
    let centered[..b, feature] = x[..b, feature] - mean[..b];
    let var[..b] = mean[feature](centered[..b, feature] ** 2.0);
    (centered[..b, feature] / (var[..b] ** 0.5 + 1e-5)) * gamma[feature] + beta[feature]
}
```

What the Einlang version makes visible: - `mean[feature]` says "I am reducing over `feature`." The name is in the bracket. - `mean[..b]` says "`mean` only has batch dimensions." The broadcast over `feature` is explicit in the subtraction `x[..b, feature] - mean[..b]`—`mean` omits `feature`, so it broadcasts along it. - `gamma[feature]` says "gamma aligns with the feature dimension." The pack `..b` absorbs whatever batch structure exists.

If the input changes from `(batch, seq, feature)` to `(batch, feature, seq)`, the Einlang code still works—`..b` now absorbs `(batch,)` and `feature` is at position 1 instead of 2. The PyTorch code silently normalizes over `seq` instead of `feature`.

RMSNorm

RMSNorm is simpler than LayerNorm: no mean subtraction, just scaling by the root-mean-square.

PyTorch:

```python
class RMSNorm(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5):
        super().__init__()
        self.gamma = nn.Parameter(torch.ones(normalized_shape))
        self.eps = eps

    def forward(self, x):
        rms = torch.sqrt(x.pow(2).mean(dim=-1, keepdim=True) + self.eps)
        return x / rms * self.gamma
```

Einlang:

```
fn rms_norm[feature](x: [f32; ..b, feature], gamma: [f32; feature])
    -> [f32; ..b, feature]
{
    let sq[..b, feature] = x[..b, feature] ** 2.0;
    let ms[..b] = mean[feature](sq[..b, feature]);
    x[..b, feature] / (ms[..b] ** 0.5 + 1e-5) * gamma[feature]
}
```

The skeleton is identical to LayerNorm—reduce over `feature`, broadcast back along it, apply elementwise. The difference is only *which statistics* are computed. In PyTorch, LayerNorm and RMSNorm are different classes with different internal logic but identical `dim=-1` interfaces. The fact that they share a skeleton is invisible in the code. In Einlang, you can overlay the two functions and see that only the body differs—the coordinate contract is the same.

GroupNorm

GroupNorm divides channels into groups and normalizes within each group. This requires splitting the `channel` dimension into `(group, channel_per_group)` and reducing over both `channel_per_group` and the spatial dimensions.

PyTorch:

```
class GroupNorm(nn.Module):
    def __init__(self, num_groups, num_channels, eps=1e-5):
        super().__init__()
        self.num_groups = num_groups
        self.eps = eps
        self.gamma = nn.Parameter(torch.ones(num_channels))
        self.beta = nn.Parameter(torch.zeros(num_channels))

    def forward(self, x):
        N, C, H, W = x.shape
        G = self.num_groups
        x = x.reshape(N, G, C // G, H, W)
        mean = x.mean(dim=(2, 3, 4), keepdim=True)
        var = x.var(dim=(2, 3, 4), keepdim=True, unbiased=False)
        x = (x - mean) / torch.sqrt(var + self.eps)
        x = x.reshape(N, C, H, W)
        return x * self.gamma.view(1, -1, 1, 1) + self.beta.view(1, -1, 1, 1)
```

The reshape-permute-reshape dance is the positional price of grouping. `dim=(2, 3, 4)` means "reduce over channel_per_group, height, and width"—but those positions are only correct after the reshape. The reader must mentally compile the grouping semantics from the reshape chain: `reshape` splits channels into groups, `mean(dim=(2,3,4))` reduces within each group, `reshape` merges them back. The grouping is a manual compilation step, performed by the programmer, invisible in the source.

Before you read the Einlang version, stop and ask: from the PyTorch code alone, which coordinates does `dim=(2, 3, 4)` reduce over? You know because the comment says `N, C, H, W` and you counted positions after the reshape. Now ask: if a temporal dimension is prepended next month, what does

`dim=(2, 3, 4)` reduce over? You can't know without redoing the positional arithmetic. The answer is in the positions. The positions change. The code doesn't tell you.

Einlang:

```
fn group_norm[group, c_in_group, ..s](
    x: [f32; ..b, group, c_in_group, ..s],
    gamma: [f32; group, c_in_group],
    beta: [f32; group, c_in_group]
) -> [f32; ..b, group, c_in_group, ..s]
{
    let mean[..b, group] = mean[c_in_group, ..s](
        x[..b, group, c_in_group, ..s]
    );
    let centered[..b, group, c_in_group, ..s] =
        x[..b, group, c_in_group, ..s] - mean[..b, group];
    let var[..b, group] = mean[c_in_group, ..s](
        centered[..b, group, c_in_group, ..s] ** 2.0
    );
    (centered[..b, group, c_in_group, ..s]
        / (var[..b, group] ** 0.5 + 1e-5))
        * gamma[group, c_in_group] + beta[group, c_in_group]
}
```

What the Einlang version makes visible: - `mean[c_in_group, ..s]` names exactly which coordinates are being reduced. No `dim=(2, 3, 4)` whose meaning depends on a reshape. - `gamma[group, c_in_group]` aligns with two coordinates. No `.view(1, -1, 1, 1)` to manually position the broadcast. The omission of batch and spatial from gamma's brackets is the megaphone at work—gamma speaks on group and c_in_group, stays silent on batch and spatial, and the silence is the broadcast. - No reshape is needed because the coordinates `group` and `c_in_group` are separate from the start. The function signature declares the grouped layout directly.

InstanceNorm: The Fourth Variant

LayerNorm normalizes over `feature`. GroupNorm normalizes over `c_in_group + spatial`. InstanceNorm normalizes over `spatial` alone—one statistic per channel per sample. It is used in style transfer, where the "style" of an image is captured by per-channel statistics.

PyTorch:

```
class InstanceNorm(nn.Module):
    def __init__(self, num_features, eps=1e-5):
        super().__init__()
        self.gamma = nn.Parameter(torch.ones(num_features))
        self.beta = nn.Parameter(torch.zeros(num_features))
        self.eps = eps

    def forward(self, x):
        # x: (N, C, H, W)
        mean = x.mean(dim=(2, 3), keepdim=True)
        var = x.var(dim=(2, 3), keepdim=True, unbiased=False)
```

```
        return (x - mean) / torch.sqrt(var + self.eps) * self.gamma.view(1, -1, 1, 1) + self.beta
```

`dim=(2, 3)` means "reduce over H and W." But how do you know H and W are at positions 2 and 3? Because the comment says `# x: (N, C, H, W)`. If the input has shape `(N, C, L)` for 1D, `dim=(2,)`. If it has shape `(N, T, C, H, W)` for video, `dim=(3, 4)`. The `dim` tuple is a function of the input layout. Change the layout, and every normalization call must be audited.

Einlang:

```
fn instance_norm[..s, channel](x: [f32; ..b, channel, ..s],
    gamma: [f32; channel], beta: [f32; channel])
    -> [f32; ..b, channel, ..s]
{
    let mean[..b, channel] = mean[..s](x[..b, channel, ..s]);
    let centered[..b, channel, ..s] =
        x[..b, channel, ..s] - mean[..b, channel];
    let var[..b, channel] = mean[..s](
        centered[..b, channel, ..s] ** 2.0
    );
    (centered[..b, channel, ..s] / (var[..b, channel] ** 0.5 + 1e-5))
        * gamma[channel] + beta[channel]
}
```

`..s` absorbs however many spatial dimensions there are—1, 2, 3. The reduction bracket `mean[..s]` doesn't change. The skeleton is the same as LayerNorm, RMSNorm, and GroupNorm. Only the coordinate in the bracket differs.

Now overlay all four normalization functions. The differences are exactly which coordinates appear in the reduction bracket:

Function	Reduction bracket	Broadcast params
LayerNorm	`mean[feature]`	`gamma[feature]`
RMSNorm	`mean[feature]`	`gamma[feature]`
InstanceNorm	`mean[..s]`	`gamma[channel]`
GroupNorm	`mean[c_in_group, ..s]`	`gamma[group, c_in_group]`

The body of every function is: reduce to get statistics, subtract-and-divide, scale-and-shift. The reduction bracket is the only structural difference. In the PyTorch versions, this unity is invisible—each function has its own `dim` argument, its own view-reshape chain, its own parameter shape conventions. The skeleton is scattered across four classes.

BatchNorm: Where the Skeleton Breaks

BatchNorm normalizes over the batch dimension. In training, it computes per-feature statistics across the batch. In inference, it uses running averages. The coordinate structure—reduce over `..b`, broadcast back over `feature`—is the same in both modes. The semantic difference is *where the statistics come from*: the current tensor or an accumulated buffer. That distinction is invisible in the positional `dim=0` and in the named `mean[..b]` alike.

```
// Both paths share the same coordinate structure:
let batch_mean[feature] = mean[..b](x[..b, feature]);     // training: fresh
let infer_mean[feature] = running_mean[feature];                  // inference: accumulated
```

The reduction bracket names what is consumed. It does not name whether the statistics are fresh or accumulated. That distinction lives in the data dependency graph, not in the coordinate structure. The coordinate skeleton can only carry so much. The rest is in the code's semantics—and being honest about that boundary is as important as celebrating what names can check.

Tracing a Reshape Bug

Let's trace a reshape bug through its entire life, in both notations. This is the bug that the coordinate audit is designed to catch before it reaches production.

Day 0. A programmer writes GroupNorm for 4D input `(N, C, H, W)`:

```
def group_norm(x, num_groups, gamma, beta, eps=1e-5):
    N, C, H, W = x.shape
    x = x.reshape(N, num_groups, C // num_groups, H, W)
    mean = x.mean(dim=(2, 3, 4), keepdim=True)
    var = x.var(dim=(2, 3, 4), keepdim=True)
    return ((x - mean) / (var + eps).sqrt()).reshape(N, C, H, W) * gamma + beta
```

The `dim=(2,3,4)` tuple means: normalizing over `C//num_groups`, `H`, and `W`. The programmer knows this because they can count: dimension 2 is `C//num_groups`, dimension 3 is `H`, dimension 4 is `W`. The code is correct.

Day 60. A colleague adds temporal dimension for video input. The tensor is now `(N, T, C, H, W)`. The colleague updates the reshape:

```
N, T, C, H, W = x.shape
x = x.reshape(N, T, num_groups, C // num_groups, H, W)
mean = x.mean(dim=(2, 3, 4), keepdim=True)  # BUG
```

`dim=(2,3,4)` now means: normalizing over `num_groups`, `C//num_groups`, and `H`. Wait — `num_groups` at position 2, `C//num_groups` at position 3, `H` at position 4. But `W` is at position 5. And `T` is at position 1. The tuple `(2,3,4)` needs to be `(3,4,5)`. The programmer forgets. The code runs. The shapes match because `keepdim=True` preserves the reduced dimensions. The bug is: GroupNorm is now normalizing over `(num_groups, c_in_group, H)` instead of `(c_in_group, H, W)`. Width is not normalized. Group is normalized — collapsing the grouped structure.

The loss still descends. The model still produces video outputs. But the normalization is wrong. The bug will surface as "the model performs worse on wide videos."

Now replay in Einlang:

```
// Original
fn group_norm[g, c_in_group, ..s](x: [f32; ..b, g, c_in_group, ..s], ...)

// With temporal dimension added - same signature
fn group_norm[g, c_in_group, ..s](x: [f32; ..b, t, g, c_in_group, ..s], ...)
```

The signature absorbs `t` into `..b` (if it's a leading dimension) or `..s` (if temporal is treated as spatial). The reduction bracket `mean[c_in_group, ..s]` doesn't change. The coordinates being reduced are still

named `c_in_group` and `..s`. The position of those coordinates in the tensor layout doesn't matter — the names find them.

The bug doesn't happen. Not because the programmer is smarter. Because the notation doesn't require positional arithmetic. The coordinate names abstract over positions. Adding a dimension changes which positions the coordinates map to, but the reduction bracket still names the same coordinates. No `dim` tuple to update. No reshape chain to re-align.

Pause here. You just watched a real bug trace across two notations. In the PyTorch version, the bug takes three days to surface, survives integration tests, and produces a model that trains but performs worse on wide videos. In the Einlang version, the bug cannot be written—`mean[c_in_group, ..s]` still means all channel-group and spatial dimensions regardless of where `t` is inserted. The difference is not that the Einlang programmer is more careful. The difference is that the notation has a place for the fact "I am normalizing over channel groups and spatial dimensions," and the PyTorch notation encodes that fact as a tuple of positions that silently rot when the layout changes.

What facts in your own codebase are encoded as positional tuples?

The Coordinate Audit

Every normalization function can be audited with three questions, each a specialization of the broadcast self-audit from Chapter 4:

1. **Which coordinates does the reduction consume?** In `mean[feature]`, the consumed coordinate is `feature`. In `mean[c_in_group, ..s]`, the consumed coordinates are `c_in_group` and all spatial dimensions. The reduction bracket names them directly. In a positional `dim=-1` or `dim=(2,3,4)`, the consumed coordinates must be inferred from the layout and the reshape chain.
2. **Which coordinates do the broadcast parameters align with?** In `gamma[feature]`, gamma aligns with `feature`—the same coordinate that was consumed by the reduction. This is the Inversion Rule from Chapter 4: the reduction consumes `feature`, then `gamma` broadcasts back along `feature`. In a positional `.view(1, -1, 1, 1)`, the alignment is encoded in the view shape, which must be reconstructed by the reader.
3. **Does the normalization axis change meaning if the layout changes?** If the input changes from `(batch, feature)` to `(batch, time, feature)`, does `dim=-1` still mean `feature`? In LayerNorm, yes—`feature` is conventionally the last axis. In GroupNorm after a reshape, no—the positions shift and `dim` must be updated. The Einlang versions are stable under layout changes because the coordinate names don't change, only the positions they map to.

Before leaving normalization, read this function — one you have never seen:

```
fn normalize[j, k](x: [f32; ..b, j, k], gamma: [f32; j, k], beta: [f32; j, k]) -> [f32; ..b, j, k
    let m[..b] = mean[j, k](x[..b, j, k]);
    let v[..b] = mean[j, k]((x[..b, j, k] - m[..b]) ** 2.0);
    (x[..b, j, k] - m[..b]) / (v[..b] + 1e-5) ** 0.5 * gamma[j, k] + beta[j, k]
}
```

The reduction bracket says `mean[j, k]` — so it consumes `j` and `k`. The output has `{..b, j, k}` and `gamma` has `{j, k}` — so `gamma` broadcasts over `..b`, the difference of those two sets. If `x` changes from `(batch, j, k)` to `(batch, time, j, k)`, the reduction bracket doesn't change. `j` and `k` are found by name. The positional equivalent `dim=(-2, -1)` would survive if `time` is prepended — but not if `time` lands between `j` and `k`. The named bracket doesn't care where `time` lands.

Three questions, three answers, all visible in the signature without reading the body. That is the audit.

Normalization established the baseline: the skeleton holds across four variants, and the coordinate name absorbs layout changes that would silently corrupt a positional `dim=`. Attention has five coordinates, three architectural variants, and a runtime cache whose correctness depends on which coordinate is concatenated. The question shifts from *does the pattern hold?* to *what does the pattern reveal that positional code cannot say?*

Chapter 12 · Comparison: Attention

> "The difference between the right word and the almost right word is the difference between lightning and a lightning bug."
>
> — Mark Twain

Comparisons · Self-attention, cross-attention, and multi-query attention in two notations

You are writing an encoder-decoder Transformer. During development, source and target sequences happen to have the same length—64 tokens. Self-attention works. Cross-attention works. The code for both is a single positional function: `attention(Q, K, V, mask)`. The shapes match. The loss descends. You ship.

Six weeks later, a configuration change sets source length to 128, target to 64. The code still runs—broadcasting absorbs the shape mismatch. But your model now attends from every target position to every source position *twice*, silently. The BLEU score drops two points. You spend three days tracing the drop to a transposed mask broadcasting along the wrong axis. The Square Matrix Test, first encountered with softmax in Chapter 3, returns with a vengeance.

Here is the root cause: self-attention and cross-attention have identical positional code. Stop and let that sink in. Two operations—different semantics, different gradient flows, different architectural implications—expressed as the exact same Python function. The shapes differ at runtime. The source code does not. When source length equals target length, the two attentions are indistinguishable even at runtime. The coordinate names are different. The code is the same. Only the names can distinguish them.

Chapter 11 showed that the pattern holds for normalization—one reduction axis, four variants, one skeleton. Attention raises the stakes: five coordinates with distinct roles, three architectural variants whose positional code is textually identical, a runtime KV-cache whose correctness depends on which axis is concatenated. The question shifts from *does the pattern hold?* to *what does the pattern reveal that positional code cannot say?*

Scaled Dot-Product Attention: The Skeleton

The core operation:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

In Einlang, the coordinate names tell the story:

```
fn attention[seq_q, seq_k, head, d](
    Q: [f32; ..b, head, seq_q, d],
    K: [f32; ..b, head, seq_k, d],
    V: [f32; ..b, head, seq_k, d]
) -> [f32; ..b, head, seq_q, d]
{
    let scores[..b, head, seq_q, seq_k] =
        sum[d](Q[..b, head, seq_q, d] * K[..b, head, seq_k, d]) / (d ** 0.5);
```

```
    let weights[..b, head, seq_q, seq_k] = softmax[seq_k](scores[..b, head, seq_q, seq_k]);
    sum[seq_k](weights[..b, head, seq_q, seq_k] * V[..b, head, seq_k, d])
}
```

Three coordinates do all the work: `seq_q` (the query source sequence), `seq_k` (the key source sequence), and `d` (the inner dimension that gets contracted). `softmax[seq_k]` normalizes over the key sequence—each query position produces a distribution over key positions.

Self-Attention vs. Cross-Attention

Self-attention uses the same sequence for queries and keys. Cross-attention uses different sequences—queries from the decoder, keys from the encoder.

PyTorch (self-attention):

```
def self_attention(Q, K, V):
    d = Q.shape[-1]
    scores = torch.matmul(Q, K.transpose(-2, -1)) / (d ** 0.5)
    weights = torch.softmax(scores, dim=-1)
    return torch.matmul(weights, V)
```

PyTorch (cross-attention):

```
def cross_attention(Q, K, V):
    d = Q.shape[-1]
    scores = torch.matmul(Q, K.transpose(-2, -1)) / (d ** 0.5)
    weights = torch.softmax(scores, dim=-1)
    return torch.matmul(weights, V)
```

They are identical. Two operations with different gradient flows and different architectural implications — textually identical. The distinction between them — whether `seq_q` equals `seq_k` — is not in the source code. It is in the shapes of the tensors passed at runtime. The notation records nothing.

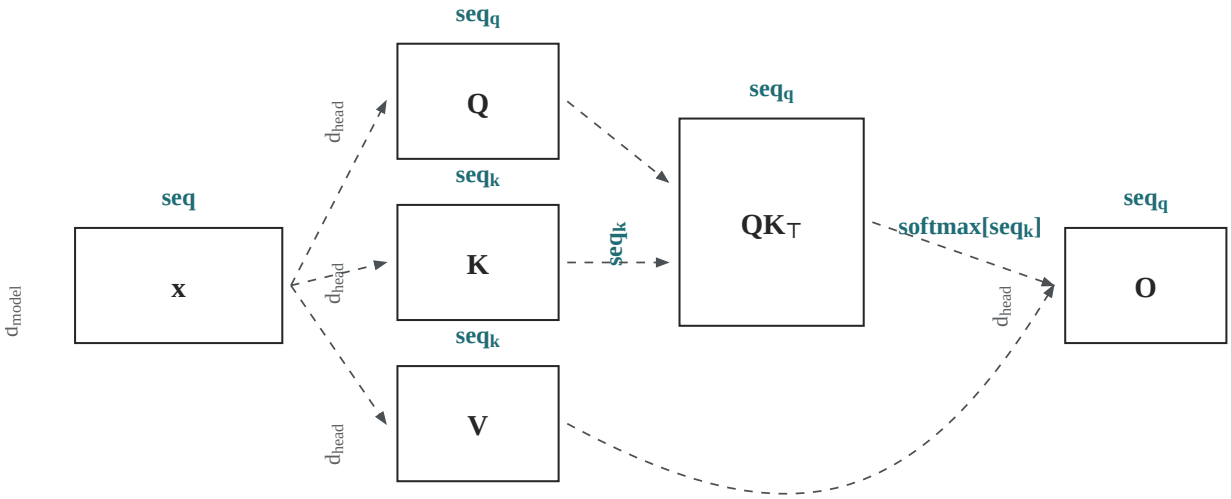
Einlang:

```
// Self-attention: same coordinate for queries and keys
fn self_attention[seq, head, d](Q: [f32; ..b, head, seq, d], K: [f32; ..b, head, seq, d], V: [f32
    -> [f32; ..b, head, seq, d]
{
    attention[seq, seq, head, d](Q, K, V)
}

// Cross-attention: different coordinates for queries and keys
fn cross_attention[seq_q, seq_k, head, d](Q: [f32; ..b, head, seq_q, d], K: [f32; ..b, head, seq_
    -> [f32; ..b, head, seq_q, d]
{
    attention[seq_q, seq_k, head, d](Q, K, V)
}
```

The distinction is in the type signatures. Self-attention uses `seq` for both queries and keys. Cross-attention uses `seq_q` and `seq_k`—two different coordinate names, potentially with different domain sizes. A reader can tell which is which without checking whether the tensors happen to have the same shape.

Here is the attention skeleton with every coordinate named:



Self-attention: $\mathbf{seq}_q = \mathbf{seq}_k$. Cross-attention: $\mathbf{seq}_q \neq \mathbf{seq}_k$. The names tell you which is which.

Figure 10: The attention skeleton. Trace where seq_q and seq_k flow.

Trace the arrows. `seq_q` rides Q into the scores and the output. `seq_k` rides K and V, and is consumed by `softmax[seq_k]` — it does not reach the output. `head` groups the attention heads. `d` is the inner dimension, contracted by `sum[d]` inside the scores. When `seq_q` and `seq_k` name the same sequence, the attention is self. When they name different sequences, it is cross. The diagram records the difference. The positional code for both is identical.

The Square Matrix Test for Attention

When `seq_q == seq_k` and `head == some_other_dimension`, the attention matrix is square. The positional code for self-attention, cross-attention, and a transposed variant are numerically identical. Consider this bug:

```
# Intended: cross-attention from decoder (seq_len=32) to encoder (seq_len=100)
# Bug: accidentally used the same tensor for Q and K
Q = decoder_hidden   # shape (batch, head, 32, d)
K = decoder_hidden   # bug: should be encoder_hidden, shape (batch, head, 100, d)
V = decoder_hidden
output = cross_attention(Q, K, V)  # silently becomes self-attention
```

If `decoder_hidden` and `encoder_hidden` happen to have the same sequence length during development (both 32, or both padded to the same length), this bug is invisible. The shapes match. The loss descends. The model learns—just not what you intended.

In Einlang, `cross_attention[seq_q, seq_k, ...](Q, K, V)` with `Q` and `K` both bound to `decoder_hidden` would trigger a coordinate mismatch if `decoder_hidden` has `seq_q` but not `seq_k` as

its declared coordinate. If both tensors carry both coordinates (because they were declared with different names), the mismatch is caught at the call site.

Multi-Query Attention (MQA)

MQA uses multiple query heads but only one key-value head, broadcasting the KV head across query heads. This is a performance optimization that changes the coordinate structure:

PyTorch:

```
def mqa_attention(Q, K, V):
    # Q: (batch, head_q, seq_q, d)
    # K: (batch, 1, seq_k, d)      -- single KV head
    # V: (batch, 1, seq_k, d)
    d = Q.shape[-1]
    scores = torch.matmul(Q, K.transpose(-2, -1)) / (d ** 0.5)
    weights = torch.softmax(scores, dim=-1)
    return torch.matmul(weights, V)
```

The code is identical to standard attention. The only difference is that `K` has shape `(batch, 1, seq_k, d)` instead of `(batch, head, seq_k, d)`. The `1` broadcasts silently over all query heads. If someone changes the KV projection to output `head_kv` heads instead of `1`, the code still runs—it just produces a different attention pattern. The `1` is a positional convention, not a checked fact.

Einlang:

```
fn mqa_attention[head_q, head_kv, seq_q, seq_k, d](
    Q: [f32; ..b, head_q, seq_q, d],
    K: [f32; ..b, head_kv, seq_k, d],
    V: [f32; ..b, head_kv, seq_k, d]
) -> [f32; ..b, head_q, seq_q, d]
{
    let scores[..b, head_q, head_kv, seq_q, seq_k] =
        sum[d](Q[..b, head_q, seq_q, d] * K[..b, head_kv, seq_k, d]) / (d ** 0.5);
    let scores_merged[..b, head_q, seq_q, seq_k] = mean[head_kv](scores[..b, head_q, head_kv, seq
    let weights[..b, head_q, seq_q, seq_k] = softmax[seq_k](scores_merged[..b, head_q, seq_q, seq
    sum[seq_k](weights[..b, head_q, seq_q, seq_k] * V[..b, head_kv, seq_k, d])
}
```

`head_q` and `head_kv` are different coordinates. The function signature declares that queries have `head_q` heads and keys have `head_kv` heads. When called as MQA, `head_kv` has size 1—but it's a named coordinate, not a silent `1` buried in the shape. If a refactoring changes the KV head count, the coordinate name `head_kv` remains, and it is verified.

`head_kv` is a coordinate whose domain happens to be size 1 in the MQA case. It is not a broadcasting hack. It is a structural fact, visible in the type.

Grouped-Query Attention (GQA): The Middle Ground

Between MHA (`head_q == head_kv`) and MQA (`head_kv == 1`) lies GQA: `head_kv` is a small number, say 4, that divides `head_q`. Each KV head is shared by a group of query heads. This is a coordinate

grouping problem—structurally identical to GroupNorm from Chapter 5.

PyTorch (GQA):

```
def gqa_attention(Q, K, V, num_kv_heads):
    # Q: (batch, head_q, seq_q, d)
    # K: (batch, num_kv_heads, seq_k, d)
    # V: (batch, num_kv_heads, seq_k, d)
    head_q = Q.shape[1]
    # Repeat KV heads: (batch, num_kv_heads, seq_k, d) → (batch, head_q, seq_k, d)
    repeat_factor = head_q // num_kv_heads
    K = K.unsqueeze(2).expand(-1, -1, repeat_factor, -1, -1).reshape(Q.shape)
    V = V.unsqueeze(2).expand(-1, -1, repeat_factor, -1, -1).reshape(Q.shape)
    # ... identical to standard attention from here
    d = Q.shape[-1]
    scores = torch.matmul(Q, K.transpose(-2, -1)) / (d ** 0.5)
    weights = torch.softmax(scores, dim=-1)
    return torch.matmul(weights, V)
```

The grouping logic—unsqueeze, expand, reshape—is spread across two lines. The fact that `head_q` is grouped into `(num_kv_heads, repeat_factor)` is encoded in a reshape chain that the reader must reverse-engineer. If the grouping factor changes, the reshape must be updated. If the layout changes (e.g., `head_q` moves from position 1 to position 2), the unsqueeze and expand must be re-aligned.

Einlang (GQA):

```
fn gqa_attention[head_group, head_kv, seq_q, seq_k, d](
    Q: [f32; ..b, head_group, head_kv, seq_q, d],
    K: [f32; ..b, head_kv, seq_k, d],
    V: [f32; ..b, head_kv, seq_k, d]
) -> [f32; ..b, head_group, head_kv, seq_q, d]
{
    let scores[..b, head_group, head_kv, seq_q, seq_k] =
        sum[d](Q[..b, head_group, head_kv, seq_q, d]
             * K[..b, head_kv, seq_k, d]) / (d ** 0.5);
    let weights[..b, head_group, head_kv, seq_q, seq_k] =
        softmax[seq_k](scores[..b, head_group, head_kv, seq_q, seq_k]);
    sum[seq_k](weights[..b, head_group, head_kv, seq_q, seq_k]
             * V[..b, head_kv, seq_k, d])
}
```

`head_group` and `head_kv` are separate coordinates from the start. No reshape. No expand. No unsqueeze. `K` and `V` are indexed by `head_kv` alone—they broadcast over `head_group` because they omit it. The broadcast is visible in the indexing pattern: `K[..b, head_kv, seq_k, d]` has no `head_group`, while `Q` has both.

Now compare the three variants side by side:

Variant	Query heads	KV heads	Coordinate structure
MHA	`head`	`head`	`head` shared by Q, K, V
GQA	`head_group × head_kv`	`head_kv`	`head_kv` shared; `head_group` only on Q

Variant	Query heads	KV heads	Coordinate structure
MQA	`head_q`	`head_kv` (size 1)	`head_kv` on K, V; `head_q` only on Q

In the Einlang signatures, the difference between the three variants is visible in which coordinates appear on which parameters. In the PyTorch implementations, the difference is buried in reshape chains and the value of `num_kv_heads`. The coordinate names make the architecture visible. The positional code makes it deducible—after counting dimensions and tracing reshapes.

The Attention Coordinate Audit

Every attention variant can be audited with four questions. Ask them of any positional attention code you encounter:

1. **Which coordinate does `softmax` normalize over?** In `softmax(scores, dim=-1)`, the answer is "whatever is last." In `softmax[seq_k](scores)`, the answer is `seq_k`.
2. **Which coordinate distinguishes queries from keys?** In MHA, it's the same (`seq`). In cross-attention, it's different (`seq_q` vs `seq_k`). In positional code, this distinction is in the tensor shapes at runtime. In named code, it's in the function signature.
3. **Which coordinate groups query heads with KV heads?** In GQA, `head_group` groups query heads over a shared KV head. In MQA, `head_kv` has size 1. In MHA, there's no grouping—`head` is the same coordinate on Q and K. The grouping structure is invisible in the positional `matmul`; visible in the named index patterns.
4. **Does the backward pass know what to sum over?** The gradient of attention sums over `seq_q` for `dK` and `dV`, over `seq_k` for `dQ`, and over the head grouping for the KV projection. In positional autodiff, these sums happen silently. In named coordinates, they follow from the coordinate sets—same coordinate set-subtraction rule from Chapter 8 applied to attention.

You don't need Einlang to ask these questions. You need to know that they are the right questions. And the right questions are only visible when the notation has a place for the answers.

Look at the last attention implementation you read. Can all four questions be answered from the code alone?

The KV-Cache Audit

Autoregressive generation uses a KV-cache: keys and values from previous time steps are stored and reused. The cache introduces a new coordinate relationship: `seq_past` (cached) and `seq_new` (current) must be concatenated into a single `seq_k` for the attention computation.

```
# Positional KV-cache
K_full = torch.cat([K_cache, K_new], dim=seq_dim)  # which axis is seq_dim?
V_full = torch.cat([V_cache, V_new], dim=seq_dim)
output = attention(Q_new, K_full, V_full)
```

The `dim` argument to `torch.cat` is a position number. If `K_cache` has shape `(batch, head, past_len, d)` and `K_new` has shape `(batch, head, 1, d)`, then `seq_dim` is `2`. But if the layout is `(batch,`

`past_len, head, d)`, `seq_dim` is `1`. The integer shifts with the layout. Change the layout, audit every `cat` call.

In Einlang, the concatenation axis is named:

```
let K_full[..b, head, seq_k, d] = concat[seq_k](
    K_cache[..b, head, seq_past, d],
    K_new[..b, head, seq_new, d]
);
let output[..b, head, seq_q, d] = attention[head, seq_q, seq_k, d](Q_new, K_full, V_full);
```

`concat[seq_k]` names the concatenation axis. The coordinate `seq_k` absorbs both `seq_past` and `seq_new` into a single coordinate. If the layout changes, the coordinate name doesn't. The `cat` happens over `seq_k` regardless of position.

The audit questions for a KV-cache: 1. Which coordinate does `concat` operate over? (`seq_k`) 2. Which coordinate does the attention reduce over? (`seq_k`—the same coordinate) 3. Does the cached `seq_k` range differ from the new `seq_k` range? (They are different domains, now merged)

The coordinate names make the cache structure visible. The positional `dim=seq_dim` records a position. The named `concat[seq_k]` records an identity.

Flash Attention: The Coordinate Structure Survives Optimization

Flash Attention is a memory-efficient exact attention algorithm that fuses the QK^T matmul, softmax, and PV matmul into a single tiled kernel. It dramatically reduces memory usage by recomputing the softmax statistics in the backward pass rather than storing the full attention matrix. From the user's perspective, the function signature is identical to standard attention. The coordinate structure is unchanged.

This is a demonstration of the principle from Chapter 9: lowering is strategy-independent. The same coordinate structure maps to different execution strategies. Flash Attention is a lowering strategy—a choice of how to execute the computation, not what computation to execute. The coordinate names `seq_q`, `seq_k`, `head`, `d` are identical whether the lowering chooses the standard attention kernel or the Flash Attention kernel.

In a positional API, Flash Attention is a drop-in replacement: replace `attention(Q, K, V)` with `flash_attention(Q, K, V)`. The shapes are the same. The coordinate structure is the same—but only implicitly, in the shapes. In an Einlang API, the coordinate contract is the same—`fn attention[seq_q, seq_k, head, d](...)` for both. The lowering strategy (`standard` vs `flash`) is an annotation, not a signature change:

```
#[strategy(flash)]
let output[..b, head, seq_q, d] = attention[head, seq_q, seq_k, d](Q, K, V);
```

The coordinate names don't change. The contract doesn't change. Only the execution strategy changes. This is the separation that the compiler's lowering pass enables: coordinate contracts define what is computed. Lowering strategies define how it is computed. The names belong to the first. The optimizations belong to the second. They are orthogonal.

When a new attention variant appears—a faster kernel, a sparse pattern, a sliding window—the coordinate contract remains the same. The lowering strategy changes. The names survive the optimization.

The coordinate structure is the invariant. The execution strategy is the variable. Named coordinates record the invariant. Positional notation records neither—it defers both to runtime.

Here is a KV-cache that compiles. Read it once, then look away and try to name the coordinates in order:

```
let K_full[..b, head, seq_k, d] = concat[seq_k](
    K_cache[..b, head, seq_past, d],
    K_new[..b, head, seq_new, d]
);
let output[..b, head, seq_q, d] = attention[head, seq_q, seq_k, d](Q_new, K_full, V_full);
```

The call to `attention` passes `head` as the first coordinate argument and `seq_q` as the second. But the declaration was `fn attention[seq_q, seq_k, head, d]`. The coordinate arguments are in the wrong order. `head` is being passed where `seq_q` is expected. The positional equivalent — passing `dim=0` where `dim=1` was expected — is invisible in the code. The named version has the names in the brackets. The reader can see the mismatch.

Which coordinate does `softmax` normalize over in the buggy call? The compiler maps positionally — the second bracket argument becomes `seq_k` in the body regardless of its name. But the *programmer* wrote `head` in that position. The coordinate name `head` is sitting in `seq_k`'s slot. A reader sees `attention[head, seq_q, ...]` and asks: why is `head` in `seq_k`'s position? The question is visible because the names are visible.

Take a breath. Three chapters of comparisons, and a single thread runs through all of them: the coordinate name is the anchor. In normalization, `mean[feature]` survived layout changes. In attention, `seq_q` vs `seq_k` made cross-attention visible in the type signature. In KV-cache, `concat[seq_k]` named the concatenation axis. In Flash Attention, the same coordinate contract survived a complete kernel rewrite. The anchor does not prevent you from writing bugs. It prevents a specific class of bugs—the ones where the meaning of an axis drifts while its position stays the same. That class is larger than most programmers believe.

Normalization showed the pattern holds. Attention showed what the pattern reveals—distinctions invisible in positional code, visible in names. The final question: *what does the pattern prevent?* Physical simulation has been silently swapping coordinates behind integer indices since before the term "tensor" entered our vocabulary, and the bugs produce plausible-but-wrong physics that no compiler catches and no test suite detects.

Chapter 13 · Comparison: Physics

> "The first principle is that you must not fool yourself — and you are the easiest person to fool."
>
> — Richard Feynman

Comparisons · Heat equations and field components in two notations

Chapters 11 and 12 showed the pattern in machine learning: normalization proved it holds, attention proved it reveals. This chapter asks the final question: **what does the pattern prevent?**

The domain is physical simulation, which predates machine learning by decades. Fortran physicists have been writing `U(I+1, J)` since before the term "tensor" entered our vocabulary—and if you ask one of them what `state[:,:,2]` means, they will give you the correct answer. Then they will tell you about the bug they fixed in 1997 where `2` was actually `3` and the simulation ran for two weeks before anyone noticed. The integer field index—`state[:,:,2]` for velocity-x—is the original ghost in the name.

The stakes are higher here. In ML, a coordinate swap degrades a metric. In physics, a coordinate swap produces negative absolute temperatures, violated conservation laws, waves that amplify instead of propagating. The results look plausible—the contour plot has the right shape, the time series has the right range. Only a physicist's eye catches them. The question is not *does the code run?* It is *does the code solve the right equations?*

The Heat Equation

The one-dimensional heat equation describes how temperature diffuses through a rod over time:

$$u_t = \alpha \cdot u_{xx}$$

In explicit Euler stepping, each point's new temperature is a weighted average of its neighbors:

$$u[t, i] = u[t-1, i] + \alpha \cdot (u[t-1, i+1] - 2 \cdot u[t-1, i] + u[t-1, i-1])$$

NumPy:

```
def heat_diffusion(initial, alpha, T):
    N = len(initial)
    u = np.zeros((T, N))
    u[0] = initial
    for t in range(1, T):
        u[t, 1:-1] = u[t-1, 1:-1] + alpha * (
            u[t-1, 2:] - 2 * u[t-1, 1:-1] + u[t-1, :-2])
    return u
```

The code works. But the Laplacian—the discrete second derivative—is spread across three slice expressions: `u[t-1, 2:]`, `u[t-1, 1:-1]`, and `u[t-1, :-2]`. The relationship between them ("these three terms form a stencil over the spatial coordinate") is invisible. If you swap `alpha` from 0.1 to 0.5 (violating the CFL condition), the code still runs—it just produces physically impossible results.

Einlang:

```
let u[t in 0..T, i] = initial[i];
let u[t in 1..T, i] = u[t-1, i] + alpha * (
    u[t-1, i+1] - 2.0 * u[t-1, i] + u[t-1, i-1]
);
```

The Laplacian is a single expression: `u[t-1, i+1] - 2*u[t-1, i] + u[t-1, i-1]`. The index arithmetic `i+1` and `i-1` makes the stencil visible. `i` is the spatial coordinate, and the offsets `+1` and `-1` are relative to it. The declaration bracket says `t in 1..T, i`—time runs from 1 to T-1, space runs over the whole domain. The recurrence is a fact about the coordinate `t`, stated in the bracket.

Multi-Field Coupling

Real simulations track multiple physical fields—temperature, pressure, velocity components—coupled through partial differential equations. In a positional array, these fields are stored along an integer axis:

```
# state shape: (T, N, 4)
# state[..., 0] = temperature
# state[..., 1] = pressure
# state[..., 2] = velocity_x
# state[..., 3] = velocity_y

def coupled_step(state, t, alpha, beta):
    temp = state[t, :, 0]
    press = state[t, :, 1]
    vx = state[t, :, 2]
    vy = state[t, :, 3]
    # ... coupled equations ...
```

`state[t, :, 0]` extracts temperature. `state[t, :, 1]` extracts pressure. The mapping from integer to physical quantity is in the comments. If a new field is added—say, humidity—it becomes `state[..., 4]`. If the order changes—temperature moves from index 0 to index 2—every `[:, 0]` silently becomes wrong. The code runs. The numbers change. No error is raised.

Einlang:

```
let state[t in 0..T, i, field] = init_field(field, i);

let temp[t, i] = state[t, i, field=0];
let press[t, i] = state[t, i, field=1];
let vx[t, i] = state[t, i, field=2];
let vy[t, i] = state[t, i, field=3];
```

`field` is a coordinate. Its values are named: `field=0` is temperature, `field=1` is pressure. If humidity is added, it becomes `field=4`—a new coordinate value, not a new integer to remember. If the field order changes, the name `field=0` still means temperature, regardless of where it sits in the array.

But the Einlang version does more: it names the *physical coordinate* `i` and the *field coordinate* `field` separately. The coupling equations can reference them by name. A term that depends on temperature reads `state[t, i, field=0]`. A term that depends on the spatial gradient reads `state[t, i+1, field=0] - state[t, i-1, field=0]`. The code says which field and which spatial offset. This is the megaphone

model at the level of physical quantities: `state` speaks on `t`, `i`, and `field`; operations that only care about `i` omit `t` and `field` from their brackets, and the omission is the claim that the stencil is spatial, not temporal or field-specific.

Adding a New Field

Suppose the simulation is extended to include humidity. In the positional version:

```
# Before: state shape (T, N, 4)
# After:  state shape (T, N, 5)
# Every [..., 0:4] slice must be audited.
# Every equation that referenced field indices must be checked.
state = np.zeros((T, N, 5))
temp = state[:, :, 0]       # unchanged - luckily
press = state[:, :, 1]      # unchanged - luckily
vx = state[:, :, 2]         # unchanged - luckily
vy = state[:, :, 3]         # unchanged - luckily
humidity = state[:, :, 4]   # new
```

Every integer index must be verified. The compiler provides no help. If humidity was inserted at index 0 instead of appended at index 4, every subsequent index shifts by one.

In the Einlang version:

```
let state[t in 0..T, i, field] = init_field(field, i);

let temp[t, i] = state[t, i, field=0];
let press[t, i] = state[t, i, field=1];
let vx[t, i] = state[t, i, field=2];
let vy[t, i] = state[t, i, field=3];
let humidity[t, i] = state[t, i, field=4];  // new line
```

The existing field assignments are unchanged. `field=0` is still temperature, regardless of whether humidity is `field=4` or `field=0` with everything else shifted. The coordinate names are stable under insertions because they are names, not positions.

The Coupled Burgers Equation

The 1D coupled Burgers equation for velocity `v` and temperature `T`:

$$v_t + v \cdot v_x = \nu \cdot v_{xx} + \beta \cdot T_x$$

Each term has a specific coordinate interpretation: `v_t` is the time derivative (difference along `t`), `v_x` is the spatial derivative (difference along `i`), `v_{xx}` is the second spatial derivative, and `T_x` is the temperature gradient driving the velocity.

NumPy:

```
for t in range(1, T):
    v_xx = (v[t-1, 2:] - 2*v[t-1, 1:-1] + v[t-1, :-2]) / dx**2
```

```
    v_x = (v[t-1, 2:] - v[t-1, :-2]) / (2*dx)
    T_x = (T[t-1, 2:] - T[t-1, :-2]) / (2*dx)
    v[t, 1:-1] = (v[t-1, 1:-1]
                  + dt * (nu * v_xx
                          - v[t-1, 1:-1] * v_x
                          + beta * T_x))
```

The field identity (`v` vs `T`) is in variable names. The coordinate identity (`t` vs `i`) is in bracket positions. The stencil structure is in the slicing patterns.

Einlang:

```
let v[t in 1..T, i] = v[t-1, i]
    + dt * (nu * (v[t-1, i+1] - 2.0*v[t-1, i] + v[t-1, i-1]) / (dx**2)
            - v[t-1, i] * (v[t-1, i+1] - v[t-1, i-1]) / (2.0*dx)
            + beta * (T[t-1, i+1] - T[t-1, i-1]) / (2.0*dx));
```

The terms are identifiable by their coordinate arithmetic: `i+1` and `i-1` are spatial derivatives. `t-1` is the time recurrence. `v[...]` and `T[...]` are different fields, named as different tensors. The equation reads like the PDE it discretizes.

The Wave Equation: A Stencil in Two Notations

The 1D wave equation describes how a displacement propagates through a medium:

$$u_{tt} = c^2 \cdot u_{xx}$$

In explicit finite differences, it becomes a three-point stencil in space and a two-point stencil in time:

$$u[t,i] = 2 \cdot u[t-1,i] - u[t-2,i] + c^2 \cdot (u[t-1,i+1] - 2 \cdot u[t-1,i] + u[t-1,i-1])$$

NumPy:

```
def wave_step(u, t, c):
    u[t, 1:-1] = (2 * u[t-1, 1:-1] - u[t-2, 1:-1]
                  + c**2 * (u[t-1, 2:] - 2 * u[t-1, 1:-1] + u[t-1, :-2]))
```

The time index `t` is in variable position `u[t, ...]`. The spatial stencil `i-1, i, i+1` is distributed across three slices: `u[t-1, 2:]`, `u[t-1, 1:-1]`, `u[t-1, :-2]`. The second derivative structure `(f[i+1] - 2*f[i] + f[i-1])` is visible only if you mentally align the three slices.

Einlang:

```
let u[t in 0..1, i] = initial[i] + dt * v_initial[i];
let u[t in 2..T, i] =
    2.0 * u[t-1, i] - u[t-2, i]
    + c**2 * (u[t-1, i+1] - 2.0 * u[t-1, i] + u[t-1, i-1]);
```

The Laplacian stencil is a single expression: `u[t-1, i+1] - 2*u[t-1, i] + u[t-1, i-1]`. The index arithmetic names the stencil offsets: `i+1` is the right neighbor, `i-1` is the left neighbor. The time recurrence names `t-1` (one step back) and `t-2` (two steps back). If someone accidentally writes `i+2`

instead of `i+1`, the stencil would be wrong—but the error is a single character in a named expression, not a misaligned slice that the reader must reconstruct.

The Navier-Stokes Skeleton

Fluid dynamics is the grand challenge of computational physics. The Navier-Stokes equations couple velocity, pressure, and vorticity across three spatial dimensions and time. The codebase is typically hundreds of thousands of lines of Fortran or C++, with integer dimension indices scattered throughout. The most common bugs are coordinate swaps—confusing `x` for `y` velocity, or the `x` momentum equation for the `y` momentum equation.

Here is a simplified 2D Navier-Stokes time step in Einlang, using the same coordinate conventions from the heat equation and Burgers equation:

```
let u[t in 1..T, i, j] = u[t-1, i, j]
    + dt * (nu * (u[t-1, i+1, j] - 2.0*u[t-1, i, j] + u[t-1, i-1, j]) / dx**2
           + nu * (u[t-1, i, j+1] - 2.0*u[t-1, i, j] + u[t-1, i, j-1]) / dy**2
           - u[t-1, i, j] * (u[t-1, i+1, j] - u[t-1, i-1, j]) / (2.0*dx)
           - v[t-1, i, j] * (u[t-1, i, j+1] - u[t-1, i, j-1]) / (2.0*dy)
           - (p[t-1, i+1, j] - p[t-1, i-1, j]) / (2.0*dx));
```

The terms are recognizable: the first two lines are the viscous diffusion (Laplacian in `i` and `j`), the third line is the advection (velocity convecting itself), the fourth line is the pressure gradient. Each term names its coordinates and offsets. `i+1` and `i-1` are always the x-differences. `j+1` and `j-1` are always the y-differences. The fields `u`, `v`, `p` are separate tensors with separate names.

In the positional Fortran/C++ version, the same code uses array indices like `U(I+1, J)`, `U(I, J+1)`, `P(I+1, J)`—the coordinate names `i` and `j` are loop variables, not part of the tensor structure. The field identity is in the variable name (`U`, `V`, `P`). The stencil is distributed across multiple array access expressions. If an index is typed wrong—`U(I, J+1)` where `U(I+1, J)` was intended—the compiler cannot catch it because both are valid array accesses. The bug survives compilation and produces physically plausible but incorrect results.

The Einlang version separates three concerns that the Fortran version merges: 1. **Field identity**: `u`, `v`, `p` are different tensors, not different array names pointing into the same multi-field state tensor. 2. **Coordinate identity**: `i` is the x-coordinate, `j` is the y-coordinate. The offsets `+1` and `-1` say which direction. 3. **Stencil structure**: the finite difference terms are grouped by physical meaning (diffusion, advection, pressure).

In Fortran, all three concerns are compressed into `U(I+1, J)`. The compression works. But it makes every stencil access look like every other stencil access. When they differ, only the reader's eye catches the difference.

The Inventory

Three chapters, three domains, one finding.

Normalization showed the pattern holds. Four variants, one skeleton. The coordinate name absorbs layout changes that silently corrupt a positional `dim=`. LayerNorm with `dim=-1` broke when `feature` moved. `mean[feature]` didn't.

Attention showed what the pattern reveals. Self-attention and cross-attention—different semantics, different gradient flows, different architectural implications—are the same Python function. The distinction lives in runtime shapes, not in source code. Named coordinates made the invisible visible: `seq` shared vs `seq_q`/`seq_k` separate.

Physics showed what the pattern prevents. The bugs here are older than machine learning—integer field indices silently swapping since Fortran, stencil slices misaligned since the first finite difference code. The symptoms look plausible. Negative temperatures that still form contour plots. Waves that amplify but the time series has the right range. Only a physicist's eye catches them.

In every domain, the root cause is the same: **the mapping from integer to meaning lives outside the notation.** `dim=-1` is `feature` because the layout convention says so. `state[..., 2]` is velocity-x because the comment says so. `u[t-1, 2:]` is the right neighbor because the reshape put it there. When the layout changes, the meaning drifts. The integer does not change. Only the meaning does.

In Einlang, the coordinate name is the anchor. `mean[feature]` stays `mean[feature]` regardless of layout. `field=2` stays velocity-x regardless of field order. `i+1` stays the right neighbor regardless of which position `i` maps to. The name is tied to the coordinate, not to its position. The integer is the implementation detail.

Here is a PDE stencil. Before reading the commentary, find the coordinate that carries recurrence:

```
let u[t, i, j] = u[t-1, i, j]
    + c * (u[t-1, i+1, j] - 2.0 * u[t-1, i, j] + u[t-1, i-1, j])
    + c * (u[t-1, i, j+1] - 2.0 * u[t-1, i, j] + u[t-1, i, j-1]);
```

`t` only appears as `t-1` on the right — never as `t+1`. The recurrence arrow constrains `t`. The compiler enforces that `t+1` cannot appear on the right-hand side of a definition for `t`. This expression passes.

Now imagine a colleague accidentally swaps `i` and `j` in the second line — writes `u[t-1, j+1, i]` instead of `u[t-1, i, j+1]`. In a positional `u[t-1, :, :]`, the swap is invisible — the code runs, produces numbers, the contour plot looks right. But the x-derivative and y-derivative have been exchanged. The physics is wrong. In the named version, `i` and `j` are different names. `u[t-1, j+1, i]` puts a `j+1` expression where `i` is declared — the coordinate mismatch is in the source. The stencil doesn't silently swap. The names won't let it.

Two Notations, One Task

The three comparison chapters end here. A fair assessment requires stating what positional notation does well.

Positional notation is concise, universal, and runs directly on every accelerator. When coordinates are genuinely anonymous—a ReLU activation, an element-wise addition—the two notations cost the same keystrokes and the same thought. Named notation earns its keep where identities diverge: `class` vs `batch`, `seq_q` vs `seq_k`, `velocity-x` vs `pressure`.

There is a subtler argument for positional notation: **sometimes `dim=-1` is correct by construction.** A softmax that normalizes over the last dimension will be correct for any tensor whose last dimension happens to be the correct one—and in many codebases, that invariant is genuinely stable. Positional notation's "ambiguity" can be a form of flexibility: the same function works on different layouts because it only cares about relative position, not absolute identity.

The coordinate habit does not deny this. It asks a narrower question: *when the operation depends on which coordinate is which, is that dependency recorded?* If your codebase enforces the convention that the last dimension is always `feature`, `dim=-1` is a shorthand for a well-understood invariant, not a bug waiting to happen. The problem is `dim=-1` in a codebase where the invariant is undocumented, unenforced, and assumed.

Names are not a replacement for conventions. They are a way to make conventions checkable. The coordinate habit says: if a convention exists, record it. If it doesn't, the name is where you discover that.

If the magnetic field index moves from 4 to 0, how many lines of code do you need to change?

The comparison chapters are not an argument that positional notation is bad. They are a demonstration that positional notation is incomplete. The integer records a position. The name records an identity. Both are facts. Only one is in the source code.

Three Chapters, One Verdict

Normalization: One Name

In normalization, the coordinate name captures *which semantic group is being reduced over.* The positional code for LayerNorm, RMSNorm, InstanceNorm, and GroupNorm is identical except for the `dim` argument — a tuple of integers whose meaning depends on the tensor layout and reshape chain. When the layout changes, the tuple must be updated. When it is not, the normalization silently operates over the wrong axes.

The Einlang versions differ only in the reduction bracket:

```
mean[feature]           // LayerNorm
mean[feature]           // RMSNorm - same reduction, different body
mean[..s]               // InstanceNorm
mean[c_in_group, ..s]   // GroupNorm
```

`feature` is `feature` whether it sits at position -1 after a reshape or position 2. The name finds it. The integer counts to it. The difference between the four variants is one name in the bracket. The skeleton — reduce, subtract, divide, scale, shift — is identical.

Attention: Two Names

In attention, the coordinate names capture *whether this is self-attention or cross-attention.* The positional code for both is literally identical — the same Python function, the same `matmul`, the same `softmax(dim=-1)`. When the shapes happen to match during development, the two attentions are indistinguishable.

The Einlang signatures make the distinction visible:

```
fn attention[seq_q, seq_k, head, d](Q, K, V)   // cross: seq_q  seq_k
fn attention[seq, head, d](Q, K, V)             // self: same coordinate
```

`seq_q` and `seq_k` are different names. A reader can see which attentions are self and which are cross without checking whether the tensors happen to have the same length. Every architectural variant — MHA, GQA, MQA, cross-attention — is a different assignment of names to parameters.

Physics: Three Names

In physics, the coordinate names capture *whether i is the x-grid or the y-grid.* A single typo — `u[j, i]` instead of `u[i, j]` — compiles, runs, and produces plausible-but-wrong numbers. In the Navier-Stokes skeleton, the Laplacian has two terms — `u[t-1, i+1, j] - 2*u[t-1, i, j] + u[t-1, i-1, j]` for x-diffusion and `u[t-1, i, j+1] - 2*u[t-1, i, j] + u[t-1, i, j-1]` for y-diffusion. The terms are identical except for which name is offset. If `i` and `j` are swapped, the x-derivative and y-derivative are exchanged. In positional Fortran — `U(I+1, J)` vs `U(I, J+1)` — the swap is a one-character typo that produces valid array accesses.

The difference between correct physics and destroyed physics is three names: `t` for time, `i` for x-grid, `j` for y-grid. The names carry the semantic roles that the integers cannot.

The One-Bit Threshold

Now step back. What do one name, two names, and three names have in common?

A positional integer carries exactly one piece of information: *which dimension number this is.* As long as every coordinate is interchangeable with every other — as long as axis 0 is axis 0 regardless of what it represents — that one piece of information is sufficient. A ReLU applied to `dim=-1` is correct whether `dim=-1` means `feature`, `channel`, or `time`.

The threshold is crossed when the semantic role of a coordinate exceeds what a position can say. When `dim=-1` means `feature` in LayerNorm but `channel-group-index` in GroupNorm after a reshape. When the integer `2` means velocity-x in one file and pressure in another. When `seq_len == seq_len` makes self-attention and cross-attention indistinguishable. At this threshold, the integer still compiles. It still runs. It just no longer means what the programmer thinks it means.

The threshold is not a matter of discipline. It is an information-theoretic limit: one integer can carry one piece of information. When you need to know not just *which axis* but *what that axis represents*, the integer has been asked to carry a payload it was never designed to hold.

Domain	What the integer hides	What the name reveals	Failure mode
Normalization	Which axes are reduced	The semantic group (spatial, channel, batch)	Silent wrong normalization
Attention	Whether two sequences are same or different	Self vs cross, query vs key	Indistinguishable forward passes
Physics	Which physical dimension	x vs y, temperature vs pressure	Compiles, produces wrong physics

In every domain, the integer records a position. In every domain, the semantic role of the coordinate exceeds what a position can say. In every domain, the failure is silent. In every domain, the name catches what the integer cannot express.

The margin of safety is the name. In normalization, one name. In attention, two names. In physics, three names. The name in the bracket is not decoration. It is the one bit of information that an integer cannot carry — and when that information matters, the integer costs more than the name ever will.

Now the question the book has been circling since Chapter 1: if names are so useful, what can they NOT do? Every tool has a boundary. The boundary is not a flaw. It is a map. And a good map tells you where the boundaries are.

Chapter 14 · The Edge of the Name

> "The map is not the territory—but a good map tells you where the boundaries are."
>
> — Adapted from Alfred Korzybski

Boundaries · What names can check, and what they cannot

Suppose the programmer writes:

```
softmax[batch](logits[batch, class])
```

It compiles. Every name matches. `batch` exists on `logits`. The reduction consumes a declared coordinate. The contract is satisfied. The shapes are correct.

But the programmer meant `softmax[class]`. Normalizing over batch instead of class is a bug—a model that silently computes the wrong probabilities, descends a valid loss, and deploys with a semantic error no tool can detect.

The compiler cannot catch this. And it was never intended to.

Consistency and Correctness

A type checker verifies that `int` and `string` are used consistently. It catches `"hello" + 3`. It does not catch `interest = principal * (1 - rate)` when the formula should be `interest = principal * (1 + rate)`. One is a type error—the code is internally incoherent. The other is a formula error—the code is internally coherent but matches the wrong formula.

Coordinate names extend this boundary. A reduction over a coordinate the tensor doesn't have is a type error—caught at compile time. A reduction over `batch` where `class` was intended is a formula error—internally consistent, semantically wrong.

Consistency is internal. Does the coordinate story cohere? Does `channel` appear wherever the reduction claims it does? Does the function signature match the call site? The compiler can check consistency, because consistency is a relationship between declarations—and declarations are all in the source.

Correctness is external. Does the computation achieve what the programmer intended? Does `mean[channel](x)` express the right thing? The compiler cannot check correctness, because correctness is a relationship between the source and the programmer's intent—and intent is not in the source.

Return to the opening example. `batch` exists on `logits`. The reduction consumes it. Every check passes. The program compiles. It produces a valid probability distribution—over the batch dimension, not the class dimension. The name was wrong. The check passed. The program is incorrect.

But the name `batch` is visible. When the next programmer reads `softmax[batch](logits)`, they see the error immediately. The positional equivalent `softmax(logits, dim=0)` hides the error behind a number. The reader sees `dim=0` and must reconstruct whether axis 0 is batch or class. The reconstruction may be wrong.

A wrong name is a visible error. A missing name is an invisible one.

What Names Check

Before what names cannot check, what they can:

- Every coordinate in an index list must exist on the tensor.
- The consumed coordinate must appear in every operand.
- Every omission is recorded for the backward pass.
- Recurrence references must be strictly backward in time.
- Function call coordinate arguments must match the declaration.

Every one of these is a consistency check. Every one operates on names declared in the source. None requires data. None requires execution. All five together define the boundary: *if a bug can be expressed as a mismatch between two declarations in the source, the compiler catches it. If the bug lives entirely in the gap between the source and the programmer's intent, the compiler cannot.*

What Names Cannot Check

Three categories. Each one is a different kind of silence between what the code says and what the programmer meant.

1. The Wrong Name

A programmer writes:

```
let result[..b, feature] = softmax[batch](logits[..b, feature]);
```

`batch` exists on `logits`. The reduction consumes it. The gradient sums over it. Every check passes. The program compiles. It produces a valid probability distribution — over the batch dimension, not the feature dimension. The name was wrong. The check passed. The program is incorrect. But the wrong name sits in the bracket, visible to the next programmer who reads `softmax[batch](logits)` and immediately asks: "should this be `feature`?"

Now the positional version:

```
result = torch.softmax(logits, dim=0)
```

`dim=0` is valid. The shapes match. The output is a valid probability distribution — over axis 0, which might be batch, might be feature, might be something else entirely. The reader sees `dim=0` and must reconstruct which coordinate axis 0 maps to. That reconstruction may be wrong. And even if it's right, the code records nothing to confirm it.

A wrong name is a visible mistake. A missing name is an invisible one. The compiler cannot prevent you from choosing the wrong coordinate — that would require reading your mind. But it can make the choice visible. And visibility is the difference between a bug found in review and a bug found in production.

2. What Arithmetic Cannot Promise

You write a convolution:

```
let conv[b, oc, oh, ow] = sum[ic, kh, kw](
    input[b, ic, oh + kh, ow + kw] * weight[oc, ic, kh, kw]
);
```

The compiler checks that `oh`, `kh`, `ow`, `kw` are declared coordinates. It knows their domains. If `oh` ranges over `0..32`, `kh` over `0..3`, and `input` has width 35, the compiler proves: worst case `oh + kh = 31 + 3 = 34 < 35`. Safe. The check is compile-time.

Now the same expression with runtime dimensions. The image is read from a file. The input width is a variable. The compiler cannot prove `oh + kh < input_width` because `input_width` is not a constant. It emits a runtime guard: `assert oh + kh < input_width`. The name is in the guard: `IndexError: oh + kh = 67 exceeds input width 64`. The coordinate names the expression that overflowed. The positional equivalent: `IndexError: dimension 3 out of bounds`. Which dimension is 3? What was supposed to be in bounds? The number answers neither question.

The boundary here is not about names. It is about static vs. dynamic knowledge. The compiler proves everything provable from declarations. What it cannot prove, it guards. The name is in the error either way.

3. The Shape That Isn't There Yet

A beam search decoder. The output sequence length depends on the input — sentences are different lengths. The compiler knows `seq` is a coordinate. It does not know the length of `seq` for any particular input. That length is the answer.

This is not a failure of the compiler. It is the nature of the computation. The output shape is data-dependent. No static system can determine it. The name is still there — `seq` names the coordinate whose length varies. The runtime code that handles the dynamic length reads `seq_len = lengths[b]`. The name connects the declaration to the dynamic bound. The positional equivalent: `axis 1 has dynamic extent`. Axis 1 might be `seq`, `class`, `head`, or `feature`. The number doesn't know. The name does.

Three Kinds of Silence

Step back from the three categories and they resolve into a single taxonomy of what the compiler cannot say.

The silence of intent. You chose `batch` when you meant `feature`. The code is consistent. It is wrong. No static check can close the gap between what you wrote and what you meant. What the name provides: the gap is visible. `softmax[batch](logits)` wears its intent in a bracket. `softmax(logits, dim=0)` wears it in a convention. Conventions can be forgotten. Brackets cannot.

The silence of data. The input width is a variable. The sequence length varies per sample. The output shape is the answer. These facts are not known at compile time because they are not knowable at compile time. The compiler checks everything that can be checked from declarations alone. What remains is guarded at runtime — with names attached. The name doesn't prevent the runtime check. It makes the runtime check readable.

The silence of the world. You are simulating the Navier-Stokes equations. The code compiles. The shapes match. The simulation runs. It produces negative absolute temperatures. The coordinate names are correct — `u`, `v`, `p` are different tensors, `i` is x, `j` is y. Every check passes. And the physics is wrong because the discretization is unstable at the chosen Reynolds number. No programming language catches this. The boundary is not between named and positional notation. It is between computation and reality. Names record identities. They do not replace physics.

The first silence is the one the book has spent fourteen chapters arguing against — the silence where the notation refuses to record a fact the programmer knows. The second silence is honest: the compiler

checks what it can, guards what it can't, and names both. The third silence is the edge of the map. Here there be dragons — and no notation, named or positional, will chart them for you. But the map that marks the boundary honestly is better than the map that pretends the boundary does not exist.

The Middle Ground

Between a purely positional notation and a complete named-coordinate compiler lie intermediate solutions. Each records more than integers. None records everything.

Defensive assertions. `assert x.shape[1] == channel_size` catches a refactoring when the sizes differ. But if `channel` and `spatial` happen to have the same extent, the assertion passes. Assertions check shapes, not identities. They protect against size mismatches, not semantic drift.

Einops. `reduce(x, "batch channel spatial -> batch spatial", "mean")` names the reduced coordinate at the call site. The string notation is minimal, library-level, and works in PyTorch today. For many projects, einops alone eliminates the most common positional bugs. But the names in the string are not checked against any declaration. If the tensor actually contains `time` rather than `channel`, the string won't catch it. And einops names die at the edge of the expression — the next function receives an anonymous tensor and must re-discover what the dimensions are called.

PyTorch Named Tensors. `x.refine_names("batch", "channel", "spatial").mean("channel")` checks that `channel` exists. But many operations strip names silently — `torch.matmul`, `torch.cat`, most `nn` layers. When a name is stripped, the protection vanishes without warning.

Einlang. The coordinate contract is part of the function type. Every call site is checked. Every operation preserves names or explicitly consumes them. The contract is global. The check is complete. The cost is a compiler.

The distance between *no checking* and *complete checking* is measurable. Einops catches local bugs. Named tensors catch the ones that survive through supported operations. Einlang catches all of them. Which step you take depends on how much correctness you need and how much infrastructure you can afford. The coordinate habit works at every step. It only asks: *is the name in the code?* In an einops string, that's a name. In a bracket a compiler checks, that's a name with a guarantee. The habit does not prescribe the tool. It prescribes the information.

Five bugs. Which ones does the compiler catch?

`softmax[class](logits[batch, channel])` — `class` is not on `logits`. Caught.

`softmax[batch](logits[batch, class])` — `batch` exists, every check passes. Not caught. The wrong name is visible in the bracket, but the compiler cannot read intent.

`sum[head](x[batch, head, d])` where the programmer meant `sum[d]`. `head` exists. Not caught. Same structure: the name is wrong, the check passes.

A convolution where `oh + kh` exceeds the input width, width is a runtime variable. The compiler can't prove the bound statically. It emits a runtime guard with the coordinate name in the error. Not caught at compile time — guarded at runtime, with the name attached.

`mean[spatial](x[batch, channel, spatial])` where `x` actually has `{batch, channel, time}` — `spatial` was renamed in a refactoring. Caught. Missing coordinate.

The compiler catches two, guards one, and is silent on two. The two it misses are the ones where the name is consistent but wrong. No static system catches those. But in the named versions the wrong name is in the bracket. In the positional versions it is not in the code at all.

What Survives

Names do not eliminate runtime shape errors. They do not replace testing. They do not guarantee correctness.

They do one thing: they prevent the class of errors where the coordinate identity exists in the programmer's head but not in the source text, because the notation provides no place to record it. The silent axis swap. The broadcast that drifts with the layout. The reduction that changes meaning without changing syntax. For that class, names are the only defense.

The boundary is not a flaw. It is a map of what is statically knowable. Every fact names cannot check is a fact no purely static system can check — the programmer's intent, the runtime data, the physics of the world. The names don't fail at these boundaries. They mark them. And a marked boundary is one you don't cross by accident.

This is the argument the book has been making since Chapter 1. Not that names make programming safe. That names make identity visible. And visibility is the difference between a bug found in review and a bug found at 3 AM, three weeks after it shipped, because the notation had no slot for the fact that would have caught it.

Before you close the book, open your most recent tensor code. Find a `dim=` argument. Ask: which coordinate is that? If you can't answer from the code alone, the coordinate isn't in the code. It's in your head. The name is missing. And the name is the only thing that can go in the bracket.

Appendix · The Complete Picture

> "The purpose of computing is insight, not numbers."
>
> — Richard Hamming

Reference · Thought map and syntax at a glance

The preceding fourteen chapters introduced Einlang's grammar piece by piece, each piece arriving when the concept it served had earned its introduction. What follows assembles them into one place, organized by category rather than by pedagogical necessity.

Think of it as the view from the summit. You climbed the mountain one trail at a time. Now you can see the whole range.

But first, draw your own map.

Build Your Own Map

Take out a piece of paper. Or open a blank document. At the top, write: `dim=1 bug`.

Now draw arrows downward. Each arrow is a question the bug forced us to ask. *"Which coordinate did I just erase?"* → naming. *"Which coordinate is being reduced?"* → reduction bracket. *"Why can't the compiler check this?"* → coordinate contracts.

Don't look at the next section yet. Draw from memory. What were the big ideas? How do they connect? Which chapters depend on which?

Five minutes. Go.

Done? Good. Now look at your map and ask three questions:

1. **Which arrow did you forget?** Everyone forgets at least one. The arrow you forgot connects two ideas you hadn't realized were dependent on each other. That connection is the thing you haven't fully internalized yet.
2. **Which arrow did you draw but can't explain why it exists?** You remembered that A depends on B, but you can't articulate the dependency. That arrow is a memory, not an understanding. Go back to the chapter where that arrow was first drawn and reread the transition.
3. **Which idea has the most arrows pointing to it?** That idea is the load-bearing concept. In this book, it is almost certainly "a coordinate has a name." Everything else depends on it. If you had to explain the book in one sentence, that concept would be in it.

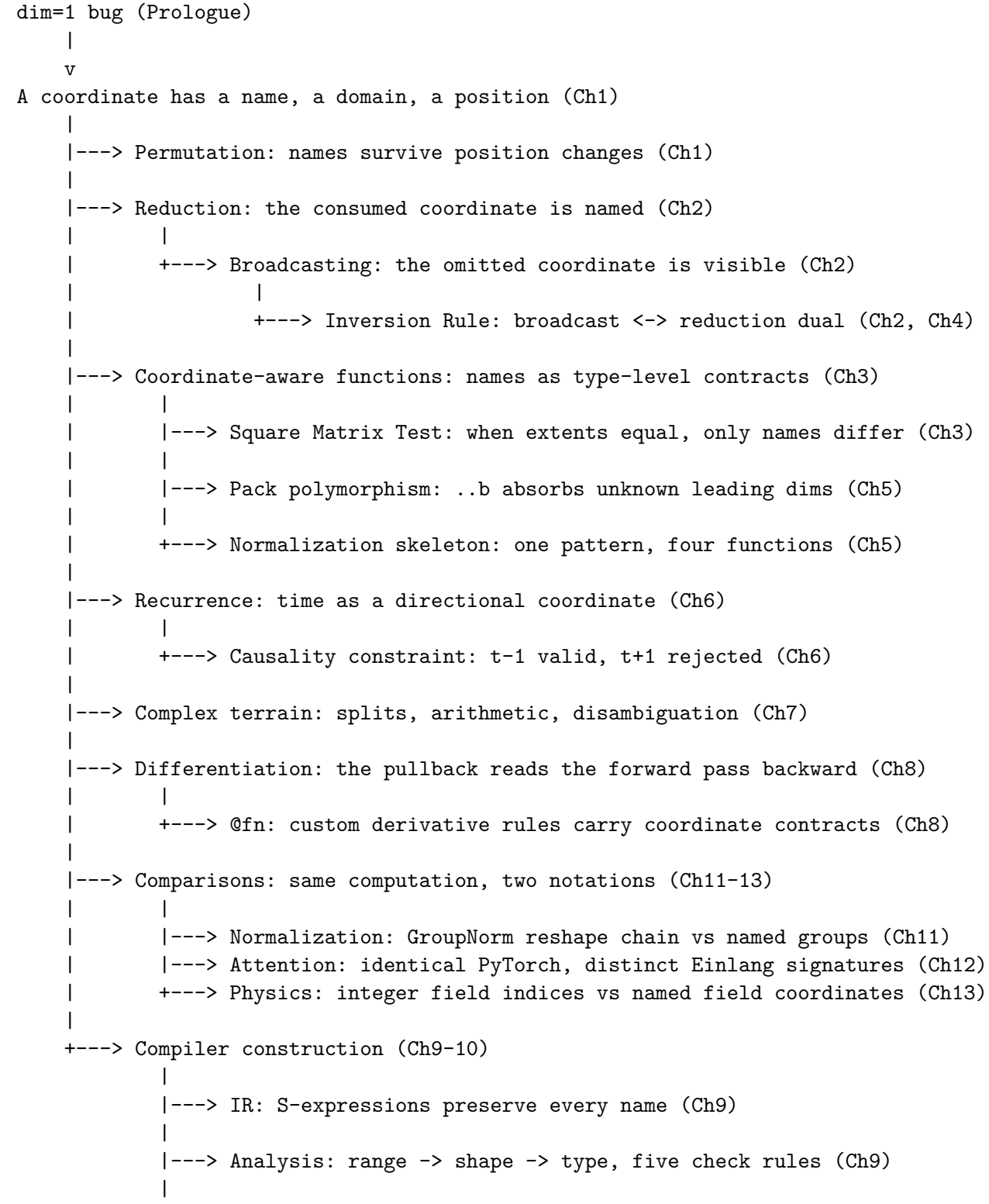
The map you drew is not the final answer. It is a snapshot of your understanding at this moment. A month from now, draw it again. The arrows will have moved. Some will have disappeared—their dependencies now obvious. Others will have appeared—connections you didn't see the first time.

Learning is not the accumulation of facts. It is the continuous redrawing of the map.

The Thought Map (One Version)

Here is one version of the map. It is not the only version. Compare it to yours. Where do they agree? Where do they differ? The differences are not errors—they are perspectives.

Before the syntax reference, a map of how the ideas connect. Each arrow is a dependency: the idea at the tail must be understood before the idea at the head.



```
+---> Lowering: names -> integers, strategies (Ch9-10)
        |
        +---> Firewood: names burn, heat remains (Ch9)
```

Every path begins at the `dim=1` bug. Every arrow is a question the bug forced us to ask. The map is not the territory—but it shows how the trails connect.

Declarations

`let` binds an immutable name to a value. *Introduced in Chapter 1.*

```
let x = 42;
let pi: f64 = 3.141592653589793;
let matrix: [f32; 2, 3] = [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]];
```

Type annotations are optional. When present, the value must be compatible. All `let` bindings are immutable.

Rectangular Declarations

A rectangular declaration binds a tensor by naming its coordinates. *Introduced in Chapter 1; extended with domains in Chapter 5.*

```
let C[i, j] = sum[k](A[i, k] * B[k, j]);
```

Index slots in the declaration bracket may be: - A name: `i`, `j`, `batch` — the standard case. - A name with an explicit domain: `t in 0..T` — for recurrences. - A literal: `0` — used for base cases. - A named rest: `..b` — absorbs zero or more adjacent axes.

Expressions are not allowed in the declaration bracket. `let fib[n-1] = ...` is an error. The left side names what is being defined. The right side computes it.

Reductions

A reduction consumes a coordinate. *Introduced in Chapter 2; selection reductions in Chapter 5.*

Operations: `sum`, `max`, `min`, `prod`.

```
let total = sum[i](data[i]);
let row_sums[i] = sum[j](matrix[i, j]);
```

Selection reductions return addresses rather than values:

```
let pred[b] = argmax[class](logits[b, class]);
```

The consumed coordinate is eliminated from the result shape. The reduction bracket names it explicitly—the reader does not need to infer which coordinate disappeared.

Broadcasting

Broadcasting is an omission in the indexing pattern. *Introduced in Chapter 2; self-audit in Chapter 4.*

```
let out[i, j] = A[i, j] + bias[j];   // bias omits i → broadcast over i
```

The omitted coordinate is the one being broadcast over. The megaphone model: `bias` is silent on `i`, so the compiler copies it across all values of `i`. The silence is a semantic claim: `bias` does not depend on `i`.

The Inversion Rule: what broadcasts in the forward pass is reduced in the backward pass. `bias[j]` omits `i` forward → `d_bias[j] = sum[i](d_out[i, j])` backward.

Named Rest Indices

`..name` stands for zero or more adjacent axes, collectively named. *Introduced in Chapter 2; pack polymorphism in Chapter 5.*

```
let result[..b, j] = x[..b, j] + bias[j];
let row_sum[..b] = sum[j](x[..b, j]);
```

The same rest name must describe the same axis span within an expression. Packs make functions rank-polymorphic: the same `layer_norm[feature]` works on 2D, 3D, or 4D inputs.

Where Clauses

A where clause filters or binds. *Introduced in Chapter 2; backward behavior in Chapter 8.*

Boolean guards narrow the domain:

```
let pos_sum = sum[i](data[i]) where data[i] > 0;
let upper[i, j] = matrix[i, j] where i <= j;
```

Variable bindings name intermediate values:

```
let output[i, j] = activated
    where z = sum[k](input[i, k] * weight[k, j]) + bias[j],
          activated = if z > 0.0 { z } else { 0.0 };
```

In the backward pass, filtered elements receive zero gradient. The domain constraint applies symmetrically in both directions.

Coordinate-Aware Functions

A function may declare coordinate parameters. *Introduced in Chapter 3; pack parameters in Chapter 5.*

```
fn softmax[j](x: [f32; ..left, j, ..right])
    -> [f32; ..left, j, ..right]
{ ... }
```

Call sites pass coordinate arguments in the bracket position:

```
let p[b, class] = softmax[class](logits[b, class]);
```

The compiler checks that `class` exists on `logits` and that the coordinate contract is satisfied. The bracketed name is part of the call contract, not a comment.

Packs (`..left`, `..right`, `..s`) make functions polymorphic over surrounding structure. A caller disambiguates by grouping: `softmax[(height, width)](x)`.

Recurrence Relations

Self-referential declarations define sequences over time. *Introduced in Chapter 6.*

```
let u[t in 0..T, i] = initial[i];
let u[t in 1..T, i] = u[t-1, i] + f(u[t-1, i]);
```

Backward references only. `u[t+1, i]` on the right-hand side with declaration index `t` is a compile error. Causality is a syntactic constraint, not a convention.

The optimizer is a recurrence:

```
let w[t in 1..T, out, in] = w[t-1, out, in] - lr * grad[t-1, out, in];
```

Automatic Differentiation

`@loss / @W` computes the gradient. *Introduced in Chapter 8.*

```
let dW = @loss / @W;
```

The gradient has the same shape as the denominator. The pullback is computed by reversing the forward graph: every forward reduction becomes a backward broadcast; every forward broadcast becomes a backward reduction. The shopping cart record, read in reverse.

Custom rules use `@fn`:

```
@fn relu(x) {
    if x > 0.0 { @x } else { 0.0 }
}
```

Coordinate-aware custom rules carry the same bracketed parameters as the primal function.

Why the Compiler Reads Coordinates Too

The preceding sections catalogued syntax. But syntax is only half the story. Each compiler pass depends on coordinate names to do its job. *These passes are described in Chapters 9–10.*

Shape inference (Ch9–10) reads coordinate names to decide whether an expression is legal before it runs. `sum[k](A[i, k] * B[k, j])` succeeds if `k` appears in both `A` and `B`. Under names, the contract is: `i` survives from `A`, `j` survives from `B`, `k` appears in both and is consumed.

Range analysis (Ch10) finds the domain of every axis: from array shapes, from literals, or from explicit declarations. Every coordinate gets a concrete range before code generation.

Five check rules (Ch9) verify the IR: index existence, reduction consistency, broadcast recording, causality, and coordinate contract at call sites. Each catches a class of bug that positional notation silently accepts.

Gradient lowering (Ch9) reads coordinate names to build the backward pass. The rule: preserve the coordinates of `W`, sum over everything else. Set subtraction, applied to coordinate names, derives the pullback.

Storage planning (Ch9) reads coordinate names to decide which tensors can share memory. A recurrence creates a dependency chain; the compiler allocates a rolling buffer.

Kernel fusion (Ch9) reads coordinate names to decide which operations can be merged. Operations that share surviving coordinates can fuse; operations across a reduction boundary cannot.

Error Codes

Three errors are especially relevant to the coordinate habit. You won't memorize error codes from a book. But reading them now means you'll recognize them when they appear:

- **E0425 (Undefined Coordinate)**: a coordinate name is referenced but does not exist on the tensor.
- **E0308 (Coordinate Range Mismatch)**: two uses of the same coordinate name infer incompatible ranges.
- **E0061 (Coordinate Contract Violation)**: a function call supplies a coordinate argument that does not match the function's declared coordinate parameter layout.

Error Walkthrough

A list of error codes is a reference. A walkthrough is a skill. Here are the three errors as you would encounter them in practice: source, message, reading, and fix.

E0425: The typo that silence would swallow.

You write a softmax with a coordinate that doesn't exist:

```
let probs[batch, class] = softmax[clss](logits[batch, class]);
```

The compiler responds:

```
error[E0425]: coordinate `clss` not found in this scope
  → line 12, column 35
  `clss` is not declared on any tensor in the reduction body.
  Declared coordinates on `logits`: batch, class
  Hint: did you mean `class`?
```

Reading the error: the compiler names the offending coordinate (`clss`), shows you which coordinates actually exist on the tensor (`batch, class`), and suggests the correction. In a positional API, `softmax(logits, dim=1)` would run without error—and silently normalize over whatever axis happens to be at position 1.

Fix: `s/clss/class/`. One keystroke, caught at compile time.

E0308: The shape mismatch that surfaces before runtime.

You multiply two matrices with incompatible contraction dimensions:

```
let C[i, j] = sum[k](A[i, k] * B[k, j]);
```

If `A` has `k = 64` but `B` has `k = 128`:

```
error[E0308]: coordinate range mismatch for `k`
  → line 8, column 25
  `k` inferred as 64 from `A[i, k]` (declared at line 5)
  `k` inferred as 128 from `B[k, j]` (declared at line 6)
  These ranges must be equal for contraction.
```

Reading the error: the compiler tracked the size of `k` through both declarations, compared them at the reduction site, and found a contradiction. The error names both tensors and both ranges. In a positional API, this becomes `RuntimeError: size mismatch, m1: [32 x 64], m2: [128 x 64]`—which tells you the shapes but not which axis is wrong, which declaration caused it, or what the expected size should be.

Fix: align the declarations of `k`. The error tells you exactly where to look.

E0061: The contract that positional APIs leave as a comment.

You call a function with a coordinate parameter that violates its contract:

```
fn layer_norm[feature](x: [f32; batch, feature]) -> [f32; batch, feature] {
    let mean[batch] = mean[feature](x);
    let var[batch] = var[feature](x, mean[batch]);
    (x - mean[batch]) / sqrt(var[batch] + 1e-5)
}

let h[batch, channel] = layer_norm[batch](x[batch, channel]);
```

The compiler responds:

```
error[E0061]: coordinate contract violation in call to `layer_norm`
  → line 20, column 27
  `layer_norm` expects coordinate parameter `feature` (consumed by reduction)
  Called with `batch`, which appears in `..left` position of argument `x`
  Expected layout: feature is consumed; batch survives
  Actual layout:   batch would be consumed; feature is not in reduction
```

Reading the error: `layer_norm`'s contract says "I consume `feature` and preserve `batch`." The call says "consume `batch`." The two don't match. The compiler shows both the expected contract and the actual call site, so you can compare them side by side. In a positional API, `layer_norm(x, dim=-1)` would run—and normalize over `batch` instead of `feature` if the tensor happened to be transposed.

Fix: `layer_norm[feature](x[batch, channel])`. The coordinate parameter matches the contract.

These three errors share a structure. Each one: (1) names the coordinate involved, (2) shows where it was declared and where it was used, (3) states what was expected versus what was found. The structure is not Einlang-specific. It is the structure of any good type error. The coordinate names make it possible.

No names → no E0425. No coordinate-aware functions → no E0061. The error codes are not arbitrary. They are the compiler saying, in structured form: "the name you wrote does not match the names the program declares."

How to Use This Chapter

This reference is built to be revisited—opened to the section you need.

If you're writing a new Einlang function and can't remember the exact syntax for a recurrence declaration, open to "Recurrence Relations." If you're debugging a coordinate mismatch and want to re-derive the pullback rule, open to "Automatic Differentiation." If you're designing a new operation and want to check whether it fits the existing primitives, trace it through the Thought Map.

The syntax reference is the scaffolding. The thought map is the blueprint. Together they let you rebuild what you need without rereading the whole book.

But the most important section here is not the syntax. It is the four-question audit table. Those four questions work in any framework. They are the coordinate habit, reduced to its smallest portable form. Copy them. Tape them to your monitor. Use them on your next tensor bug.

Five Principles

1. Coordinates have identities. `batch`, `channel`, `time`, `feature` are not positions — they are names. A position records where; a name records what.

2. Reductions must name what they consume. `sum[class](x)` names the erased coordinate. `x.sum(dim=1)` erases a position.

3. Broadcasts must name what they copy along. `bias[j]` omits `i` — the omission records the broadcast.

4. Functions must declare their coordinate contracts. The bracketed parameter is part of the type. The compiler checks every call site.

5. Gradients read the forward pass backward. What consumed forward broadcasts backward. What omitted forward sums backward.

These principles are not Einlang-specific. They apply in any notation that records coordinate identities — brackets, einops strings, comments. The habit outlasts any particular syntax.

Five Principles in Practice

Each principle is a claim about what the notation should record. But principles read differently when you see them applied to a single program. Here is one program—a linear layer with LayerNorm—written first without the principles, then with each applied in turn.

Without any principles. PyTorch:

```
def forward(x, W, b, gamma, beta):
    h = x @ W.T + b
    mean = h.mean(dim=-1, keepdim=True)
    var = (h - mean).pow(2).mean(dim=-1, keepdim=True)
    return gamma * (h - mean) / torch.sqrt(var + 1e-5) + beta
```

Seven lines. The operations are correct. The axes are implicit. `dim=-1` appears twice. If `h` changes from `(batch, feature)` to `(batch, seq, feature)`, `dim=-1` silently changes meaning from `feature` to `seq`. The code runs. The normalization is over the wrong axis.

Principle 1 applied: Coordinates have identities.

```
# h: (batch, feature) - feature is dim=-1
def forward(x, W, b, gamma, beta):
    h = x @ W.T + b
    mean = h.mean(dim=-1, keepdim=True)  # dim=-1 = feature
    var = (h - mean).pow(2).mean(dim=-1, keepdim=True)  # dim=-1 = feature
    return gamma * (h - mean) / torch.sqrt(var + 1e-5) + beta
```

The comments record identity. They can rot. But they are present—a reader six months later can see what `dim=-1` was supposed to mean. When `h` gains a `seq` dimension, the comments say `feature` but the code now normalizes over `seq`. The comment is wrong. The reader has a chance to notice the mismatch. Without comments, there is no mismatch to notice—the code changed silently.

Principle 2 applied: Reductions must name what they consume.

```
// h: (batch, feature)
let avg[batch] = mean[feature](h[batch, feature]);
let var[batch] = mean[feature]((h[batch, feature] - avg[batch]) ** 2.0);
let normalized[batch, feature] = gamma[feature] * (h[batch, feature] - avg[batch]) / (var[batch]
```

The reduction names `feature`. The name appears in the bracket, not in a comment. If `h` gains a `seq` dimension, its declaration becomes `h[batch, seq, feature]`. The reduction `mean[feature]` still names `feature`—it does not silently switch to `seq`. The name protects the reduction from the layout change.

Principle 3 applied: Broadcasts must name what they copy along.

```
# mean[batch], var[batch] - silent on feature, broadcast back over it
# gamma[feature], beta[feature] - silent on batch, broadcast along batch
let normalized[batch, feature] = gamma[feature] * (h[batch, feature] - mean[batch]) / sqrt(var[ba
```

Two broadcasts. `gamma` and `beta` silently copy over `batch`. `mean` and `var` silently copy over `feature`. Every omission is visible in the index patterns—the coordinate that is absent from the bracket is the coordinate the tensor broadcasts over. The backward pass will sum over the appropriate coordinate for each parameter.

Principle 4 applied: Functions must declare their coordinate contracts.

```
fn layer_norm[feature](x: [f32; ..b, feature], gamma: [f32; feature], beta: [f32; feature])
    -> [f32; ..b, feature]
```

The coordinate parameter `feature` is part of the function's type. Every call site that passes `feature` is checked: does the tensor have a coordinate called `feature`? The contract is not a docstring. It is verified.

Principle 5 applied: Gradients read the forward pass backward.

```
Forward: mean[feature](h[batch, feature]) → mean[batch] (broadcasts over feature)
         gamma[feature] * ... (broadcasts over batch)
Backward: d_mean[batch] = sum[feature](d_norm[batch, feature] * ...)
          d_gamma[feature] = sum[batch](d_norm[batch, feature] * ...)
```

The backward sums are over the coordinates that were broadcast forward. `mean` consumed `feature` → backward sum consumes `feature`. `gamma` omitted `batch` → backward sum consumes `batch`. The

Inversion Rule, applied mechanically from the forward coordinate sets.

The Principles Stack

None of the five principles requires the others. You can apply Principle 1 (name the coordinates) without changing your framework—add comments. You can apply Principle 2 (name the reductions) by choosing reduction functions that accept axis names. You can apply Principle 5 (the Inversion Rule) as a manual check when debugging gradient shapes.

But the principles compose. When you name coordinates (1), you can name what reductions consume (2). When you name broadcasts (3), you can check the backward pass against the forward pass (5). When you declare coordinate contracts (4), the compiler can check every call site against every principle simultaneously.

The five principles are a ladder. Each rung makes the next possible. The first three rungs are available in any framework—they require only discipline, not tooling. The last two require compiler support. But the first three, practiced consistently, catch the majority of coordinate bugs at code-review time, if not at compile time.

The habit begins at rung one. Name the coordinate. The rest follows.

One Table: The Coordinate Audit

Every tensor operation can be audited with four questions. They are not Einlang-specific. They work in any framework because they are questions about meaning, not syntax.

Question	What it catches	Chapter
Which coordinate is consumed?	Reduction over wrong axis	2, 8
Which coordinate is copied along?	Broadcast over wrong axis	2, 4
Can you trace a coordinate from source to destination?	Silent permutation/transpose	1, 5, 10
Does the backward reduction match the forward broadcast?	Gradient shape mismatch	8

Ask these four questions of any tensor line. The answers tell you whether the notation preserved the facts that correctness depends on.

Debugging with the Audit

The audit table is also a debugging tool. When a bug manifests as a wrong output shape or a wrong gradient, walk the audit questions backward from the symptom.

Symptom: gradient has wrong shape. The backward reduction doesn't match the forward broadcast. Check Question 4: which coordinate was broadcast forward? Sum over it backward. If the backward sum is over a different coordinate, the shapes will differ at exactly that coordinate. Trace the forward broadcast. Find where the coordinate was omitted. The omission is the bug.

Symptom: output values look normalized over the wrong axis. A softmax output summing to 1.0 over rows instead of columns. Check Question 1: which coordinate was consumed by the softmax reduction? If it was `dim=-1` but the intended coordinate is not the last one, the consumption is wrong. The fix is a `dim` change or a transpose before the softmax. The audit question tells you what to look for.

Symptom: loss is slightly worse after a refactoring, but all tensor shapes match. A coordinate was silently permuted. Check Question 3: trace one coordinate from the data entry point (data loader) through every operation to the loss. Find where the coordinate's position changed without the code recording the change. The position change is the bug. The name that wasn't there is the root cause.

Symptom: batch normalization behaves differently after adding a sequence dimension. The batch statistics are computed over the wrong set of coordinates. Check Question 1: which coordinates are reduced by `mean`? If the reduction consumed `batch` (correct) but also consumed `seq` (wrong), the statistics are being pooled across the wrong dimensions. In a positional API, this is a `dim` tuple audit. In a named API, the reduction bracket names the consumed coordinates, and adding a dimension doesn't change the bracket.

The four questions are a checklist. Run through them in order. The answer to at least one will be "I don't know from reading the code." The I-don't-know is a gap. The gap is where the bug lives.

Bug Bounty: Spot the Silent Bug

The best way to internalize what the coordinate habit catches is to try catching bugs yourself. Below are five real-world patterns. Each one compiles and runs without error in a positional API. Each one is wrong. For each: (1) find the bug, (2) explain why the positional compiler is silent, (3) write the Einlang version that would have caught it.

Bug 1: The Shifting Axis.

```
def process(x, w):
    x = x.transpose(1, 2)           # swap spatial and channel
    out = torch.softmax(x, dim=1)    # softmax over... which axis now?
    return out @ w
```

The transpose changed which axis sits at position 1. `dim=1` refers to a different coordinate before and after the transpose. The code runs. The softmax normalizes over the wrong axis.

In Einlang: `softmax[channel]` — the name doesn't shift when the tensor is transposed.

Bug 2: The Vanishing Dimension.

```
def aggregate(features):
    pooled = features.mean(dim=(2, 3))    # pool spatial dims
    return pooled @ classifier             # [b, c] @ [c, classes]
```

`features` is `(batch, channel, height, width)`. `mean(dim=(2, 3))` pools over positions 2 and 3—which are `height` and `width`. Correct. But six months later, someone adds a temporal dimension: `(batch, channel, time, height, width)`. `(2, 3)` now pools `time` and `height`. `width` survives. The code runs. The pooling is over the wrong coordinates.

In Einlang: `mean[height, width](features)` — the names are the same regardless of position. Adding `time` doesn't shift them.

Bug 3: The Broadcast That Shouldn't Be.

```
def apply_mask(scores, mask):
    return scores * mask
```

`scores` is `(batch, heads, seq_q, seq_k)`. `mask` is `(batch, 1, 1, seq_k)`. Broadcasting expands `mask` along `heads` (dim=1) and `seq_q` (dim=2). The code runs. But should the mask broadcast over `heads`? If different heads should see different masks—for example, head 0 attends locally, head 1 attends globally—broadcasting is semantically wrong. The positional broadcast is silent on *why* `heads` and `seq_q` are omitted.

In Einlang: `scores[batch, head, q, k] * mask[batch, k]` — the absence of `head` and `q` from `mask`'s brackets is recorded. The compiler confirms: broadcast over `head` and `q`. If that broadcast is not justified, the brackets make the unjustified claim visible.

Bug 4: The Gradient Gap.

```
def contrastive_loss(embeddings_a, embeddings_b):
    logits = embeddings_a @ embeddings_b.T   # [b, b]
    labels = torch.arange(logits.shape[0])
    return F.cross_entropy(logits, labels)
```

`cross_entropy` consumes `dim=-1` (the class dimension). The output is a scalar. But the user intended `loss[batch]` — per-sample loss for later weighting. `cross_entropy(logits, labels, reduction='none')` would preserve the batch dimension, but that's a keyword argument, not a coordinate specification. The default reduction consumed a coordinate silently.

In Einlang: `cross_entropy[class](logits[batch, class], labels[batch])` — the bracket says which coordinate is consumed. If you want `loss[batch]`, you see that `class` was consumed and `batch` survived. The output coordinates are explicit.

Bug 5: The Contract That Was Only a Comment.

```
# x: (batch, seq, feature)
# mask: (batch, seq) - broadcasts over feature
def masked_mean(x, mask):
    return (x * mask.unsqueeze(-1)).sum(dim=1) / mask.sum(dim=1, keepdim=True)
```

The comment says `mask.sum(dim=1)` reduces over `seq`. Someone refactors: `x` now has shape `(batch, seq, num_heads, feature)`. The `unsqueeze(-1)` still adds one dimension. `dim=1` now refers to `seq`—but wait, is `num_heads` at position 2? The `dim=1` reduction silently shifted from `seq` to… still `seq`? It depends on whether the refactor inserted `num_heads` before or after `seq`. The comment doesn't enforce anything. The position depends on convention, and convention is not checked.

In Einlang: `fn masked_mean[seq](x: [f32; batch, seq, ..rest], mask: [f32; batch, seq])` — the contract declares that `seq` is consumed, and the compiler checks it at every call site. Inserting a new dimension doesn't change which coordinate `seq` refers to.

Five bugs. None of them throw an error in a positional API. All of them produce wrong results that pass silently into downstream computations, where the symptom will be a slightly worse metric, not a crash.

The compiler checks described in this book—the five rules, the error codes, the lowering verifications—exist to catch these five bugs before they reach runtime. The compiler is not a luxury. It is a tool for making the coordinate habit machine-checkable.

A `dim=` argument in a positional codebase is an integer. Which coordinate it refers to is a question the code cannot answer—the answer lives in the programmer's head, or in the data loader's output shape, or in the documentation comment that may or may not be up to date. When the answer is "run the program to find out," the integer has already won. The gap between the integer and the identity is the bug's hiding place.

The Book's Vocabulary

This book built a naming system for the ideas it introduced. Here they are, gathered in one place.

Megaphone. A tensor speaks on the coordinates in its brackets and stays silent on all others. Broadcasting is the repetition of silence.

Consume. A reduction consumes a coordinate—eliminates it from the output. A broadcast consumes silence—repeats a value along a coordinate it does not have. A compiler consumes the name—burns it into an integer after all checks pass.

Shopping cart / Ledger. The forward pass records which coordinates each tensor omits. The backward pass reads the record in reverse: what was omitted forward becomes summed backward.

Skeleton. A normalization operation has a fixed coordinate structure: reduce some coordinates, broadcast statistics back, apply affine parameters. The skeleton is the same for BatchNorm, LayerNorm, InstanceNorm, GroupNorm, RMSNorm. Only which coordinates are reduced changes.

Firewood. A name is firewood for the compiler. It burns into an integer at lowering.

Panorama. The five forms of a name seen simultaneously: Source → IR → After Analysis → After Lowering → Generated Code. One name, five forms, zero loss of identity.

Coordinate habit. The reflex of pausing before a tensor operation and asking: which coordinate is being consumed, copied, or moved—and is its name in the code? Not a skill. A change in what you notice.

Shape-meanings gap. The shape says *how many*. The role says *which one*. Every framework knows the shape. None of them know the role. The gap is where the bugs live.

Inversion Rule. Forward broadcast becomes backward reduction. Forward reduction becomes backward broadcast. The coordinate names are the thread connecting the two directions.

Five check rules. Index Existence (Rule 1), Reduction Consistency (Rule 2), Broadcast Recording (Rule 3), Causality (Rule 4), Coordinate Contract (Rule 5). The five ways a name can be wrong, and the five questions the compiler asks to catch it.

Lowering. The final stage of the compiler: names become integers. The name is burned. The integer is correct because the name was verified.

These words are not decoration. They are the text's own coordinate system. Their job is the same as the job of the bracket: to give a fact a place to live, so it can be checked.

Three Scenarios

The legacy codebase. You inherit a PyTorch model with 200 occurrences of `dim=-1`. Spend one afternoon adding a comment at every `dim` argument: `# dim=-1 = feature`, `# dim=1 = channel`. One afternoon now beats ten afternoons of shape-tracing over the next year. The names need to be visible—not checked, not guaranteed, just visible.

The new project. Name your dimensions at the data loader, not in the model. The moment a tensor enters your program, attach coordinate names—in a docstring, in a convention (`batch` always first, `spatial` always last), in a project README. Six months from now, the convention tells you what `dim=1` means.

The bug investigation. Before you print another shape, write down which coordinate you think each dimension is. If `x.mean(dim=0)` is normalizing over `batch`, something is wrong—regardless of whether the shapes match. Which coordinate is consumed? Is the answer visible in the code? The question is the audit.

A Practical Guide for Non-Migrators

The coordinate habit does not require Einlang. It requires only that you put the name where the next reader can see it. Here are the patterns that work in PyTorch, JAX, and NumPy today.

At the data loader. The moment a tensor enters your program, its coordinates have identities. Record them before they are lost.

```
# x: (batch, channel, spatial) - order guaranteed by DataLoader
x = next(iter(dataloader))
```

This is a single line. It costs nothing to maintain—when the DataLoader changes, the comment is right next to the code that needs updating. Six months later, a reader tracing `x.mean(dim=1)` through the model sees the comment and knows: `dim=1` is `channel`.

At every reduction. A `dim=` argument consumes a coordinate. Which one? Write it.

```
h = x.mean(dim=1)           # dim=1 = channel
h = logits.softmax(dim=-1)  # dim=-1 = class
h = scores.sum(dim=(2, 3))  # dims=(2,3) = (height, width)
```

The comment records intent. When a refactor changes the shape, the comment is a flag: *this integer should match the coordinate named here*. If they no longer match, the reader knows to investigate.

At every broadcast. An operation between tensors of different ranks is a broadcast. Which coordinates are being replicated? Write the pattern.

```
# broadcasting: bias[channel] over (batch, channel)
out = x + bias

# broadcasting: scale[1, channel, 1, 1] over (batch, channel, height, width)
out = x * scale
```

The comment makes the silence audible. It records which coordinates the smaller tensor is silent on—the same information the compiler's broadcast ledger would record.

At every reshape. A reshape changes the coordinate layout. What was the layout before? What is it after? The names answer both questions.

```
# (batch, group, c_per_group, height, width) -> (batch, group, -1)
x = x.reshape(batch, group, -1)
```

The comment is the map from the old layout to the new one. Without it, the reader must reconstruct the layout from context—or run the code and print shapes.

At every permutation. A `permute`, `transpose`, or `swapaxes` reorders coordinates. Which coordinates moved? Write the correspondence.

```
# (batch, seq, heads, d_head) -> (batch, heads, seq, d_head)
x = x.permute(0, 2, 1, 3)
```

Or use einops, which records the correspondence as part of the expression:

```
x = rearrange(x, "batch seq heads d -> batch heads seq d")
```

The einops string is checked at runtime. The comment is not. Both record the intent. Choose based on whether you need the runtime check.

At function boundaries. A function that takes a tensor and returns a tensor has a coordinate contract. What does it consume? What does it produce? Write the contract in the docstring.

```
def layer_norm(x: Tensor) -> Tensor:
    """
    x: (batch, ..., feature)
    Returns: (batch, ..., feature)
    Normalizes over: feature
    """
    mean = x.mean(dim=-1, keepdim=True)
    var = x.var(dim=-1, keepdim=True)
    return (x - mean) / (var + eps).sqrt() * gamma + beta
```

A reader of the call site does not need to read the implementation. The docstring tells them which coordinate is consumed and which survive. The contract is not checked by the compiler, but it is checked by the next programmer—and that is enough to catch the mistake where the author intended `layer_norm` but the reader expects `instance_norm`.

In code review. Add one question to the checklist: *for each `dim=` argument in this diff, is the coordinate identity documented?* If the answer is no, ask the author to add a comment. The habit compounds.

When you can't name it. Write the number. But write the uncertainty too.

```
x = x.mean(dim=-1)  # dim=-1: last dim, currently feature-like; may change
```

The confession is better than silence.

Each of these patterns is a bridge. It does not check. It does not enforce. It does not survive a refactoring automatically. But it records. The name moves from the programmer's head into the source file, where the next reader—the colleague, the reviewer, the future you—can find it. The bridge is imperfect. But a bridge that exists is better than a bridge that was never built.

Epilogue · A Friend Named Einlang

> "Programs must be written for people to read, and only incidentally for machines to execute."
>
> — Harold Abelson and Gerald Jay Sussman, *Structure and Interpretation of Computer Programs*

You just reviewed the complete grammar. Now step back and ask what it adds up to.

In Chapter 5, Einlang was a name. A label for a notation under construction. Now, at the end, the name has a referent.

Einlang is not a language in the sense that Python is a language, or C++, or Rust. It does not aspire to run your web server or render your UI. It is a language built on three ideas: primitive expressions, means of combination, and means of abstraction—organized around a single purpose.

The idea is that **coordinates have identities, and those identities belong in the source code.**

Everything else in the language serves that idea. Reductions name the coordinate they consume. Broadcasts name the coordinate they replicate along. Permutations state the coordinate correspondence explicitly. Functions declare which coordinates they use by identity, and the compiler checks those declarations at every call site. Gradients preserve coordinate structure through differentiation. Recurrences make the direction of time a syntactic constraint.

The SICP quotation that opens this epilogue is famous for a reason. It states a truth that is obvious once you hear it and difficult to practice consistently: code is communication between humans before it is instruction to machines.

The coordinate habit is an application of that truth to tensor programming. When you write `x.mean(dim=1)`, you are communicating to the machine ("reduce axis 1") but not to the human ("eliminate channel"). When you write `mean[channel](x)`, you are communicating to both. The machine still knows what to do. The human now also knows what you intended. And the compiler can check that your intent is consistent with the tensor's actual coordinate structure.

The bet: that the extra keystrokes of naming coordinates are repaid, with interest, in debugging sessions avoided, in refactoring confidence gained, in the quiet satisfaction of reading code that says what it means.

SICP ends with a meta-circular evaluator—a Scheme interpreter written in Scheme—as if to say: you now understand the language deeply enough to implement it yourself.

The corresponding question here is smaller and harder.

Take the four habits into your next tensor program. Not an Einlang program—the language is young, the tooling is sparse, and deadlines don't move. Take them into PyTorch. Into JAX. Into whatever framework gets the work done.

When you write a reduction, pause. Ask: which coordinate am I eliminating? Is the name in the code?

When you write a broadcast, pause. Ask: which coordinate am I copying along? Is independence genuinely justified?

When you write a permutation, pause. Ask: can I trace one coordinate from source to destination without reconstructing the position map?

When you inspect a gradient, pause. Ask: does the backward reduction match the forward broadcast?

These questions cost seconds. The bugs they catch cost hours. The ratio is favorable.

The Life of a Name

A coordinate name in Einlang lives through four stages. It is **written** in source, as a letter between brackets: `[i, j]`, `[class]`, `[batch]`. It is **preserved** in the intermediate representation, stripped of syntax but keeping every name intact: `(let-decl (output C (i j)) ...)`. It is **verified** by analysis, where the compiler derives its range, checks its consistency across call sites, and records which operations consume it. And it is **burned** in lowering, translated into the integers that machines require: `class → axis=1`, `i → loop 0..b`.

Four stages. Written, preserved, verified, burned. At no point is the name decoration. At every point, it is load-bearing.

A decoration can be omitted without consequence. You can remove the comment `# dim=1 is channel` and the code still runs. You can rename the variable `spatial_features` to `x` and the compiler says nothing. Decorations are for humans. The machine does not read them.

A name in Einlang is not a decoration. The compiler reads it. The checker verifies it. The lowering pass translates it. If you write `softmax[class](logits)` and `logits` has no coordinate called `class`, the compiler stops. Not at runtime—at compile time. The name is part of the contract.

This is the difference between a comment and a coordinate. A comment records intent. A coordinate enforces it.

What Names Caught

Walk back through the book and ask, at each stage: what would a positional notation have let through?

Chapter 1: `x.mean(dim=1)`. The positional notation let it through for three weeks. The named version `mean[channel](x)` would have broken at compile time the moment `channel` moved to position 2, because the tensor would no longer have a coordinate called `channel` at the position the compiler expected. Or, more precisely: the compiler would have required the refactoring programmer to update the coordinate declaration, and that update would have surfaced the fact that `mean[channel]` was still referencing the old layout.

Chapter 2: `A + bias`. The positional notation broadcasts `bias` along whichever dimensions happen to be missing. If `A` changes from `(batch, feature)` to `(feature, batch)`, the broadcast silently flips. The named version `out[i, j] = A[i, j] + bias[j]` makes the omission visible: `bias` has no `i`, so it broadcasts over `i`. If `i` and `j` swap meanings upstream, the indexing pattern breaks visibly.

Chapter 3: `softmax(logits, dim=-1)`. When `batch_size == num_classes`, the square matrix test applies: `softmax(logits, dim=0)` and `softmax(logits, dim=-1)` both produce valid probability distributions. The named version `softmax[class](logits)` does not let you silently normalize over `batch`. The name `class` is either present on `logits` or it isn't.

Chapter 6: `u[t, i] = u[t+1, i] + f(...)`. In a positional recurrence, writing `t+1` instead of `t-1` produces a forward reference—a read from the future. The positional loop runs. The values are whatever was in memory. The named version rejects it: the causality check sees `t+1 > t` and halts.

Chapter 8: The gradient of a broadcast. Forward: `bias` omits `batch`, broadcasting over it. Backward: the gradient must sum over `batch` to recover `bias`'s shape. In a positional framework, this sum is implicit

in the autodiff engine. If the broadcast changes because the shape changed, the gradient sum changes with it—silently. In the named version, the coordinate sets tell you exactly what the gradient must sum over: `C` has `{i, j}`, `A` has `{i, k}`, sum over `{j}`. The set subtraction is checkable.

Chapter 11: GroupNorm's reshape chain: `x.reshape(N, G, C//G, H, W).mean(dim=(2,3,4))`. The positions `(2,3,4)` are only correct after the reshape. If the reshape changes, the positions change. The named version `mean[c_in_group, ..s]` names the coordinates directly. The reshape is unnecessary because the coordinates are separate from the start.

Chapter 12: Self-attention and cross-attention in PyTorch have identical code. The difference is only in the shapes of the tensors passed at runtime. The named version distinguishes `self_attention[seq, ...]` from `cross_attention[seq_q, seq_k, ...]` in the type signatures. A reader can see which is which without checking runtime shapes.

Every one of these bugs was shape-correct. Every one survived the checks that positional frameworks perform. Every one was caught by a name.

When You Don't Know the Name

Naming is a discipline—a habit to practice, an audit to perform. But there is a prior question: *what if you don't know what to call it?*

You are designing a new attention mechanism. An intermediate tensor has shape `(batch, heads, seq1, seq2, features)`. You stare at it. Is `seq1` the query sequence and `seq2` the key sequence? Are they symmetric? Should you call the last dimension `features` or `d_model` or `embedding`? You are not sure. The mechanism is still taking shape in your mind. Committing to a name feels premature—like naming a child before it is born.

So you write `dim=-1`. Not because you think positional notation is better. Not because you are lazy. But because the number `-1` does not ask you to decide what the dimension *means*. It asks only what the dimension *is*—the last one. And that you can answer.

This is **delayed commitment**. A name is a claim. A number is a placeholder. When you are in the early stages of designing a computation, you may not be ready to make the claim. The number lets you defer it. The number says: "I know where this dimension is. I do not yet know what it is."

The coordinate habit does not require you to name everything immediately. It requires you to name things before the code is read by someone else—including your future self. A number in a draft is a scaffold. A number in a merged pull request is a landmine. The difference is not whether the name exists at the moment of writing. The difference is whether the name exists at the moment of reading.

Practical advice: if you cannot name a dimension, write `dim=-1`—but write a comment next to it recording your uncertainty. `# dim=-1: last dim, currently feature-like but may change`. The comment is not a name. The comment is a flag. It tells the next reader: "the author was uncertain here. Check whether this dimension still means what you think it means." A number with a confession is better than a number with silent confidence.

And sometimes, after reflection, you realize the dimension genuinely does not have a stable identity. It is a transient intermediate that exists only inside this function, consumed by the next operation, never exposed to a caller. In that case, `dim=-1` may be the right choice permanently. Not every coordinate deserves a name. The coordinate habit is not a moral obligation. It is a judgment: *does the correctness of this operation depend on which coordinate this is?* If the answer is no, a number is fine. If the answer is yes, the name earns its keystrokes.

When you do commit to a name, the boundary from Chapter 14 still holds: the compiler cannot catch `softmax[batch]` where `softmax[class]` was intended. But a reader sees the wrong name and asks the question. The positional equivalent `softmax(logits, dim=0)` hides the error behind a number; the reader must reconstruct which coordinate axis 0 refers to, and the reconstruction may be wrong. The name earns its keystrokes twice: once by recording intent, once by making errors visible when intent is misrecorded.

The Invariant

Chapter 14 surveyed the landscape — assertions, einops, named tensors, compiler. The coordinate habit works at every step. It only asks: *is the name in the code?* The habit does not prescribe the tool. It prescribes the information.

Fifteen chapters. One invariant. Say it once more before you go:

Every tensor operation that depends on a coordinate's identity must record that identity in the source code.

This is not a language rule. It is a practice rule. It applies in Einlang, in PyTorch, in JAX, in NumPy, in any framework where tensors carry coordinates that mean different things. The notation you use determines *how* you record the identity—brackets, comments, variable names, einops strings—but the invariant is the same.

The invariant does not prevent all bugs. It prevents the class of bugs where the coordinate identity was lost before the operation was performed. A reduction over `dim=1` does not know it's reducing over `channel`. A reduction over `channel` knows. When the channel moves to `dim=2`, the first reduction silently becomes wrong. The second reduction becomes a compile error. The difference is whether the identity was recorded.

You now know how to record it. The rest is practice.

Day 100, Replayed

Two files. One refactoring. Three months.

`data.ein`, updated 90 days ago:

```
fn load_samples(path: &str) -> [f32; batch, spatial, feature] {
    // column 1 = batch, column 2 = spatial, column 3 = feature
    read_csv(path)
}
```

`model.ein`, untouched for 90 days:

```
fn predict(x: [f32; batch, channel, spatial]) -> [f32; batch, spatial] {
    mean[channel](x[batch, channel, spatial])
}
```

`main.ein`, today:

```
let x = load_samples("train.csv");
let y = predict(x);
```

Save.

```
error[E0061]: coordinate contract mismatch
  --> main.ein:2:16
   |
 2 | let y = predict(x);
   |                 ^
   |                 in call to `predict`
   |
   = argument `x`:
   =   provided:  batch, spatial, feature
   =   expected:  batch, channel, spatial
   |
   = missing coordinate:  `channel`
   = unexpected:           `feature`
   |
help: `channel` was renamed to `feature` in data.ein:1
help: update parameter declaration in model.ein:1:
      fn predict(x: [f32; batch, spatial, feature]) -> [f32; batch, spatial]
```

The positional equivalent:

```
x = x.mean(dim=1)
```

`dim=1` was `channel` before the refactoring and `spatial` after. It compiled. It ran. It passed integration tests. It deployed to staging. It failed silently in production for three weeks. Found at 3 AM on Day 100, by a human tracing one number backward through twelve layers.

The fix: rename `channel` to `feature` in model.ein:1. Ten seconds.

What the Coordinate Habit Does Not Solve

Chapter 14 catalogued the limits: names check consistency, not correctness; they do not replace testing; they cost keystrokes. Two limits belong here, at the end, because they define the boundary between this book's argument and its honest modesty.

Names don't write the program. The coordinate habit tells you to record which coordinate a reduction consumes. It does not tell you whether the reduction should be a mean or a sum, whether the normalization should be over `feature` or `batch`, whether the attention should be self or cross. Those decisions are modeling decisions. The names record them; they don't make them. A well-named wrong model is still a wrong model. The difference is that the names make the model's structure visible, so the next reader—the colleague, the reviewer, the future you—can see what the model assumed and judge whether the assumption still holds.

Einlang itself is young. The language in these pages is a research prototype—no CUDA backend, no package manager, no PyTorch integration. The coordinate habit works through comments, einops strings, and naming conventions in any framework today. But if you want to build the rest: the IR, the check rules, and the lowering pass described in Chapters 9 and 10 are a starting point. The distance from here

to a production compiler is measured in engineering years, not ideas. The ideas are in this book. They are ready. The compiler will catch up.

If You Want More

Three works shaped the thinking behind these pages.

Structure and Interpretation of Computer Programs (Abelson, Sussman, and Sussman) taught generations of programmers that a language is built from primitive expressions, means of combination, and means of abstraction—and that building a metacircular evaluator is the final proof of understanding.

Learn You a Haskell for Great Good (Miran Lipovača) showed that a book about a programming language can be warm, direct, and relentlessly focused on the reader's understanding rather than the author's expertise.

"Tensor Considered Harmful" (Aleksander Mądry, 2018) and the named-tensor work at Harvard and Stanford asked: what happens when tensor dimensions have names that the compiler can check?

They point to a destination—code that says what it means—and leave the walking to you.

Close the Cover

You are about to close this book.

Maybe you're at a desk. Maybe on a train. Maybe it's late and you're reading by a screen's glow. Wherever you are, there is a moment—right now, or in five minutes, or tomorrow morning—when you will look up from this page and return to the code you were writing before you opened the book.

What will be different?

Not the framework. You're still using PyTorch, or JAX, or NumPy. Not the deadline. It hasn't moved. Not the model architecture. The layers are the same. The loss function is the same. The optimizer is the same.

But something in how you read has changed.

You will type `x.mean(dim=1)` and pause. Not because the line is wrong—because `dim=1` is a number, and you know which coordinate it refers to, and you wonder whether the next person to read this line will know too.

You will write a broadcast and think: *which coordinate am I silent on?* Not because the framework requires you to answer—because you now know that the silence is a claim, and claims should be checkable.

You will trace a bug through a reshape-permute-reshape chain and think: *if these dimensions had names, this chain would be three lines instead of fifteen, and the bug would have been caught before runtime.*

You will read attention code and notice whether `seq_q` and `seq_k` are the same coordinate or different coordinates, because you now know that when they happen to have the same length at development time, the positional code for self-attention and cross-attention is identical.

You will not convert your entire codebase to Einlang. The language is young. The tooling is sparse. You have a deadline. But you will start putting names where they cost nothing and prevent everything: in comments, in variable names, in the structure of your tensor shapes. `# dim=1 is channel`. `x:`

`Tensor["batch", "channel", "spatial"]`. `rearrange(x, "batch channel spatial -> batch spatial channel")`.

A name in a comment can rot. But it rots slower than a name that was never written.

Close the cover. Open your editor. Read the first tensor line you see. Ask the question that the preceding pages have taught you to ask:

Where is the name?

If the answer is *nowhere*—you have found your starting point.