Learn You a Bend

Contents

Learn You a Bend (and Keep Your Haskell Hat On) 1
Run this tutorial in your browser . . . 2
0. Pack your compiler . . . 3
Two sorts of `main` . . . 3
1. Values, not a shopping list of commands . . . 4
2. Functions and the suspicious little plus . . . 4
3. Shapes in tiny labelled boxes . . . 5
A small but important matching fence . . . 6
4. Lists: one biscuit and the rest of the tin . . . 6
Why the shrinking argument goes first . . . 7
5. Functions in your backpack . . . 7
6. Quantities: the librarian is counting . . . 8
Kinds are the rules for a whole type . . . 9
7. Effects, without pretending printing is pure . . . 9
The same idea works for `Maybe` . . . 10
8. Two elves, two independent jobs . . . 11
Balance matters . . . 11
9. Laws: bring a proof, not just good intentions . . . 12
An induction in miniature . . . 12
What proofs do and do not buy you . . . 12
10. Bonus: a mutable array with one owner . . . 13
Where to wander next . . . 14
Pocket translation guide . . . 14
Running the whole shelf . . . 15

Learn You a Bend (and Keep Your Haskell Hat On)

A friendly, example-first tour of **Bend 2**: little programs, slightly silly metaphors, and a Haskell travelling companion. This is an original tutorial inspired by the approachable teaching format of *Learn You a Haskell*, not a translation of that book.

Bend wears Python-shaped clothes, carries a functional heart, and counts how often you use its belongings. If you know Haskell, some doors will look familiar. Do not assume they all open the same way.

No Haskell experience is required. The Haskell boxes are comparisons, not prerequisites; skip them on your first trip if you like.

Run this tutorial in your browser

From the repository root:

```
just open
```

This builds the page, its example modules, and a browser-adapted Bend compiler, starts a **localhost-only static-file server**, and opens the tutorial. Keep the terminal running; **Ctrl+C** stops the server. Use `just open 8001` if the default port is busy, or `just serve` to serve without opening a browser. Do not open `index.html` directly: browser modules and workers need HTTP.

The **Bend code boxes in the tutorial are editable directly**. Change the code and click **Run** (or Ctrl/Cmd+Enter): it compiles and executes that box, then displays the type and result. **Reset** restores the snippet and cancels any active run. There is no separate editor or compilation button, and a chapter can have several runnable boxes. Haskell comparisons and shell commands remain read-only.

Partial snippets run with the original chapter's remaining definitions, so a function, datatype, or indented statement can be tried on its own. Each box is independent; edits do not alter another box's context. Paste a complete program beginning with `import Base` to run without chapter context.

Compilation happens entirely in a fresh **compiler Web Worker**, using the pinned Bend 2 TypeScript normalchecker and JavaScript emitter with embedded `Base`. A **separate execution worker** runs the generated program. Neither compiler work nor edited program execution runs on the page's main JavaScript thread. Each phase is limited to five seconds; source is limited to 32 KiB UTF-8, generated JavaScript to 2 MiB, and formatted results to 64 KiB. These limits are not a general memory or JavaScript security sandbox.

The editor supports standalone **pure programs**: only `import Base`, a non-template `main` with zero runtime arguments, and returned data rather than terminal effects. Foreign C/JS, other module/package imports, and unsafe definitions are rejected, including indirect effects in unused helpers. Chapters 7 and 8 replace their IO entry points with labelled pure starters; the original downloads and CLI files are unchanged. JavaScript runs parallel calls sequentially.

Bend's normal type, affinity, termination, and proof checks apply to edits. Try changing chapter 9's law without fixing its proof: the browser compiler reports an error with a source location. Erased evidence displays `null`; that value is not itself proof validation. The independent Lean `--verdict` checker is **not** embedded.

Edited source is sent only between browser workers, never uploaded. There is **no compilation or execution server**, no compiler CLI in the browser, and no Node/Bun runtime dependency. Once built, the tutorial can be served by any static host with JavaScript module MIME types. The compiler's Apache-2.0 attribution is distributed in `generated/compiler.LICENSE` and `generated/compiler.NOTICE`; see `compiler/README.md` for the adaptation and pin-upgrade notes.

Without `just` installed globally, use `nix run .#just -- open`. All build, serving, and test tools come from the project flake; there are no CDN dependencies or remote execution services.

Each numbered `.bend` file is a complete program. Run it from the repository root. Read a chapter, run its program, then change one thing. This is a tutorial, not a spectator sport.

0. Pack your compiler

From the repository root:

```
nix develop
bend guide
bend src/01-values.bend
```

If Nix complains that `flake.nix` is untracked, follow the development-environment instructions in the workspace README. Do not silently switch to an old Bend 1 installation: Bend 1/HVM syntax is not this language.

Useful companions:

```
bend --help
bend base --types
bend base List
bend base Nat
```

This tutorial follows the upstream Bend 2 guide consulted on 2026-10-17. Bend is evolving quickly; **your installed compiler's `bend guide` and `bend base` are the authority** if a syntax or library detail differs.

Two sorts of `main`

A pure `main` can return a value:

```
import Base

def main() -> U32:
  42
```

`bend file.bend` checks it, normalizes its result, and prints the value. This is handy for tiny experiments, but normalization is not a performance benchmark.

A `main` returning `IO(Unit)` runs effects, such as printing, through the compiled runtime. We meet that version in chapter 7.

> **Haskell companion:** GHC's executable entry point has type `IO ()`. To explore pure Haskell expressions, you normally use GHCi. Bend's command also evaluates a pure `main`, so our early programs need no printing ceremony.

The compiler may print checking information alongside the result. When a chapter says "expect 42", it means the program's value, not an exact transcript of every compiler message.

1. Values, not a shopping list of commands

Run 01-values.bend:

```
bend src/01-values.bend
```

Its entire computation is:

```
def main() -> U32:
  (6 * 7 : U32)
```

Expect **42**. The body produces a value; there is no ordinary statement-level `return` here. We will use `return` later inside monadic `do` blocks.

`U32` is an unsigned 32-bit numeric type. The `: U32` on the arithmetic expression tells Bend which operations to use: multiplication here is `U32.mul`. Keep the spaces around operators. Bend does very little type inference, so even a typed function often needs an annotation on an operator expression.

Other literal families include `1.5` for `F32`, `3n` for `Nat`, `'c'` for `Char`, and `"hello"` for `String`. `Nat` represents natural numbers and will also give us a structurally shrinking counter.

> **Haskell companion:** `(6 * 7 :: Word32)` is a close comparison, using `Word32` from `Data.Word`. Haskell's numeric literals and operators use type classes and inference; `(6 * 7 :: Integer)` chooses a different, arbitrary-precision type. Bend's `(a * b : U32)` explicitly selects operations for `U32`. Do not bring `Integer` assumptions to a `U32` party.

Try it: replace multiplication with addition, then add parentheses to compute `((6 + 1 : U32) * 7 : U32)`. Predict the result before running it.

2. Functions and the suspicious little plus

Run 02-functions.bend. Expect **49**.

```
def square(+x: U32) -> U32:
  (x * x : U32)

def main() -> U32:
  square(7)
```

Read the signature aloud: "`square` receives a reusable `U32` named `x` and produces a `U32`." A call uses parentheses: `square(7)`.

Why `+x`, rather than `x`? Bend variables are **affine by default**: they may be used at most once. We use `x` twice in multiplication, so we explicitly request reuse. `U32` supports copying; chapter 6 explains how types tell us that.

The `+` in a binder is not addition, and it is not a sign on the number. It is permission to use that binding more than once.

> **Haskell companion:**

```haskell
square :: Word32 -> Word32
square x = x * x
```

Ordinary Haskell lets you use `x` zero, one, or many times without an annotation. Bend makes that resource choice explicit. Haskell's optional `LinearTypes` extension is related territory, but linear "exactly once" and Bend's affine "at most once" are not identical rules.

A local binding looks like this:

```
def main() -> U32:
  answer = square(7)
  (answer + 1 : U32)
```

Think of `answer = ...` as a `let`, not an update to a global mutable variable. Its final expression gives the result.

Try it: remove `+` from `square` and read the compiler's complaint. Restore it afterwards. Then define a `cube` function; how often does its argument appear?

3. Shapes in tiny labelled boxes

Run 03-shapes.bend. Expect **25**.

```
type Shape is Data:
  Circle{r: U32}
  Square{s: U32}
```

This is an algebraic datatype: a `Shape` is either a circle carrying a radius or a square carrying a side length. `Circle` and `Square` are constructors, not subclasses. `Square{5}` constructs a value.

`is Data` says values of this type may be copied. It does **not** make every binding automatically reusable.

To inspect a shape, use pattern matching:

```
def area(x: Shape) -> U32:
  match x:
    case Circle{+r}:
      (3 * r * r : U32)
    case Square{+s}:
      (s * s : U32)
```

Each pattern opens one kind of box and names its contents. `+r` and `+s` let the selected numeric field be reused. The circle formula deliberately uses a crude approximation of pi: this is a datatype lesson, not a geometry certification.

Haskell companion:

```haskell
data Shape = Circle Word32 | Square Word32

area :: Shape -> Word32
```

```
area (Circle r) = 3 * r * r
area (Square s) = s * s
```

Both languages describe alternatives with constructors and handle them with patterns. Bend adds a kind (`Data` here) and explicit reuse annotations. Constructor braces in Bend are construction/pattern syntax, not Haskell record-update syntax.

A small but important matching fence

Current Bend has no `if` syntax: branch by matching `True{}` and `False{}`. It also restricts the thing you match on: use a parameter or a pattern-bound variable, in binder order. You cannot directly write `match area(shape):`, nor introduce a local computation before matching a parameter.

For a computed decision, pass the computed value to a helper that matches its parameter. Haskell's unrestricted `case expression of` is not a safe syntax guide here.

Try it: change `main` to `area(Circle{4})`. Expect **48**. Then add a `Rectangle{w: U32, h: U32}` constructor and its matching branch.

4. Lists: one biscuit and the rest of the tin

Run 04-lists.bend. Expect **10**.

A list is empty, or it contains a head and another list:

- `Nil{}` is empty.
- `Con{h, t}` contains head `h` and tail `t`.
- `[1, 2, 3]` is list-literal sugar.
- `h <> t` constructs a list by putting `h` before `t`.

Our sum carries an accumulator:

```
def sum(xs: List<U32>, acc: U32) -> U32:
  match xs:
    case Nil{}:
      acc
    case Con{h, t}:
      sum(t, (acc + h : U32))
```

Walk through `sum([1, 2], 0)`: consume `1`, continue with `[2]` and accumulator `1`, consume `2`, then reach the empty list and answer `3`.

Each variable in a branch is used at most once. Different branches are alternatives, not two executions that both consume the same value.

Haskell companion:

```
sumAcc :: [Word32] -> Word32 -> Word32
sumAcc [] acc = acc
sumAcc (h:t) acc = sumAcc t (acc + h)
```

> This is the same accumulator pattern. Haskell writes `[]` and `(:)` where Bend patterns use `Nil{}` and `Con{h, t}`. Haskell's `<>` usually means semigroup combination; Bend's `<>` here means **cons**, not list concatenation.

Why the shrinking argument goes first

Bend checks termination. In the recursive call, `t` is a smaller structural part of `xs`, obtained by pattern matching. That is the compiler's evidence that we eventually run out of biscuits.

Recursive arguments are checked left to right: arguments stay unchanged until one becomes a smaller part of its parameter; arguments after that may change. Putting `xs` first lets the accumulator change after the shrinking argument.

Tail calls compile to loops. You do not need a mutable loop counter to repeat work.

> **Haskell contrast:** Haskell permits nontermination, mutual recursion, and infinite lazy lists. Ordinary safe Bend recursion must terminate, and mutual recursion is currently disallowed. Do not translate `repeat 7` into a supposedly infinite Bend list. `@unsafe` can relax recursion restrictions, but sacrifices proof guarantees; it is not our tutorial escape hatch.

Try it: sum the empty list; expect **0**. Then sum `[4, 5, 6]`; expect **15**. For a stretch exercise, write a function over `Nat` using `0n` and `1n+p` patterns instead of attempting to convince the checker that arbitrary `U32` subtraction is structural recursion.

5. Functions in your backpack

Run 05-functions-as-values.bend. Expect **44**.

Functions can be returned as values:

```
def adder(k: U32) -> U32 -> U32:
  x => (x + k : U32)
```

`x => ...` is a lambda. The returned closure remembers `k`; `adder(2)` gives us a function that adds two.

> **Haskell companion:** `adder k = \x -> x + k`. Haskell often writes the equivalent `adder k x = x + k`. Both languages can make functions that capture a local value.

But Bend has a surprise: **a runtime closure is affine**. It can be called at most once, even if everything it captures is copyable. Partial applications, such as `U32.add(2)`, are closures too. You cannot fix that by putting `+` on a closure binding: its type does not permit copying.

Top-level definitions can be called freely. For reusable higher-order code, Bend also has compile-time templates:

```
def twice(~f: U32 -> U32, x: U32) -> U32:
  f(f(x))
```

`~f` is substituted as syntax at compile time rather than passed as a runtime closure. That is why it can appear twice.

```
def main() -> U32:
  add2 = adder(2)
  twice(~(x => (x + 1 : U32)), add2(40))
```

`add2` is called once. The template adds one twice. Result: 44.

Template arguments must be **closed**: they can mention top-level definitions, but not a caller's local variable. Template parameters go first. Each distinct set of template arguments specializes the template, so this flexibility can cost compiled code size. Templates may call only templates declared above them.

> **Haskell contrast:** ordinary Haskell's `twice f x = f (f x)` freely reuses the runtime function `f`. Bend's `~f` is not just a different spelling of that argument: it changes when and how the function is supplied. Base's `List.map` uses templates for this reason; inspect `bend base List` before guessing its call signature.

Try it: replace the incrementing template with `~square`, after adding the top-level `square` definition from chapter 2 above `twice`. Use a small input! Separately, try calling `add2` twice and examine the rejection.

6. Quantities: the librarian is counting

Run 06-quantities.bend. Its result contains **three copies of 7**.

```
def replicate(-A: Data, n: Nat, +x: A) -> List<A>:
  match n:
    case 0n:
      Nil{}
    case 1n+p:
      x <> replicate(A, p, x)
```

Now we are generic: `A` is a type, not a particular number type. The call is `replicate(U32, 3n, 7)`.

Binder	Meaning	What to remember
`x: A`	Affine	Use zero or one times. Dropping is allowed.
`+x: A`	Reusable	Requires a type whose values support copying.
`-A: Data`	Erased	Seen by the checker, removed at runtime; usable in types/proofs and erased positions.
`~f: ...`	Template	Compile-time syntax substitution, not a runtime ownership quantity.

The type argument `A` is erased. We can pass it to the recursive call's erased parameter without manufacturing a runtime type object. The value `x` appears both in the new list node and in the recursive call, so it needs `+`.

Kinds are the rules for a whole type

A kind tells us how reusable the values of a type can be:

- `Type` is shorthand for `Kind(&1)`: affine values, such as closures and arrays.
- `Data` is shorthand for `Kind(&2)`: values that can be copied.
- `&0`, `&1`, and `&2` describe erased, affine, and reusable quantities.

Generic containers can be quantity-aware too. Base's full list type is `List<a, A>`, with `A: Kind(a)`. `List<U32>` is an affine-list shorthand; `+List<U32>` denotes the reusable quantity specialization. A `+` on a **type expression** and a `+` on a **variable binder** are related, but are not the same syntactic operation.

You do not need every kind formula to start writing programs. Remember the useful distinction: "this type *can* be copied" is not "this binding *has* permission to be used twice."

> **Haskell companion:** `replicate :: Int -> a -> [a]` expresses genericity through a type variable, normally inferred and implicitly quantified. Bend exposes an erased type argument and its copyability requirement. `Data` here is a Bend kind, **not** Haskell's `Data.Data` class or the `data` keyword.

Try it: change the counter to `0n`, then to `5n`. Remove `+` from `x` to see why two uses require permission. Restore it before moving on.

7. Effects, without pretending printing is pure

Run 07-effects.bend. Expect:

```
Hello, functional adventurer!
```

A pure function calculates a value. An `IO` computation describes effects which the runtime performs. Bend is pure, but purity does not mean we are forbidden to say hello.

```
def greeting(name: String) -> String:
  "Hello, " ++ name ++ "!"

def greet(name: String) -> IO(String):
  do IO<String>:
    return greeting(name)

def main() -> IO(Unit):
  do IO<Unit>:
    text : String <- greet("functional adventurer")
    IO.print(text)
```

`greeting` is the shared pure function used by both the browser panel and the IO wrapper. `++` joins strings. `return` wraps a pure result in the monad; it is not an imperative early exit. `<-` binds the result of an effect. Bend requires the bind's type annotation. `IO.print` is a `Unit` step; `Unit` is the no-interesting-result type.

> **Haskell companion:**
>
> ```haskell
> greet :: String -> IO String
> greet name = pure ("Hello, " ++ name ++ "!")
>
> main :: IO ()
> main = do
>   text <- greet "functional adventurer"
>   putStrLn text
> ```
>
> The sequencing idea is familiar, but Bend spells `Unit` rather than `()` and explicitly names the monad/result in the `do` header. Haskell usually infers the binding's type.

The same idea works for `Maybe`

The file also defines:

```
def add_strs(a: String, b: String) -> Maybe<&2, U32>:
  do Maybe<&2, U32>:
    x : U32 <- U32.read(a)
    y : U32 <- U32.read(b)
    return (x + y : U32)
```

`U32.read` returns an optional result. If either parse is `None{}`, the whole block returns `None{}`. Otherwise, it returns `Some{sum}`. This is a tidy pipeline with failure, not an exception secretly hiding behind a type.

To try it, replace the file's `main` with:

```
def main() -> Maybe<&2, U32>:
  add_strs("40", "2")
```

Expect a `Some` containing **42**. Then try `"biscuits"` instead of `"2"`; expect `None`.

> **Haskell companion:** this resembles `do` over `Maybe`, using `readMaybe` from `Text.Read`. Bend desugars the block through named `Maybe.bind` and `Maybe.pure` definitions, including the explicit `&2` argument. It is not relying on Haskell's `Monad` type-class instance inference. The same naming convention works for custom monads.

IO handles are affine too: effects on a file or socket hand back its handle beside the result. Concurrency (`IO.fork` / `IO.join`) is a separate concern from the pure parallel calls in the next chapter.

Try it: print a second message after the greeting. Then run both `Maybe` experiments. Notice that the greeting function itself never prints anything.

8. Two elves, two independent jobs

Run 08-parallelism.bend. Expect **1024**.

```
def pow2(+n: Nat) -> U32:
  match n:
    case 0n:
      1
    case 1n+p:
      a b = pow2(p) pow2(p)
      (a + b : U32)
```

Two results double the previous answer. At zero, the answer is one. After ten doublings, it is 1024.

The unusual line is the point:

```
      a b = pow2(p) pow2(p)
```

This is a **parallel call**, a binary fork followed by a join. Each recursive job has the same size. Since `n` is reusable, matching it gives reusable fields, allowing both jobs to receive `p`.

This is intentionally a toy workload: calculating powers of two this way repeats lots of work. Parallelism does not rescue a poor algorithm. We use it to see the fork-join structure, not to recommend a production exponentiation function.

> **Haskell companion:** you might expose parallel pure work with strategies from `Control.Parallel.Strategies`. Two ordinary Haskell `let` bindings do not by themselves request multicore scheduling. Likewise, not every pair of ordinary Bend calls is the explicit parallel form above. Think "fork independent pure calculations", not "start two IO threads".

Balance matters

Bend's current scheduler is binary fork-join, without moving tasks between cores after assigning them. Uneven work can leave one worker waiting while the other struggles with a mountain of biscuits. Aim for independent jobs of roughly equal cost.

The JavaScript target runs these calls sequentially. The native target uses CPU parallelism; a call spelled `pow2!(10n)` requests GPU execution for it and its nested parallel calls. GPU compilation needs the platform toolchain; machines without a GPU can run those calls on the CPU. This tutorial deliberately avoids `!` so you do not need a GPU setup.

To explore the native runtime explicitly, compile outside the tutorial's source files:

```
bend src/08-parallelism.bend -o /tmp/bend-tutorial-parallel
/tmp/bend-tutorial-parallel --threads 2
```

Small examples can spend more time scheduling than calculating. Do not infer a speedup from this one! Read `bend guide shaders` before building a serious parallel or GPU workload.

Try it: use `5n`; expect **32**. Keep the counters small. Contrast this branching computation with chapter 4's single recursive chain: where would independent work come from in each?

9. Laws: bring a proof, not just good intentions

Run 09-laws.bend:

```
bend src/09-laws.bend
```

The example states and proves that adding zero to any natural number changes nothing:

```
law add_zero:
  for x: Nat
  {Nat.add(x, 0n) == x : Nat}
```

This is a claim for **every** `x`, not a test of two or three convenient numbers. The paired `def add_zero(x)` supplies evidence of that claim.

Important punctuation alert: `{a == b : T}` is an **equality type**. It is not a Boolean comparison. To compare values at runtime, use operations such as `U32.is_eq(a, b)` or `Nat.is_eq(a, b)`.

An induction in miniature

The proof in the file follows the shape of `Nat`:

1. For `0n`, both sides compute to the same thing. `{==}` is reflexivity evidence.
2. For `1n+p`, recursively prove the law for the smaller `p`.
3. Use `%add_zero(p) : ...` to rewrite with that evidence. The `_` marks the occurrence to rewrite in the goal.
4. Finish with `{==}` when the resulting sides compute identically.

A proof is a definition whose type is the proposition. Bend has no tactic script language here; pattern matching, recursion, and rewriting are doing the work.

> **Haskell companion:** a QuickCheck property might say `\x -> x + 0 == (x :: Integer)`. QuickCheck tests generated cases; this Bend law requires a proof covering all natural numbers. Ordinary Haskell signatures do not by themselves demand this equality proof. More advanced Haskell encodings and external verification tools exist, but the closest mental model here is a dependently typed proof assistant such as Lean, not ordinary `Eq` or a test suite.

The file's `main` asks for evidence at `2n`, but that is merely a use of the already general proof. The law is not limited to the value in `main`.

What proofs do and do not buy you

A checker can establish the claims you actually wrote, under its trusted semantics and dependencies. It cannot guess a missing specification, fix a wrong law, or make foreign effects magically correct. `@unsafe` code is outside Bend's proof guarantees.

Tests remain useful for IO behavior, integrations, and whether your laws match the intended requirements.

For this lesson, keep the law and proof together. Larger projects conventionally separate human-written `LAWS.bend` from implementation and `PROOF.bend`; the proof file must import the laws. `bend PROOF.bend` is then the proof gate. Current Bend also offers `--verdict` to recheck through its small BendTT kernel.

Try it: change the law's right-hand side to `1n+x` while leaving the proof alone. It must not pass. Restore the original law. Then remove a proof branch and read the reported goal or error instead of treating the checker as an obstacle to work around.

10. Bonus: a mutable array with one owner

Run 10-arrays.bend:

```
def array(value: U32, index: U32) -> Array<U32> & U32:
  a = [0 : U32*8n]
  a[5] <- value
  a[index]

def main() -> Array<U32> & U32:
  array(42, 5)
```

This creates eight zero-filled slots, writes slot five, then reads it. The result is a **pair containing the array and 42**, not just 42. The checker may display the array structurally rather than as a friendly literal.

`A & B` is a pair type. An array read returns ownership of the array along with the element, so the array has not vanished after you inspect it.

`Array<T>` is affine, with one owner. That ownership lets a write update storage in place without observable shared mutation. A write on its own line rebinds the array; this syntax does not grant arbitrary mutation to every binding in Bend.

> **Haskell companion:** ordinary immutable arrays and `Data.Vector` values can be shared freely; mutable structures can be managed through `ST` or `IO`. Bend uses affine ownership to make this array update compatible with purity. It is neither "all Haskell updates copy" nor "Bend is secretly imperative Python". The resource rules are the key difference.

There are sharp edges: the slot count after * must be a power of two, indexes wrap around, and current `a[i]` sugar assumes `Array<U32>`. For other element types or an explicit copy, inspect `Array.get`, `Array.swap`, `Array.set`, and `Array.clone` with `bend base Array`. Do not assume Haskell-style bounds errors or automatic copying.

Try it: change the written value to 99; expect the returned element to be **99**. Then move the read to a different, unwritten slot and expect **0**.

Where to wander next

You have met values, functions, algebraic data, structural recursion, closures, quantities, monads, parallel calls, proofs, and ownership-aware arrays. That is a respectable backpack for a first walk.

A few small next projects:

- Extend the shape datatype and check several concrete areas.
- Write `Nat`-recursive list helpers before reaching for unsafe recursion.
- Parse two inputs with `Maybe`, then keep printing confined to `IO`.
- State a law for a tiny function and prove it by matching its input.
- Inspect Base's quantity-aware container signatures rather than guessing their Haskell equivalents.

Pocket translation guide

Haskell idea	Bend 2 neighbour	Important difference
`f x`	`f(x)`	Python-shaped calls do not imply Python semantics.
`let x = ...`	`x = ...`	Bindings are affine by default.
`data` and constructors	`type ... is Data/Type` and `K{...}`	The kind determines copyability.
`[]`, `h:t`	`Nil{}`, `Con{h, t}`, `h <> t`	`<>` is cons here, not semigroup append.
`\x -> ...`	`x => ...`	Runtime closures may be called at most once.
Polymorphic functions	Erased type/kind parameters	More explicit arguments and fewer inferred types.
Reusing a function argument	A ~ template argument	Compile-time substitution, with closed arguments.
`IO ()` and `do`	`IO(Unit)` and `do IO<Unit>:`	Explicit result annotations and affine handles.
`Maybe` pipelines	`do Maybe<&2, T>:`	Named `bind`/`pure` conventions rather than instance inference.
Recursive definitions	Structural recursion	Termination required; no ordinary mutual recursion.
QuickCheck properties	Laws plus proof definitions	Sampling examples is not proving a universal claim.
Parallel strategies	Binary parallel calls	Balanced work and target/runtime choices matter.
`ST`/mutable arrays	Affine `Array<T>`	Ownership is returned alongside reads.

Keep the official guide, `bend guide`, and `bend base` nearby. When an analogy and the

compiler disagree, the analogy loses. Even a charming analogy.

Running the whole shelf

After entering the development shell, run all chapter files from the repository root:

```
for file in src/*.bend; do
  bend "$file" || break
done
```

For the browser build and its regression tests, use `just test`. For the original CLI programs, use `just check`. The browser build copies the numbered chapters unchanged into `generated/chapters/` with import-safe names, because the pinned Bend 2.0.37 compiler rejects digit-leading module imports; the source filenames remain unchanged. `browser-examples.bend` delegates to those chapter functions and compiles to `generated/examples.mjs`. Build artifacts are ignored by Git.