Lecture 05: Functional Programming in Lean

Recursive definitions, higher-order functions, monads, and interpreters

Duolei Wang, CUHK-Shenzhen

July 17, 2026

Learning Goals

By the end of this lecture, students should be able to:

- Read recursive Lean definitions as mathematical equations over inductive data.
- Write and test basic list-processing functions using pattern matching.
- Explain why monads are useful for failure, state, and input/output.
- Build a small expression evaluator and understand how it scales toward a project.

This deck is written to be readable after class as a permanent file: each major block begins from intuition, then gives a formal pattern, then points to a nearby exercise in the local Lean files.

Four-Part Outline

1. Functional programming foundations: equations, recursion, lists, and folds.
2. More inductive data structures and monads: trees, graphs, `Option`, `Except`, `StateM`, and `IO`.
3. Expression systems and meta level: syntax trees, environments, evaluators, macros, and elaborators.
4. Project outline and reading: how to extend today's material into a small Scheme-like interpreter.

Three Lecture Blocks

- Block I: 50 minutes on functional-programming foundations through lists and termination, followed by about 10 minutes for list exercises.
- Block II: 50 minutes from inductive trees and graphs to monad examples, followed by about 10 minutes for a pause and monad exercises.
- Block III: 50 minutes on expression systems, the mini-language project, and Lean's meta level, followed by about 10 minutes for project requirements and questions.

The rhythm is intentional: students see a full idea, try a nearby problem while the pattern is still fresh, and only then move to the next layer of abstraction.

Outline of Sections

Outline of this Section

Functional Programming: A Paradigm Shift

Contrasting classical sequential programming with FP:

- **Execution Model:**
  - *Sequential:* Mutating state step-by-step (`x = x + 1`).
  - *FP:* **Pure mathematical mappings** (no hidden side effects).

Functional Programming: A Paradigm Shift

Contrasting classical sequential programming with FP:

- **Execution Model:**
  - *Sequential:* Mutating state step-by-step (`x = x + 1`).
  - *FP:* **Pure mathematical mappings** (no hidden side effects).
- **Data Structures:**
  - *Sequential:* Memory pointers and mutable arrays.
  - *FP:* **Algebraic Data Types (ADTs)** (sums and products).

Functional Programming: A Paradigm Shift

Contrasting classical sequential programming with FP:

- **Execution Model:**
  - *Sequential:* Mutating state step-by-step (`x = x + 1`).
  - *FP:* **Pure mathematical mappings** (no hidden side effects).
- **Data Structures:**
  - *Sequential:* Memory pointers and mutable arrays.
  - *FP:* **Algebraic Data Types (ADTs)** (sums and products).
- **Advanced Typing (Lean 4):**
  - ADTs elevate to **GADTs / Inductive Families**.
  - Types carry *logical invariants* (e.g., length-indexed Vectors).

Functional Programming: Definition

Definition (Functional Programming)

A declarative programming paradigm where applications are constructed by composing **pure functions**.

Functional Programming: Definition

Definition (Functional Programming)

A declarative programming paradigm where applications are constructed by composing **pure functions**.

- **Immutability:** Data cannot be modified post-creation.
- **Referential Transparency:** Identical inputs strictly yield identical outputs.

Functional Programming: Definition

Definition (Functional Programming)

A declarative programming paradigm where applications are constructed by composing **pure functions**.

- **Immutability:** Data cannot be modified post-creation.
- **Referential Transparency:** Identical inputs strictly yield identical outputs.

Unlike sequential programming, this is not just a style constraint. It is a **strict mathematical contract** enabling formal software verification.

The Curry-Howard Correspondence

Why use FP in Lean 4? The **Curry-Howard Isomorphism** establishes a deep structural equivalence between code and logic:

- **Propositions as Types:** A theorem is represented as a computational *Type*.
- **Proofs as Programs:** A pure function returning that Type is a verified *Proof*.

The Curry-Howard Correspondence

Why use FP in Lean 4? The **Curry-Howard Isomorphism** establishes a deep structural equivalence between code and logic:

- **Propositions as Types:** A theorem is represented as a computational *Type*.
- **Proofs as Programs:** A pure function returning that Type is a verified *Proof*.

The Core Takeaway

Because our data is defined inductively, computation is pattern matching. **The key to functional programming is writing inductive proofs.**

Example: The Factorial Function

Example

Define a function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n) = n!$.

Example: The Factorial Function

Example

Define a function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n) = n!$.

Proof.

The standard mathematical definition uses a product series:

$$f(n) := \begin{cases} 1 & \text{if } n = 0 \\ \prod_{i=1}^{n} i & \text{otherwise} \end{cases}$$

This yields the result theoretically, but how do we compute it? □

The Imperative Approach (Impure)

In sequential programming, we typically compute this via state mutation:

```
def f(n: int) -> int:
    acc: int = 1
    for i in range(1, n + 1):
        acc = acc * i  # State Mutation
    return acc
```

The Imperative Approach (Impure)

In sequential programming, we typically compute this via state mutation:

```
def f(n: int) -> int:
    acc: int = 1
    for i in range(1, n + 1):
        acc = acc * i  # State Mutation
    return acc
```

- **The Problem:** Continuously modifying `acc` is an *impure* operation.
- **The FP Question:** Can we compute this using only mathematical mappings, without modifying variables?

The Inductive Approach

Instead of a loop, we define *f inductively* (recursively):

$$f(n) := \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{otherwise} \end{cases}$$

The Inductive Approach

Instead of a loop, we define *f inductively* (recursively):

$$f(n) := \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{otherwise} \end{cases}$$

Inductive Proof that $f(n) = n!$.

- **Base case ($n = 0$):** $f(0) = 1 = 0!$. (Trivial)

The Inductive Approach

Instead of a loop, we define *f inductively* (recursively):

$$f(n) := \begin{cases} 1 & \text{if } n = 0 \\ n \cdot f(n-1) & \text{otherwise} \end{cases}$$

Inductive Proof that $f(n) = n!$.

- **Base case ($n = 0$):** $f(0) = 1 = 0!$. (Trivial)
- **Inductive step:** Assume $f(k) = k!$. For $k+1$:

$$f(k+1) = (k+1) \cdot f(k) = (k+1) \cdot k! = (k+1)!.$$

Thus, $f(n) = n!$ for all $n \in \mathbb{N}$. □

Functional Translation

The Lean code is an **exact 1:1 translation** of the inductive proof. The pattern match happens on the data structure `Nat` itself:

Lean 4:

```
def f : Nat -> Nat
  | Nat.zero   => 1
  | Nat.succ k => (k + 1) * f k
```

Functional Translation

The Lean code is an **exact 1:1 translation** of the inductive proof. The pattern match happens on the data structure `Nat` itself:

Lean 4:

```
def f : Nat -> Nat
  | Nat.zero   => 1
  | Nat.succ k => (k + 1) * f k
```

Here "exact" means the recursion equations match the proof. Unlike a Python `if/else`, this is not branching on `Bool`; it is constructor matching on `Nat`.

Pattern Matching vs. Boolean Branching

- A Python `if/else` first evaluates a Boolean condition such as `n == 0`; the branch is chosen by a value of type `Bool`.
- Lean's first definition of factorial pattern matches on the input data structure itself:

  `Nat.zero` or `Nat.succ k`.

- This is why the recursive equation exposes the smaller argument k automatically.

Why Is This Recursive Definition Allowed?

In untyped lambda calculus, recursion can be explained by fixed-point combinators such as the Y-combinator: a function receives a self-reference and unfolds itself. Lean cannot accept unrestricted fixed points, because nonterminating definitions would break the logic. Instead, recursive definitions must be justified by an eliminator or a well-founded measure.

- For factorial, the recursive call is on k, structurally smaller than `Nat.succ` k.
- Conceptually, Lean compiles this definition through the natural-number eliminator, often written as $\mathrm{ind}_{\mathbb{N}}$ or Nat.rec.

Mathematical Recursion (Collatz Conjecture)

Recursion is natural in mathematics. Consider the Collatz sequence:

$$C(n) := \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

Mathematical Recursion (Collatz Conjecture)

Recursion is natural in mathematics. Consider the Collatz sequence:

$$C(n) := \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n+1 & \text{if } n \text{ is odd} \end{cases}$$

- We evaluate by applying the definition repeatedly.
- **The Issue:** Does it terminate for all $n \geq 1$? (Open math problem)
- **In Lean 4:** This highlights why *well-founded termination proofs* are mandatory for recursive functions (covered later).

Takeaway: Computation as Structural Induction

We can conclude from above examples that:

- Rather than iterating over an infinite domain, we define functions over **finite inductive cases**.

Takeaway: Computation as Structural Induction

We can conclude from above examples that:

- Rather than iterating over an infinite domain, we define functions over **finite inductive cases**.
- This logic scales to *any* inductively defined data structure.

Takeaway: Computation as Structural Induction

We can conclude from above examples that:

- Rather than iterating over an infinite domain, we define functions over **finite inductive cases**.
- This logic scales to *any* inductively defined data structure.
- *Advanced Note:* Principles like Path Induction in Homotopy Type Theory (HoTT) rely on these exact computational foundations.

Takeaway: Computation as Structural Induction

We can conclude from above examples that:

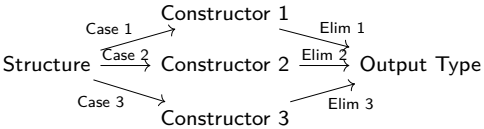
- Rather than iterating over an infinite domain, we define functions over **finite inductive cases**.
- This logic scales to *any* inductively defined data structure.
- *Advanced Note:* Principles like Path Induction in Homotopy Type Theory (HoTT) rely on these exact computational foundations.

The Two Pillars of ADTs

- **Constructors:** The specific rules for building a type.
- **Eliminators:** The pattern-matching rules for decomposing a type.

Visualizing Constructors and Eliminators

Just as a mathematician handles every case of an axiom, a functional programmer handles every constructor of a type:



Strict Typing Rule: The output type must be identical across all eliminator branches.

The Natural Number Inductor $\mathrm{ind}_{\mathbb{N}}$

Before equality, recall what an eliminator looks like for a familiar type. For natural numbers, the induction principle says:

$$\mathrm{ind}_{\mathbb{N}} : C(0) \to \left(\prod_{n:\mathbb{N}} C(n) \to C(\mathrm{succ}(n)) \right) \to \prod_{n:\mathbb{N}} C(n).$$

To define or prove something for every natural number, give the zero case and the successor step. Pattern matching code is the programming face of this eliminator.

Equality as an Identification Type

Following Rijke's *Introduction to Homotopy Type Theory*, equality is treated as an **identification type**.

Equality as an Identification Type

Following Rijke's *Introduction to Homotopy Type Theory*, equality is treated as an **identification type**.

In Homotopy Type Theory (HoTT) and Lean, **an equality proof is a term of a type**.

- For any type A and elements $x, y : A$, there is an identification type, denoted $x =_A y$.
- Elements of this type are identifications, also called paths or proofs of equality.
- **The fundamental insight:** This type is defined *inductively*.

The Constructor of Identification

If identification is an inductive family, what is its constructor?

The Constructor of Identification

If identification is an inductive family, what is its constructor?

There is one primitive constructor: reflexivity.

```
inductive Eq {A : Type} (a : A) : A -> Type where
| refl : Eq a a
```

The Constructor of Identification

If identification is an inductive family, what is its constructor?

There is one primitive constructor: reflexivity.

```
inductive Eq {A : Type} (a : A) : A -> Type where
| refl : Eq a a
```

Notice the type signature carefully:

- $a : A$ is a fixed parameter.
- The second argument (the endpoint) varies.
- The only constructor gives $\mathrm{refl}_a : a =_A a$.

Optional Detour: Identification Elimination

The next few frames are optional for a first programming pass. I will move through them quickly because they show the power of an eliminator beyond ordinary data such as $\mathbb{N}$ or `List`.
For $\mathbb{N}$, the inductor feels natural: prove zero and successor. For identification types, the eliminator can feel like a strange axiom at first: to reason about all identifications from a, it is enough to reason about refl_a.

Formal Definition: Identification Elimination

Because $x =_A y$ is inductive, it has an **eliminator**. Rijke writes this eliminator as ind-eq; it is also commonly called path induction.

Formal Definition: Identification Elimination

Because $x =_A y$ is inductive, it has an **eliminator**. Rijke writes this eliminator as ind-eq; it is also commonly called path induction.

Definition (Identification Elimination)

Given a fixed element $a : A$, and a property $C(x, p)$ that depends on both:

1. An endpoint $x : A$
2. A path $p : a =_A x$

To prove $C(x, p)$ for **all** x and **all** paths p, it is sufficient to prove $C(a, \text{refl}_a)$.

The Eliminator ind-eq

Formally, the path induction principle is given by the identification eliminator ind-eq. Given a fixed type A and element $a : A$, let $C(x, p)$ be a type family dependent on an endpoint $x : A$ and a path $p : a =_A x$.

Identification Elimination

If we have a proof c for the reflexivity case, ind-eq generates a proof for all paths:

$$\text{ind-eq} : \prod_{C:\left(\prod_{x:A} a =_A x \to \mathcal{U}\right)} C(a, \text{refl}_a) \to \prod_{x:A} \prod_{p:a=_A x} C(x, p)$$

The Eliminator ind-eq

Formally, the path induction principle is given by the identification eliminator ind-eq. Given a fixed type A and element $a : A$, let $C(x, p)$ be a type family dependent on an endpoint $x : A$ and a path $p : a =_A x$.

Identification Elimination

If we have a proof c for the reflexivity case, ind-eq generates a proof for all paths:

$$\text{ind-eq} : \prod_{C:\left(\prod_{x:A} a =_A x \to \mathcal{U}\right)} C(a, \text{refl}_a) \to \prod_{x:A} \prod_{p:a=_A x} C(x, p)$$

By definition, applying this eliminator to the trivial reflexivity path simply returns our base case proof c. This gives us the **computation rule**:

$$\text{ind-eq}(C, c, a, \text{refl}_a) \equiv c$$

The Power of Path Induction: Transport

How do we prove **Leibniz's Law** (the indiscernibility of identicals)? If $x = y$, any property P of x must also hold for y.

The Power of Path Induction: Transport

How do we prove **Leibniz's Law** (the indiscernibility of identicals)? If $x = y$, any property P of x must also hold for y.

In HoTT, this is called **Transport**. We need a function that "transports" a proof across a path:

$$\text{transport} : \prod_{P:A\to\text{Type}} \prod_{x,y:A} (x =_A y) \to P(x) \to P(y)$$

The Power of Path Induction: Transport

How do we prove **Leibniz's Law** (the indiscernibility of identicals)? If $x = y$, any property P of x must also hold for y.

In HoTT, this is called **Transport**. We need a function that "transports" a proof across a path:

$$\text{transport} : \prod_{P:A\to\text{Type}} \prod_{x,y:A} (x =_A y) \to P(x) \to P(y)$$

Proof via Path Induction

By path induction, we fix x. We assume y is x, and the path is refl_x. The required return type definitionally reduces from $P(x) \to P(y)$ to $P(x) \to P(x)$. We simply return the **identity function**.

Transport in Lean 4

We can write this monumental logical theorem as a trivial three-line functional program.

```
def transport {A : Type} (P : A -> Type) {x y : A}
  (p : x = y) (px : P x) : P y :=
  match y, p with
  | _, Eq.refl => px
```

Transport in Lean 4

We can write this monumental logical theorem as a trivial three-line functional program.

```
def transport {A : Type} (P : A -> Type) {x y : A}
  (p : x = y) (px : P x) : P y :=
  match y, p with
  | _, Eq.refl => px
```

- **The Magic:** The compiler sees `Eq.refl`, automatically unifies y with x, and accepts `px` (which has type $P(x)$) as a valid inhabitant of $P(y)$.
- **Takeaway:** Substitution is not a separate logical axiom; it is just an algorithmic consequence of the equality eliminator.

Congruence: Functions Preserve Equality

Another fundamental property is **Congruence**: applying a function f to equal inputs yields equal outputs.

In topology, this means f is continuous—it maps paths to paths. We want to construct:

$$\mathsf{ap}_f : (x =_A y) \to (f(x) =_B f(y))$$

Congruence: Functions Preserve Equality

Another fundamental property is **Congruence**: applying a function f to equal inputs yields equal outputs.

In topology, this means f is continuous—it maps paths to paths. We want to construct:

$$\mathsf{ap}_f : (x =_A y) \to (f(x) =_B f(y))$$

Proof.

We do path induction on the path $p : x =_A y$.

- Assume y is x, and p is refl_x.
- The goal $(f(x) =_B f(y))$ computationally reduces to $(f(x) =_B f(x))$.
- We simply return $\text{refl}_{f(x)}$.

Congruence in Lean 4 (congrArg)

In Lean's standard library, this is known as congrArg. Again, the proof is mathematically trivialized by pattern matching.

```
def ap {A B : Type} (f : A -> B) {x y : A}
(p : x = y) : f x = f y :=
match y, p with
| _, Eq.refl => Eq.refl
```

Congruence in Lean 4 (congrArg)

In Lean's standard library, this is known as congrArg. Again, the proof is mathematically trivialized by pattern matching.

```
def ap {A B : Type} (f : A -> B) {x y : A}
(p : x = y) : f x = f y :=
match y, p with
| _, Eq.refl => Eq.refl
```

- By matching on Eq.refl, the type checker knows x and y are identical.
- Therefore, $f(x)$ and $f(y)$ are identical, allowing us to return Eq.refl for the output type.
- **Conclusion:** Functional programming translates profound topological and logical rules into basic constructor matching.

Why is Path Induction Strange?

To classically trained mathematicians, this can feel like an unexpected logical step.

Why is Path Induction Strange?

To classically trained mathematicians, this can feel like an unexpected logical step.

The Counter-Intuitive Leap

We are proving a theorem for *any* x and *any* path connecting a to x. But we only do the work for the trivial case where x is literally a and the path is the trivial loop refl_a.

Why is Path Induction Strange?

To classically trained mathematicians, this can feel like an unexpected logical step.

The Counter-Intuitive Leap

We are proving a theorem for *any* x and *any* path connecting a to x. But we only do the work for the trivial case where x is literally a and the path is the trivial loop refl_a.

- **The "Free Endpoint" Trap:** It *only* works if the second endpoint x is allowed to vary. You cannot do path induction on a closed loop $p : a = a$.
- **Topology in Logic:** It treats paths not as rigid equations, but as topological deformations. Because one endpoint is free, any path can be continuously contracted back to a point (refl).

Why is it Powerful?

- Path induction trivially solves what would otherwise require deep structural axioms. For example, how do we prove that equality is **symmetric** (if $a = x$, then $x = a$)?
- Let $C(x, p)$ be the type $x = a$. We need a function from $a = x \to x = a$.
- By path induction, we assume x is a and p is refl_a.
- The goal $x = a$ becomes $a = a$. We just return refl_a.

Lean 4 Proof:

```
def symm {A : Type} {a x : A} (p : a = x)
: x = a
:= match x, p with
| _, Eq.refl => Eq.refl
```

We literally pattern-matched on equality itself. This bridges functional programming and the foundations of mathematics.

Outline of this Section

The Lambda Calculus Bridge

Functional Programming is, at its mathematical core, **applied Lambda Calculus**.

The Lambda Calculus Bridge

Functional Programming is, at its mathematical core, **applied Lambda Calculus**.

When we write programs in Lean 4, we implicitly inherit its foundational reduction rules. The compiler automatically handles these equivalences:

- α-**Conversion (Renaming):** Bound variable names are arbitrary.
- β-**Reduction (Substitution):** Application is computation.
- η-**Conversion (Extensionality):** Functions are defined by their behavior.

α-Conversion and β-Reduction

α-Conversion (Variable Renaming)

The names of *bound variables* do not change the essence of the computation.

$$\lambda x.M[x] \equiv \lambda y.M[y]$$

Lean: `fun x => x` is structurally identical to `fun y => y`.

α-Conversion and β-Reduction

α-Conversion (Variable Renaming)

The names of *bound variables* do not change the essence of the computation.

$$\lambda x.M[x] \equiv \lambda y.M[y]$$

Lean: `fun x => x` is structurally identical to `fun y => y`.

β-Reduction (The Engine of Computation)

Applying a function directly substitutes the argument into the bound variable.

$$(\lambda x.M)N \longrightarrow M[x := N]$$

Lean: `(fun x => x^2 + 1) 5` mechanically reduces to `5^2 + 1`.

η-Reduction: Extensionality and Point-Free Style

If two functions return the same result for all inputs, they are the same function.

η-Reduction: Extensionality and Point-Free Style

If two functions return the same result for all inputs, they are the same function.

η-Reduction

If x does not appear free in f, we can abstract the tail application:

$$\lambda x.f(x) \equiv f$$

η-Reduction: Extensionality and Point-Free Style

If two functions return the same result for all inputs, they are the same function.

η-Reduction

If x does not appear free in f, we can abstract the tail application:

$$\lambda x.f(x) \equiv f$$

In Functional Programming, this justifies **Point-Free Style** (tacit programming)—dropping explicit parameters for cleaner code:

Currying as the Default Convention

In classical mathematics, functions of multiple variables take a tuple: $f(x, y)$. In Functional Programming, we default to **Currying** (named after logician Haskell Curry).

The Currying Isomorphism

In a Cartesian closed category, there is a natural bijection between functions taking tuples and functions returning functions:

$$(A \times B \to C) \cong (A \to (B \to C))$$

Why this matters in Lean 4:

- Under the hood, **every function takes exactly one argument**.

Partial Application in Action

Because of Currying, supplying fewer arguments than a function requires does not throw an error—it yields **Partial Application**, returning a specialized function awaiting the remaining arguments.

```
def add (x : Nat) (y : Nat) : Nat := x + y

-- 'add' has type: Nat -> (Nat -> Nat)
-- We can partially apply the first argument!
def addTwo : Nat -> Nat := add 2

#eval addTwo 5 -- Returns 7
```

Partial Application in Action

Because of Currying, supplying fewer arguments than a function requires does not throw an error—it yields **Partial Application**, returning a specialized function awaiting the remaining arguments.

```
def add (x : Nat) (y : Nat) : Nat := x + y

-- 'add' has type: Nat -> (Nat -> Nat)
-- We can partially apply the first argument!
def addTwo : Nat -> Nat := add 2

#eval addTwo 5 -- Returns 7
```

This inherited rule is precisely why Higher-Order Functions (like mapping or filtering over a list) are so seamless and powerful to write in Lean.

Outline of this Section

List: intro

- In this section, we will learn the fundamental inductive structure, which is very similar with Peano natural number.
- I suggest you to do all the examples and exercises by your hand; all the high technicals are abstracted from such basic things.

List: definition

A list is defined as:

```
inductive List (a : Type u) where
| nil : List a
| cons (head : a) (tail : List a) : List a
```

recall that:

```
inductive Nat: Type where
| zero : Nat
| succ : Nat -> Nat
```

So, it's really similar to write some basic functions over list.

List: constructors

- Firstly, let try to define some new lists:

```
def lst1 : List Nat := []
def lst2 : List Nat := [ 1 ]
def lst3 : List Nat := List.cons 2 lst2
```

- I know you my be eager to write some real codes; but before we start, we need some necessary IO to show your result:

```
def main : IO ()
:= do
  ...
  return ()
```

List Function 0: write some Nat function

Exercise

*Write the following functions on `Nat` using only pattern matching — no + or *:*

First write down the type signatures and constructor cases. During the break, complete at least one definition fully and test it with `#eval`.

```
-- add 2 to any Nat (use .succ twice)
def add2 : Nat -> Nat := by admit
-- add two Nats (recursion on first argument)
def add : Nat -> Nat -> Nat := by admit
-- optional: multiply two Nats (use add in the step)
def mul : Nat -> Nat -> Nat := by admit
```

Live Lean: `lean-code/D5_01_ListFunctions_Exercises.lean`

Solution to the Exercise: add2

A direct constructor-by-constructor solution is:

```
def add2 : Nat -> Nat
  | .zero => .succ .succ .zero
  | .succ k => .succ .succ .succ k

-- equivalently: def add2 := .succ .succ
```

Notice the abbreviated constructor notation: `.zero` instead of `Nat.zero`.

Live Lean: `lean-code/D5_01_ListFunctions_Solutions.lean`

Solution to the Exercise: add

Recursive addition follows the first argument until it reaches `.zero`:

```
def add : Nat -> Nat -> Nat
  | .zero, n => n
  | .succ k, n => .succ (add k n)
```

Live Lean: `lean-code/D5_01_ListFunctions_Solutions.lean`

Solution: Multiple Pattern Matching

You may use simultaneous pattern matching on both arguments:

```
def add : Nat -> Nat -> Nat
  := fun m n =>
      match m, n with          -- note the comma.
      | .zero, .zero   => .zero
      | .zero, .succ l => .succ l
      | .succ k, .zero => .succ k
      | .succ k, .succ l => .succ (.succ (add k l))
-- multiplication: m * (n+1) = m + m*n
def mul : Nat -> Nat -> Nat
  | _, .zero   => .zero
  | m, .succ k => add m (mul m k)
```

Basic functions about const. and elim.

Here are some basic tools to build and build a List, one can try to write them by yourselves:

- range (Nat required);
- dup;
- concat;
- zip (Tuple required);

Higher Order Function

In this subsection, we motivate **higher-order functions**: functions that accept other functions as parameters, or return functions as results.
In ordinary programming this often appears as callbacks or closures. In type theory, it is simpler to say: a function type can itself be an input type.

List Function 1: basic tools

Let's write some list function. Try to write the following functions on List a:

- head;
- length;
- take n (could return empty list if n $>$ length);
- tail.

Live Lean: `lean-code/D5_01_ListFunctions_Exercises.lean`

Solution to List Function 1: Head and Length

```
-- head returns Option: none when the list is empty
def head : List a -> Option a
  | []     => none
  | x :: _ => some x
-- length: count constructors
def length : List a -> Nat
  | []      => 0
  | _ :: xs => length xs + 1
```

Live Lean: `lean-code/D5_01_ListFunctions_Solutions.lean`

Solution to List Function 1: Take and Tail

```
-- take: stop at n = 0 or empty list
def take : Nat -> List a -> List a
  | 0, _           => []
  | _, []          => []
  | n + 1, x :: xs => x :: take n xs
-- tail: drop the head, return the rest
def tail : List a -> List a
  | []      => []
  | _ :: xs => xs
```

Live Lean: `lean-code/D5_01_ListFunctions_Solutions.lean`

List Function 2: apply some function

Assume in the following context, we only discuss the List over Nat. Try to write: add one; multiple one; apply g.

```
def add1 : List Nat -> List Nat
| []      => []
| x :: xs => (x + 1) :: (add1 xs)

def mul2 : List Nat -> List Nat
| []      => []
| x :: xs => (x * 2) :: (mul2 xs)

def applyF (f : Nat -> Nat) : List Nat -> List Nat
| []      => []
| x :: xs => f x :: (applyF f xs)
```

All three follow the same recursive skeleton — `applyF` is the general pattern.

List Function 2: apply

This case can be abstracted as:

```
def map (f : a -> b) : List a -> List b
| []      => []
| x :: xs => f x :: map f xs
```

All the functions above are just use apply function over our list, we can define those function through the “map”.

List Function 3: acc

In this section, we try to handle two lists by induction.

- Reverse a list;
- Concact two lists;

These examples use a result container, similar in spirit to a local variable in sequential programming. In functional code, we call this container the **accumulator**.

List Function 3: acc

We firstly check the solution:

```
def rev'
: List a -> List a -> List a
:= fun acc lst => match lst with
  | [] => acc
  | x :: xs => rev' (x :: acc) xs

def rev : List a -> List a
:= rev' []
```

The first argument is an accumulator: it stores the partial result while the recursive call walks through the remaining list.

List Function 3: acc

Here we use `let rec` to define a local recursive function.

```
def rev : List a -> List a
:= let rec loop : List a -> List a -> List a
    :=  fun acc lst => match lst with
    | [] => acc
    | x :: xs => loop (x :: acc) xs
  fun lst =>
  loop [] lst
```

Inside a function body, `let rec` is the keyword that marks the local helper as self-calling.

List Function 4: fold

Before we introduce the abstracted version of the reverse, we need to first see more easy case; let's write something like:

- Sum of a List;
- Prod of a List;

List Function 4: Fold Pattern for Sum

The accumulator pattern for summing a list is:

```
-- pattern clauses for the accumulator
def sumlst : List Nat -> Nat
:= let rec loop : List Nat -> Nat -> Nat
  | [], acc => acc
  | x :: xs, acc => loop xs (acc + x)
  fun lst => loop lst 0
#eval sumlst [ 1, 2, 3, 4 ] -- 10
```

List Function 4: Fold Pattern for Product

The product version changes only the operation and the initial value:

```
def prodlst : List Nat -> Nat
:= let rec loop : List Nat -> Nat -> Nat
  | [], acc => acc
  | x :: xs, acc => loop xs (acc * x)
  fun lst => loop lst 1
#eval prodlst [ 1, 2, 3, 4 ] -- 24
```

In both cases, the inner function is just a locally bound recursive definition.

List Function 4: foldr

We can abstract the pattern above:

```
def foldr (f : a -> b -> b) (init : b) : List a -> b
  | []      => init
  | x :: xs => f x (foldr f init xs)
```

- `foldr` processes from right to left, combining with `f`.
- `foldl` processes from left to right; this is the pattern in our `rev` function.

```
def sumlst'  := List.foldr Nat.add 0
def prodlst' := List.foldr Nat.mul 1
#eval sumlst' [1, 2, 3, 4]   -- 10
```

List Function 5: filter

Sometimes, we want to collect some elements on a list, it's the filter pattern:

- GE1, LE10;
- filter.

The pattern is the "filter":

```
def filter : (a -> Bool) -> List a -> List a
:= fun p lst => match lst with
  | [] => []
  | x :: xs =>
    if p x
    then x :: (filter p xs)
    else filter p xs
```

Live Lean: `lean-code/D5_01_ListFunctions.lean`

Essential List Functions in Lean 4

In Lean 4, all list utilities live in the `List` namespace. Functions that can fail return `Option` rather than panicking on empty input.

Function (Lean 4)	**Description**
`List.head? xs`	`Option` α — first element, `none` if empty
`List.getLast? xs`	`Option` α — last element, `none` if empty
`xs.length`	`Nat` — number of elements ($O(n)$)
`List.take n xs`	First n elements
`List.drop n xs`	All elements after the first n
`List.reverse xs`	Reversed list
`List.isEmpty xs`	`Bool` — true iff `[]`
`List.contains xs x`	Membership check: `Bool`

Essential List Functions in Lean 4 (Transform)

Function (Lean 4)	**Description**
`List.map f xs`	Apply `f` to every element
`List.filter p xs`	Keep elements where `p x = true`
`List.foldl f init xs`	Left fold: `f (f init` x_1`)` x_2 …
`List.foldr f init xs`	Right fold: `f` x_1 `(f` x_2 …`init)`
`List.flatMap xs f`	Map then flatten (`f :` α `→ List` β)
`List.zip xs ys`	Pair up: `List (`$\alpha \times \beta$`)`
`List.zipWith f xs ys`	Combine element-wise with `f`
`List.join xss`	Flatten one level: `List (List` α`) → List` α

Essential List Functions in Lean 4 (Search & Sort)

Function (Lean 4)	**Description**
`List.find? p xs`	First element where `p x`: `Option` α
`List.findIdx? p xs`	Index of first match: `Option Nat`
`List.any p xs`	Any element satisfies `p`: `Bool`
`List.all p xs`	All elements satisfy `p`: `Bool`
`List.partition p xs`	Split into (passing, failing) pair
`List.sort xs`	Sort ascending (requires `Ord` α)
`List.sortBy cmp xs`	Sort with custom comparator
`List.dedup xs`	Remove consecutive duplicate elements

Note: Linked Lists and Performance

- The inductive `List` we have been using is a **singly-linked list**. Head access is $O(1)$, but random access and appending to the end are $O(n)$.
- In verified algorithm development, we first prove **correctness** using these simple inductive structures, then optimize performance separately.
- Lean 4's standard library also provides `Array` (backed by a contiguous heap array) for $O(1)$ random access when performance matters.

Optional Challenge: Merge Sort in Lean 4: Helpers

This is a stretch task, not part of the easy exercise file. If the basic list exercises are finished, try the helper functions first.

```
def splitHalf (xs : List Nat) : List Nat × List Nat :=
  (List.take (xs.length / 2) xs,
   List.drop (xs.length / 2) xs)

-- Merge two sorted lists into one sorted list
def merge : List Nat -> List Nat -> List Nat
  | [], ys => ys
  | xs, [] => xs
  | x :: xs, y :: ys =>
      if x <= y then x :: merge xs (y :: ys)
      else y :: merge (x :: xs) ys
```

Optional Challenge: Merge Sort in Lean 4: Recursion

After the helper functions are clear, write the recursive driver. The remaining Lean task is to justify that the recursive calls use shorter lists.

```
-- Merge sort (add termination_by xs.length for Lean)
def mergeSort (xs : List Nat) : List Nat :=
  if xs.length <= 1 then xs
  else let (l, r) := splitHalf xs
       merge (mergeSort l) (mergeSort r)
termination_by xs.length
```

Outline of this Section

1. Why Functions Must Terminate in Lean 4

Lean 4 is a **total functional language** rooted in dependent type theory. Every function must be proven to terminate in finite steps.

Logical Soundness (Curry-Howard Isomorphism)

In Lean's logical fragment, propositions are types and proofs are programs. If arbitrary infinite recursion were accepted as a definition, a nonterminating term could be assigned a proof type such as `False`, destroying soundness.

```
    -- If infinite loops were permitted:
def proveFalse : False := proveFalse -- Logical collapse.
```

2. Structural vs. Well-Founded Recursion

Lean verifies termination using two distinct mechanisms:

Structural Recursion (Automatic)

- Peels off exactly one constructor per call (`x :: xs` → `xs`).
- Lean automatically inspects the inductive type definition.
- No annotations needed.

Well-Founded Recursion

- Non-linear reductions (e.g., division, arithmetic, splitting lists).
- Automatic inspection cannot see that arguments shrink.
- Requires annotation: `termination_by`.

3. Syntax Pattern A: Direct Parameter Reference

When your function explicitly names its parameters in the header, write the decreasing expression directly after `termination_by`.

```
    /-- Euclidean algorithm for Greatest Common Divisor -/
def gcd (a b : Nat) : Nat :=
  if b == 0 then a
  else gcd b (a % b)
termination_by b
```

Why this is legal: The expression `b` has type `Nat`. Because `a % b < b` for any $b > 0$, Lean verifies that the natural number strictly decreases toward 0.

4. Syntax Pattern B: Explicit Variable Binding

When pattern matching anonymously across multiple lines, you must bind the parameter names before the arrow (=>) so the measure expression can reference them.

```
def merge : List Int -> List Int -> List Int
  | [], ys => ys
  | xs, [] => xs
  | x::xs, y::ys =>
    if x <= y then x :: merge xs (y::ys)
    else y :: merge (x::xs) ys
termination_by xs ys => xs.length + ys.length
```

Measure: The sum `xs.length + ys.length` strictly drops by exactly 1 on every recursive branch.

5. Advanced: Lexicographical Tuples

For complex multi-variable recursion where one variable might increase while another decreases, use a **tuple expression** `(x, y)`.

```
/-- Ackermann function requires lexicographical order -/
def ack : Nat -> Nat -> Nat
  | 0,   y   => y + 1
  | x+1, 0   => ack x 1
  | x+1, y+1 => ack x (ack (x + 1) y)
termination_by x y => (x, y)
```

Well-Founded Relation: Lean compares tuples from left to right: either x strictly shrinks, or x stays identical and y strictly shrinks.

6. Manual Proofs with `decreasing_by`

If Lean's automatic tactics cannot prove your measure decreases, attach a `decreasing_by` block immediately following `termination_by`.

```
def div2 (n : Nat) : Nat :=
    if h : n < 2
        then 0
        else 1 + div2 (n - 2)
    termination_by n
    decreasing_by
        -- Provide explicit proof that n - 2 < n
        omega
```

The omega Tactic

The tactic omega solves many arithmetic goals over natural numbers and integers, especially goals built from linear inequalities.

- In a termination proof, Lean often reduces the problem to an arithmetic inequality such as $n - 2 < n$.
- omega is useful when the recursive structure is clear to us, but Lean needs help with the generated arithmetic side condition.
- It is not a general theorem prover: it is specialized for Presburger-style linear arithmetic.

7. Summary of Legal Measure Types

Measure Type	Example Syntax	Typical Use Case
`Nat`	`termination_by n`	Countdown loops, arithmetic steps
List Length	`termination_by xs.length`	Custom list splitting / slicing
Combined Sum	`... => xs.length + ys.length`	Interleaved multi-list processing
Tuple (Lex)	`... x y => (x, y)`	Nested recursive calls (Ackermann)
Custom Proof	`decreasing_by exact ...`	Non-standard arithmetic bounds

Interesting Project: Sudoku in Lean 4

Sudoku is a classic exercise for functional programmers:

- **Grid as data**: represent a 9×9 grid as `Array (Array (Option Nat))`
- **Constraint checking**: write `validRow`, `validCol`, `validBox` using `List.nodup`
- **Backtracking solver**: recursive search where `Option` records failure explicitly
- **Proof challenge**: prove that your checker is complete (never rejects a valid board)

Not required, but highly recommended if you want hands-on FP practice beyond the exercises.

Break 1: List and Recursion Checkpoint

This is the end of the first 50-minute block. The goal is not to memorize every list function; the goal is to see the recurring shape.

- Take about 5 minutes for a short break.
- Then spend about 5 minutes in `lean-code/D5_01_ListFunctions.lean`.
- Choose one exercise close to the examples: rewrite `myMap`, `myFilter`, or `myFoldr` on a fresh list.

After this, we start the second block with inductive trees and graphs, then move to monads.

Outline of this Section

Tree definition

We can define tree as:

```
inductive Tree := Nil | Node Tree Tree
```

But it carry no information

Live Lean: `lean-code/D5_02_Structures.lean`

More Trees

We modify our tree to:

```
inductive Tree a :=
  Nil
| Node (Tree a) a (Tree a)
```

Live Lean: `lean-code/D5_02_Structures.lean`

Exercises

The easy in-class tree exercises are:

- `size`: count nodes.
- `depth`: compute the height of the tree.
- `reflect`: swap left and right recursively.
- `search`: use `BEq` to find one value.

Harder functions such as `isBST`, `balanced`, and insertion are left as optional after-class extensions.

Live Lean: `lean-code/D5_02_Structures_Exercises.lean`

Optional Problem

Write a binary balanced tree.

- Simple implementation: build from a sorted list.
- Regular implementation: maintain balance while inserting.

Inductive Graph: Three Representations

Graphs can also be defined inductively. Three common approaches:

- **Algebraic Graph** (Mokhov): build graphs via constructors

```
inductive Graph (a : Type) where
  | empty   : Graph a
  | vertex  : a -> Graph a
  | overlay : Graph a -> Graph a -> Graph a  -- union
  | connect : Graph a -> Graph a -> Graph a  -- add all edges
```

- **Edge list**: `List (a × a)` — pairs of connected vertices
- **Adjacency list**: `List (a × List a)` — vertex → neighbour list

Live Lean: `lean-code/D5_02_Structures.lean`

Exercises: Graph Functions

The easy in-class graph exercises are:

- `vertexCount : Graph a -> Nat` — count constructor occurrences of vertices.
- `directEdges : Graph a -> List (a × a)` — collect the easiest visible edges from `connect`.

Distinct-vertex counting, edge membership, and graph proofs are optional after-class extensions.

Live Lean: `lean-code/D5_02_Structures_Exercises.lean`

Outline of Sections

Outline of this Section

Monad intro 1: why do we need monads?

- In the previous section, most functions returned plain values such as `Nat` or `List Nat`.
- Real programs often return a value together with a context: possible failure, an error message, shared configuration, state, or interaction with the outside world.
- A monad is the standard functional-programming pattern for sequencing computations that all live in the same context.

We will read every example in the same format: main feature, simplest usage, and the reason `do`-notation helps.

Monad intro 1: motivation from possible failure

- In the previous context, we had list functions such as `take`. Some operations, such as `head`, genuinely may fail on an empty list.
- The first idea is to make failure explicit in the return type:

```
inductive Optional (a : Type) : Type where
| none : Optional a
| some : a -> Optional a
```

- Lean's standard type is `Option a`. It is better than using a special dummy value, because the type tells us that failure is possible.
- If we also want to explain the failure, we use `Except error a`.

Monad intro 2: failure with information

A common Haskell name for this shape is `Either`; in Lean, the usual programming type is `Except`. The idea is:

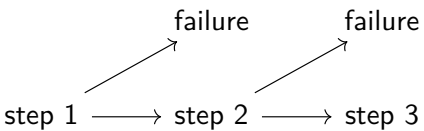
```
inductive Either (a b : Type) : Type where
| left : a -> Either a b
| right : b -> Either a b
```

For a safe list lookup, the success branch carries the value and the failure branch carries a message:

```
def safeIndexE : Nat -> List Nat -> Except String Nat
  | _, [] => .error "index out of bounds"
  | 0, x :: _ => .ok x
  | n + 1, _ :: xs => safeIndexE n xs
```

Monad intro 3: the sequencing pattern

The examples show a great idea:



Once every step returns the same kind of contextual value, we want one uniform way to say: run this step, and if it succeeds, continue with the next step.

Monad intro 3: Monad bind

The procedure we need to define is:

$$\texttt{m a} \xrightarrow{\texttt{bind}} \texttt{a -> m b} \longrightarrow \texttt{m b}$$

The operation is called `bind`. It does not give us a general way to escape from `m a` to `a`; instead, it lets the next computation use the successful value while staying inside the same context.

The Inventors of Monads

Eugenio Moggi (1960–)

- *"Notions of Computation and Monads"* (1991)
- Monads as semantics for computational effects

Philip Wadler (1956–)

- *"Comprehending Monads"* (1992)
- Brought monads into practical FP and Haskell

Small Tips

One can try some *ligature fonts*, which has a great visual appearance on Monadic symbols.

Monad intro 4: formal definition

Formally, a **Monad** on a type universe is a type constructor m : Type u -> Type v equipped with two fundamental operations:

```
class Monad (m : Type u -> Type v) where
  pure : a -> m a
  bind : m a -> (a -> m b) -> m b
```

- **pure** (or *return*): Embeds a clean value into the monadic effect container.
- **bind** (»=): Chains a computation that produces an effect onto an existing effectful value.

Monad intro 5: syntactic sugar (do-notation)

Writing nested bind functions manually becomes unreadable. Lean provides **do-notation** as syntactic sugar for sequential pipelines:

```
-- Raw Bind:
def calc : Option Nat :=
  findX >>= fun x =>
  findY >>= fun y =>
  pure (x + y)
-- do notation:
def calc : Option Nat := do
  let x <- findX
  let y <- findY
  return (x + y)
```

Under the hood, Lean desugars the <- arrow directly into bind.

Live Lean: `lean-code/D5_03_Monad.lean`

Exercise: Another Small `Option` Program

Before moving to `IO`, try a nearby exercise in the same style:

```
def safePred : Nat -> Option Nat := by admit

def predThenHalf : Nat -> Option Nat := by admit
```

Read this exercise as follows: first write a predecessor that can fail at zero, then reuse the old `safeDiv` pattern to continue the computation. The point is not a new theorem; it is simply more practice reading `Option` as a computation with possible failure.

Live Lean: `lean-code/D5_03_Monad_Exercises.lean`

Except: failure with a message

Main feature: `Except String a` returns either a value of type `a` or an explanatory error string.

Simplest usage: chain lookups that may fail, and let `do`-notation stop at the first error.

```
def addThirdAndFourth (xs : List Nat)
    : Except String Nat := do
  let a <- safeIndexE 2 xs
  let b <- safeIndexE 3 xs
  pure (a + b)
```

Compared with `Option`, `Except` is better for teaching and debugging because the failed branch explains what went wrong.

Live Lean: `lean-code/D5_03_Monad.lean`

IO: interaction with the real world

Main feature: `IO a` describes a computation that may interact with the runtime system and eventually produce an `a`.
Simplest usage: print a line or read a line.

- Pure definitions remain mathematical and referentially transparent.
- Effects are pushed to the program boundary and marked in the type.

```
def greet : IO Unit := do
  IO.println "Functional programming keeps effects explicit."
```

Live Lean: `lean-code/D5_03_Monad.lean`

IO: controlled mutability with `IO.Ref`

In functional programming, ordinary variables are immutable. When a program really needs a mutable cell, Lean keeps that operation inside `IO`.

- An `IO.Ref a` is a reference to a mutable cell holding a value of type `a`.
- Reading and mutating it are explicitly tracked as `IO` effects.
- The rest of the development can still use pure values and pure functions.

IO: a small mutable counter

Simplest usage: allocate a reference, modify it, and read the value back. The key APIs are `IO.mkRef`, `IO.Ref.modify`, and `IO.Ref.get`.

```
def counterDemo : IO Unit := do
  let r <- IO.mkRef 0
  r.modify (fun n => n + 1)
  r.modify (fun n => n + 1)
  let val <- r.get
  IO.println s!"Counter value: {val}"
```

Live Lean: `lean-code/D5_03_Monad.lean`

IO: exception handling at the boundary

Real-world operations fail: files may be missing or permissions may be wrong. Lean handles these failures inside `IO` with structured `try`/`catch`.

```
def readConfigSafely (path : String) : IO String := do
  try
    -- IO.FS handles raw filesystem streams
    let content <- IO.FS.readFile path
    return content
  catch e =>
    -- e has type IO.Error
    IO.eprintln s!"[Warning] Could not load {path}: {e}"
    return "DEFAULT_CONFIG_DATA"
```

IO: advanced runtime features

Lean's `IO` layer is also where systems features live.

- `Task`: run computations concurrently when the runtime should manage background work.
- **Foreign-function interface**: connect Lean code to external compiled code.
- For this lecture: keep the core algorithm pure; use `IO` only at the boundary.

The Abstraction Hierarchy: Functor & Applicative

In Lean 4, a **Monad** is not an isolated concept—it is the top of a strict typeclass hierarchy: `Functor` → `Applicative` → `Monad`.

```
class Functor (f : Type u -> Type v) where
  -- <$>
  map : {a b : Type u} -> (a -> b) -> f a -> f b
class Applicative (f : Type u -> Type v)
       extends Functor f where
  pure : {a : Type u} -> a -> f a
  -- <*>
  seq  : {a b : Type u} -> f (a -> b) -> (Unit -> f a) -> f b
```

- `Functor` (`<$>`): Applies a pure function inside an effect container.
- `Applicative` (`<*>`): Applies an *effectful function* to an effectful argument (allowing independent, static computation trees).

Why Applicative before Monad?

Why not jump straight to `Monad`? Because `Applicative` handles **independent effects** cleanly without requiring full sequential evaluation chains.

```
-- Combines independent Option computations
def addOpt_app (x y : Option Nat)
  : Option Nat :=
  pure Nat.add <*> x <*> y

-- Sequential style: later lines may depend on earlier values
def addOpt_do (x y : Option Nat)
  : Option Nat := do
  let a <- x
  let b <- y
  return (a + b)
```

Difference: monadic lines may depend on earlier values; applicative shape is fixed in advance.

Outline of this Section

How to Read a Monad Example

For each monad, ask three questions:

- **What is the type shape?** For example, `Option a`, `Except String a`, or `StateM s a`.
- **What context does it represent?** Failure, an error message, a read-only environment, state, logging, or runtime interaction.
- **What does <- mean here?** It runs one contextual step and binds the successful result for the next line.

There is usually a `pure : a -> m a`, but there is not generally an "escape" function `m a -> a`. We keep sequencing inside the context.

Common Monad Examples

Monad	**Main feature**	**Simplest usage**
`Option`	failure without a message	stop when a lookup returns `none`
`Except e`	failure with a reason	return `.error "..."` from a checker
`ReaderM r`	shared read-only environment	read configuration without passing it manually
`StateM s`	pure state threading	update a counter and return the new value
Writer-style logs	accumulated output	use `StateM (Array String)` with `modify`
`IO`	runtime interaction	print, read files, or allocate mutable references

The Reader Monad (`ReaderM`)

Main feature: `ReaderM r a` is a computation that can read a shared environment of type `r`.

Simplest usage: package a function from configuration to result.

```
structure AppConfig where
  dbHost : String
  port   : Nat

def buildConnectionUrl : ReaderM AppConfig String := do
  let cfg <- read
  pure s!"http://{cfg.dbHost}:{cfg.port}/api"
```

We choose `ReaderM` when many functions need the same read-only input and manual parameter passing would obscure the main algorithm.

Live Lean: `lean-code/D5_03_Monad.lean`

Exercise: ReaderM and an Environment

This is the easiest Reader-style exercise: use Lean's standard `read` API, then format one string.

```
structure AppConfig where
  dbHost : String
  port   : Nat

def exBuildConnectionUrl : ReaderM AppConfig String := do
  let cfg <- read
  admit
```

The goal is to feel that `ReaderM` is just a function with a shared input. Try to write the one-line body first, then test it with a sample configuration.

Live Lean: `lean-code/D5_03_Monad_Exercises.lean`

The State Monad (`StateM`)

Main feature: `StateM s a` represents a pure state transition. Conceptually, it has the shape

$$s \to a \times s.$$

It does not mutate a variable in place; it returns a value together with the next state.

```
#check StateM
#check get
#check set
#check modify
```

The State Monad (`StateM`)

Simplest usage: update a counter and return its current value.

```
def incrementAndGet : StateM Nat Nat := do
  modify (fun s => s + 1)
  get
```

Live Lean: `lean-code/D5_03_Monad.lean`

Exercise: One State Step More

The next exercise should feel very close to the previous frame:

```
def tickTwice : StateM Nat Nat := by admit
```

Your task is to increase the counter twice and return the final value. If the previous example made sense, this one is only one line more complicated. That is exactly the right difficulty for the in-class writing block.

Live Lean: `lean-code/D5_03_Monad_Exercises.lean`

Writer-Style Logging with `StateM`

Main feature: a writer-style computation accumulates secondary output, such as logs or diagnostics, beside the main result.
Lean 4 core provides standard `StateM` and `modify`. In this toolchain there is no separate standard `WriterT`, so we model the writer pattern by storing an append-only array of log messages.

```
def logOneMessage : StateM (Array String) Unit := do
  modify (fun logs => logs.push "started")
```

Exercise: Writer-Style Logging

Write one short logging computation in the same style:

```
def exDivideWithLog (a b : Nat)
    : StateM (Array String) Nat := do
  modify (fun logs => logs.push s!"Dividing {a} by {b}")
  admit
```

Start with a one-message version, then add the zero-division branch only if you still have time before the break.

Live Lean: `lean-code/D5_03_Monad_Exercises.lean`

Writer-Style Logging with `StateM`

Simplest usage: record a message before returning the main value.

```
def divideWithLog (a b : Nat)
: StateM (Array String) Nat := do
  if b == 0 then
    modify (fun logs => logs.push "division by zero")
    return 0
  else
    modify (fun logs => logs.push s!"Dividing {a} by {b}")
    return a / b
```

Live Lean: `lean-code/D5_03_Monad.lean`

Monad Transformers: stacking effects

What if we want both error handling (`Except`) *and* state (`State`)? We use **monad transformers**, whose names usually end in `T`.

$$\text{Base effect} \xrightarrow{\texttt{StateT SymbolTable}} \texttt{StateT SymbolTable (Except String) a}$$

```
abbrev SymbolTable := List String
abbrev CompilerM := StateT SymbolTable (Except String)
```

The result is one monad with two capabilities: it can update a symbol table and also report a typed error.

Live Lean: `lean-code/D5_03_Monad.lean`

Break 2: Tree, Graph, and Monad Checkpoint

We have now finished the second block: more inductive data structures, then monads as a uniform way to sequence contextual computations.

- Take 5 minutes to rest your eyes.
- Then spend 5 minutes in `lean-code/D5_03_Monad_Exercises.lean`: run the `#eval` examples and complete one nearby exercise.
- After the break, we move from "programs over data" to "programs that implement a language".

Outline of Sections

Block III Roadmap

The last block connects functional programming with language implementation and Lean's meta level.

- First, we build a small expression language: syntax tree, environment, evaluator.
- Then, we outline a Scheme-like interpreter project, following the staged spirit of *Write Yourself a Scheme in 48 Hours*.
- Finally, we look at Lean's own levels: `Syntax`, macros, elaboration, `Expr`, and kernel checking.

BNF

In programming language theory, a grammar describes which strings count as well-formed programs. A Backus–Naur Form (BNF) rule can be read as an inductive definition of syntax.

$$e ::= n \mid x \mid e + e \mid e \times e$$

In a functional language, each production rule usually becomes one constructor of an inductive type. This is the bridge from strings to trees.

Abstract Syntax Trees (AST)

To implement a small language, we first model syntax with an inductive type:

```
inductive Expr where
  | val  : Nat -> Expr
  | var  : String -> Expr
  | add  : Expr -> Expr -> Expr
  | mul  : Expr -> Expr -> Expr
  deriving Repr
```

Values are leaves; the other constructors describe computation rules. The expression $(x + 2) \times 3$ becomes:

```
def myExpr : Expr :=
  Expr.mul (Expr.add (Expr.var "x") (Expr.val 2))
           (Expr.val 3)
```

Reduction and Evaluation

- An expression tree is syntax, not yet a value. For example, `Expr.add (Expr.val 2) (Expr.val 3)` is a tree that still needs to be evaluated.
- A reduction step simplifies an expression according to rules. Evaluation repeatedly applies such rules until a value is reached, or until the evaluator reports failure.
- Different languages choose different evaluation rules. For example, call-by-value and call-by-name treat self-application and fixed-point encodings differently.

Expression Language: A Capstone Example

Implementing a mini programming language ties together everything from this course:

- **Inductive types** — define syntax (AST) as a recursive data structure
- **Pattern matching** — evaluate expressions by case analysis on constructors
- **Option monad** — handle variable lookup failures cleanly with `do`-notation
- **Proof by induction** — verify that the evaluator satisfies algebraic laws

The same pipeline (AST $\rightarrow$ evaluator $\rightarrow$ prover) scales to real language implementations.

Evaluating Expressions: An Environment

To evaluate expressions containing variables (Expr.var), we need an **Environment** (a symbol table) mapping variable names to numerical values.

```
abbrev Env := String -> Option Nat
def emptyEnv : Env := fun _ => none
def extendEnv (k : String) (v : Nat) (env : Env)
: Env :=
  fun name =>
    if name == k
    then some v
    else env name
```

Because lookup might fail for undefined variables, our evaluator must return an Option Nat.

The Evaluator (Pattern Matching & Monads)

We write the evaluator by induction on the Expr structure. do-notation handles lookup failures:

```
def eval (e : Expr) (env : Env)
: Option Nat := match e with
  | Expr.val n => pure n
  | Expr.var s => env s
  | Expr.add l r => do
      let vl <- eval l env
      let vr <- eval r env
      return (vl + vr)
  | Expr.mul l r => do
      let vl <- eval l env
      let vr <- eval r env
      return (vl * vr)
```

Live Lean: `lean-code/D5_04_Expr.lean`

Exercise: Another Function on the Same Syntax Tree

At this point, students should try one function whose difficulty is close to `eval` but conceptually simpler:

```
def depth : Expr -> Nat := by admit
def countVars : Expr -> Nat := by admit
```

The important observation is that both definitions follow the constructors of `Expr` exactly. This is a good moment to let students feel that syntax trees are just inductive data, not mysterious compiler objects.

Live Lean: `lean-code/D5_04_Expr_Exercises.lean`

Lean's Expr

Lean is selfhosted, you can write functional programs in Lean that analyze, generate, and prove properties about Lean expressions directly!

```
-- Expr:
inductive Lean.Expr where
| bvar (deBruijnIndex : Nat)
| fvar (fvarId : FVarId)
| const (declName : Name)
      (us : List Level)
| app (fn : Expr) (arg : Expr)
| lam (binderName : Name)
(ty : Expr) (body : Expr)
```

Syntax and Evaluation: The Pipeline

- **Syntax** (BNF $\leftrightarrow$ inductive type): each production rule maps directly to a constructor.

$$\mathsf{e} ::= n \mid x \mid e_1 + e_2 \mid e_1 \times e_2$$

  corresponds exactly to `val`, `var`, `add`, `mul`.
- **Evaluation** (pattern match on constructors): each constructor has one case in `eval`, making the correspondence between syntax and semantics structurally explicit.
- **Failure propagation**: variable lookup returns `Option Nat`; the monad threads failure automatically through the recursive cases.

Project: Build a Mini Language in Lean 4

The project is to extend the `Expr` language into a small interpreter. The model is the classic Haskell tutorial *Write Yourself a Scheme in 48 Hours*, but our Lean version should start smaller and stay type-directed.
A good minimum target is:

1. arithmetic expressions: natural numbers, addition, multiplication;
2. boolean expressions: comparison and equality;
3. control flow: `if`-then-else;
4. local binding: `let` with an environment;
5. error reporting with `Except String`.

This is already a complete language core: it has syntax, values, environments, and evaluation.

Project Requirements

Component	Required result
Abstract syntax	Define an inductive `RichExpr` for numbers, booleans, variables, arithmetic, comparison, `if`, and `let`.
Values	Define `Val` so the evaluator can return either numeric or boolean results.
Environment	Represent variables with `Env := String -> Option Val`; implement `lookup` and `extend`.
Evaluator	Implement `eval : RichExpr -> Env -> Except String Val`.
Tests	Add several `#eval` examples for successful programs and useful error messages.

The project is complete when a reader can run the examples and understand every failure from its message.

Following the 48-Hour Scheme Framework

The Haskell tutorial grows a Scheme interpreter in stages. We can follow the same architecture, but adapt it to Lean:

- **Parser/front end**: optional for the first version; start directly with an AST if time is short.
- **Evaluation**: required; this is today's `eval` function generalized to richer values.
- **Error handling**: use `Except String`, the Lean analogue of the tutorial's explicit error monad.
- **Variables and environment**: use functions or finite maps to model variable lookup and extension.
- **Functions and closures**: advanced extension after the core evaluator is stable.
- **REPL**: optional; in Lean, a collection of `#eval` tests is enough for the first milestone.

Project Milestones

1. **Milestone 1**: arithmetic and booleans run locally in `D5_ProjectStarter.lean`.
2. **Milestone 2**: errors are readable; wrong-type operations and unbound variables do not crash silently.
3. **Milestone 3**: `let` expressions use an environment correctly.
4. **Milestone 4**: add one advanced feature: parser, lambda/closure, primitive library, or a small proof about `eval`.

For this lecture, finishing Milestones 1–3 is better than starting all four and leaving none testable.

Exercise: A Small Local Extension of the Project

The project should begin with a tiny extension, not with an entire parser all at once:

```
-- choose one extra constructor only
| mul : RichExpr -> RichExpr -> RichExpr
| eq  : RichExpr -> RichExpr -> RichExpr
```

Then extend the value type and the evaluator with the corresponding case. This is a good exercise because students can finish it within the writing block and still feel that they have genuinely extended a language implementation.

Live Lean: `lean-code/D5_05_ProjectStarter.lean`

Project References

These are the main references for continuing the project after class:

- *Write Yourself a Scheme in 48 Hours*: `https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours`
- *Functional Programming in Lean*, especially the chapters on lists, monads, and IO: `https://lean-lang.org/functional_programming_in_lean/`
- Lean Language Reference, for `syntax`, `macro`, and elaboration APIs: `https://lean-lang.org/doc/reference/latest/`

Meta-Programming in Lean 4

Lean 4 is also its own meta-language: you can extend the syntax from within.

- **Macro system** (`macro`, `syntax`): define new surface syntax that expands into existing Lean terms at parse time — no runtime cost.
- **Elaboration** (`elab`): run arbitrary Lean code during type-checking to construct terms, perform tactics, or generate definitions.
- **`Lean.Expr`**: the internal AST that represents all Lean terms — the same inductive tree structure you built above, but for Lean itself.

The `Expr` language you built is a small model of the same idea: syntax is data, and evaluation or elaboration gives that data meaning.

Architectural Trade-offs: Extrinsic vs. Intrinsic

Dimension	**Extrinsic Proofs**	**Intrinsic Proofs**
Readability	Often simpler at first	More information in types
Workflow	Code now, prove later	Prove while coding
API Safety	Caller must preserve invariant	Type enforces invariant
DTT Load	Standard theorem proving	More motive/type design

Practical heuristic: use extrinsic proofs while exploring, then add intrinsic structure at public boundaries.

Metaprogramming in Lean 4

Metaprogramming in Lean 4

Syntax, Macros, and the Elaboration System

Lean 4: Self-Hosting and Reflection

- In many theorem-proving systems, important implementation layers are written in a separate metalanguage.

Lean 4: Self-Hosting and Reflection

- In many theorem-proving systems, important implementation layers are written in a separate metalanguage.
- **Lean 4 is self-hosted.** The compiler, the parser, and the tactic engine are all written in Lean 4 itself.

Lean 4: Self-Hosting and Reflection

- In many theorem-proving systems, important implementation layers are written in a separate metalanguage.
- **Lean 4 is self-hosted.** The compiler, the parser, and the tactic engine are all written in Lean 4 itself.
- This lets users write Lean programs that inspect and generate Lean syntax and terms.

Lean 4: Self-Hosting and Reflection

- In many theorem-proving systems, important implementation layers are written in a separate metalanguage.
- **Lean 4 is self-hosted.** The compiler, the parser, and the tactic engine are all written in Lean 4 itself.
- This lets users write Lean programs that inspect and generate Lean syntax and terms.
- For large verification projects, this allows us to build **Domain Specific Languages (DSLs)** that hide boilerplate and automate proofs.

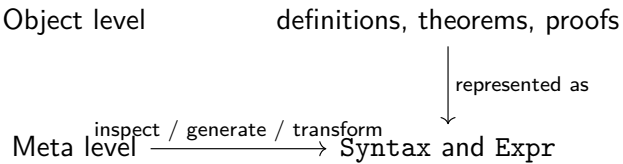
The Compilation Pipeline

To write a custom automated prover, you must understand how Lean translates a string of characters into a mathematically verified theorem.

$$\text{source text} \xrightarrow{\text{parse}} \texttt{Syntax} \xrightarrow{\text{macro}} \texttt{Syntax} \xrightarrow{\text{elaborate}} \texttt{Expr} \xrightarrow{\text{kernel}} \text{accepted term}$$

- **Parsing**: build an untyped syntax tree from text.
- **Macro expansion**: rewrite syntax into other syntax.
- **Elaboration**: resolve names, infer implicit arguments, synthesize typeclasses, and produce typed `Expr`.
- **Kernel checking**: verify the final term in Lean's small trusted core.

Object Level and Meta Level



A macro works mostly on `Syntax`. An elaborator can use the current environment and expected types to construct `Expr`. A tactic is an elaborator specialized to proof goals.

Step 1: Extending the Parser (`syntax`)

Before we can evaluate custom code, we must teach the Lean parser to recognize it. We do this using the `syntax` command.

```
-- Define a new syntax category for our project if needed
declare_syntax_cat my_cmds

-- Inject new syntax into the global `term` category
-- `syntax` [precedence] [elements] `:` [category]
syntax:10 term:10 " XOR " term:11 : term
```

- Lean's parser is an extensible Pratt parser.
- If you stop here and type `true XOR false`, Lean recognizes the syntax but still does not know its meaning. We must add a macro or elaborator.

Step 2: Macros (Syntax-to-Syntax)

- Macros are the simplest way to give meaning to new syntax.

Step 2: Macros (Syntax-to-Syntax)

- Macros are the simplest way to give meaning to new syntax.
- A macro operates purely on the `Syntax` level. It behaves like a highly structured, AST-aware search-and-replace.

Step 2: Macros (Syntax-to-Syntax)

- Macros are the simplest way to give meaning to new syntax.
- A macro operates purely on the `Syntax` level. It behaves like a highly structured, AST-aware search-and-replace.
- **Crucial Limitation:** Macros are *type-blind*. A macro cannot check if a variable is a `Nat` or a `List`. It only knows about the shape of the code.

Step 2: Macros (Syntax-to-Syntax)

- Macros are the simplest way to give meaning to new syntax.
- A macro operates purely on the `Syntax` level. It behaves like a highly structured, AST-aware search-and-replace.
- **Crucial Limitation:** Macros are *type-blind*. A macro cannot check if a variable is a `Nat` or a `List`. It only knows about the shape of the code.
- **Quotations (`):** We use the backtick to construct syntax trees.

Step 2: Macros (Syntax-to-Syntax)

- Macros are the simplest way to give meaning to new syntax.
- A macro operates purely on the `Syntax` level. It behaves like a highly structured, AST-aware search-and-replace.
- **Crucial Limitation:** Macros are *type-blind*. A macro cannot check if a variable is a `Nat` or a `List`. It only knows about the shape of the code.
- **Quotations (`):** We use the backtick to construct syntax trees.
- **Antiquotations ($):** We use the dollar sign to inject captured variables into those syntax trees.

Writing a Macro

Let's write the macro rule for our `XOR` operator.

```
-- Tell Lean how to rewrite the syntax tree
macro_rules
| `($l XOR $r) => `((!$l && $r) || ($l && !$r))

-- Usage
#eval true XOR false  -- Output: true
```

Lean provides a shortcut `macro` command for syntax-to-syntax expansions:

```
macro "twice " x:term : term => `($x + $x)
```

Live Lean: `lean-code/D5_06_Meta.lean`

Step 3: The Elaborator (`Elab`)

When macros bottom out or you need to inspect types, the **Elaborator** takes over. Its job is to bridge the gap between human-readable text and kernel-readable math.

Step 3: The Elaborator (`Elab`)

When macros bottom out or you need to inspect types, the **Elaborator** takes over. Its job is to bridge the gap between human-readable text and kernel-readable math.

- **The Input:** Untyped `Syntax`.

Step 3: The Elaborator (`Elab`)

When macros bottom out or you need to inspect types, the **Elaborator** takes over. Its job is to bridge the gap between human-readable text and kernel-readable math.

- **The Input:** Untyped `Syntax`.
- **The Output:** A fully typed `Expr` (Expression) or a modified State.

Step 3: The Elaborator (`Elab`)

When macros bottom out or you need to inspect types, the **Elaborator** takes over. Its job is to bridge the gap between human-readable text and kernel-readable math.

- **The Input:** Untyped `Syntax`.
- **The Output:** A fully typed `Expr` (Expression) or a modified State.
- **The Work:** To build an `Expr`, the Elaborator must resolve implicit arguments, synthesize typeclasses, unfold definitions, and solve unification problems using Metavariables (holes).

The Three Flavors of Elaboration Monads

Depending on what your project is building, Lean places you in a specific Monadic context with different capabilities:

The Three Flavors of Elaboration Monads

Depending on what your project is building, Lean places you in a specific Monadic context with different capabilities:

1. **`CommandElabM`:** Operates at the top level of a file. Used to build commands like `#check`, `def`, or custom environment modifiers. Returns `Unit`.

The Three Flavors of Elaboration Monads

Depending on what your project is building, Lean places you in a specific Monadic context with different capabilities:

1. **`CommandElabM`:** Operates at the top level of a file. Used to build commands like `#check`, `def`, or custom environment modifiers. Returns `Unit`.
2. **`TermElabM`:** Operates inside definitions. Takes a `Syntax` tree and an expected type, and attempts to return an `Expr`.

The Three Flavors of Elaboration Monads

Depending on what your project is building, Lean places you in a specific Monadic context with different capabilities:

1. **`CommandElabM`:** Operates at the top level of a file. Used to build commands like `#check`, `def`, or custom environment modifiers. Returns `Unit`.
2. **`TermElabM`:** Operates inside definitions. Takes a `Syntax` tree and an expected type, and attempts to return an `Expr`.
3. **`TacticM`:** Operates inside a `by` block. It manages a list of open Metavariables (the "Goals") and applies transformations to close them.

Writing a Command Elaborator

Let's write a custom top-level command that just prints "Hello" to the compiler trace. We use the `elab` shortcut.

```
open Lean Elab Command

-- Define the syntax and bind it directly to CommandElabM
elab "#hello" : command => do
  logInfo "Hello from the Elaborator!"

-- Invoke it
#hello
-- Lean compiler outputs: Hello from the Elaborator!
```

The `elab` command binds a piece of syntax to code running in an elaboration monad.

Live Lean: `lean-code/D5_06_Meta.lean`

Writing a Tactic Elaborator: Interacting with State

Tactics are just elaborators that run in `TacticM`. Let's write a tactic that inspects the current goal and prints its type.

```
open Lean Elab Tactic

elab "print_goal" : tactic => do
  let mvarId <- getMainGoal
  let decl <- mvarId.getDecl
  logInfo m!"The current goal type is: {decl.type}"
```

These primitives let a tactic inspect the current goal and decide what proof step to run next.

Live Lean: `lean-code/D5_06_Meta.lean`

Reusing Existing Tactics: `evalTactic`

You do not always have to manually manipulate `Expr` trees. Often the best design is to parse a small custom command and then call existing tactics.

```
elab "custom_intro " name:ident : tactic => do
  evalTactic ←( `(tactic| intro $name:ident))
```

- This style is useful when your extension mainly packages an existing proof pattern.
- Direct `Expr` manipulation is still available, but it should be introduced only when the simpler tactic-composition style is not enough.

Live Lean: `lean-code/D5_06_Meta.lean`

Summary: Macro vs. Elab

When designing the architecture of a major Lean project, how do you choose?

Feature	**Macros**	**Elaborators (`elab`)**
Shape	Syntax -> Syntax	Syntax -> Expr / State
Type Aware	Blind	Fully Type-Aware
Complexity	Usually simpler	More powerful API
Use Case	Syntactic Sugar, Notation	Custom Tactics, Inference

Design rule: use a macro when the feature is only syntactic sugar. Use an elaborator when the feature must inspect types, goals, local context, or the environment.

Break 3: Project Writing Block

This is the final 10-minute writing block. Students should not try to finish a whole interpreter here; they should make one concrete, testable extension.

- Option A: add one constructor to `RichExpr` and one case to `eval`.
- Option B: add one better error message to `lookup`, `asNat`, or `asBool`.
- Option C: run `D5_06_Meta.lean` and explain where `Syntax` ends and `Expr` begins.

The permanent goal is a readable project outline, not a rushed final implementation.

What We Covered Today

Part 1 — Functional Programming

- Pure functions, immutability, pattern matching
- Recursive types: `Nat`, `List`, `Tree`
- Higher-order functions: `map`, `filter`, `fold`
- Termination: structural vs. well-founded recursion

Part 2 — Monads

- `Functor` / `Applicative` / `Monad`
- do-notation desugaring
- `Option`, `Except`, `IO`, `State`
- Monad transformers (preview)

What We Covered Today

Part 3 — Expression Languages

- Syntax as an inductive type
- Evaluation pipeline: `parse` $\rightarrow$ `eval`
- Typed vs. untyped interpreters
- Meta-programming with `Syntax` and `Macro`

Takeaways

- Lean programs and proofs are both typed terms
- Monads generalize sequential computation
- DSLs let you extend Lean while keeping type checking explicit

The Hierarchy We Have Built

Layer	Abstraction	Lean typeclass
Data	Inductive types	(built-in)
Transformation	Pure functions	`Function`
Mapping	Structure-preserving map	`Functor`
Sequencing	Context-aware chaining	`Monad`
Effects	I/O, state, error	`IO`, `StateT`, …
Language	Syntax + semantics	`Macro`, `Elab`

Each layer builds on the one below. Lean's type system ensures the connections are consistent.

Connections to the Rest of the Course

- **D3 (Typeclasses)** — `Functor`, `Monad` are typeclass instances; today's code *is* the typeclass system in action.

Connections to the Rest of the Course

- **D3 (Typeclasses)** — `Functor`, `Monad` are typeclass instances; today's code *is* the typeclass system in action.
- **D4 (Analysis / Topology)** — `Filter` in Mathlib uses map-like structure; limits are encoded with `Tendsto` by moving information through filters.

Connections to the Rest of the Course

- **D3 (Typeclasses)** — `Functor`, `Monad` are typeclass instances; today's code *is* the typeclass system in action.
- **D4 (Analysis / Topology)** — `Filter` in Mathlib uses map-like structure; limits are encoded with `Tendsto` by moving information through filters.
- **D6 (Verify)** — monadic models can track extra information, such as cost or state, while keeping the main program readable.

Connections to the Rest of the Course

- **D3 (Typeclasses)** — `Functor`, `Monad` are typeclass instances; today's code *is* the typeclass system in action.
- **D4 (Analysis / Topology)** — `Filter` in Mathlib uses map-like structure; limits are encoded with `Tendsto` by moving information through filters.
- **D6 (Verify)** — monadic models can track extra information, such as cost or state, while keeping the main program readable.
- **Meta-programming** — `Macro` and `Elab` (Part 3) are the foundation for writing custom tactics and notation in Lean 4.

Further Reading

- *Functional Programming in Lean* — David Thrane Christiansen, free online.
- *Theorem Proving in Lean 4* — Jeremy Avigad et al., free online.
- *Lean 4 Metaprogramming* — Arthur Paulino et al., free online.
- *Write Yourself a Scheme in 48 Hours* — Wikibooks, Haskell interpreter tutorial.
- *Introduction to Homotopy Type Theory* — Egbert Rijke.
- *Types and Programming Languages* — Benjamin C. Pierce.

Project Directions

- Write a **Sudoku solver** with backtracking in `StateT`.
- Write a **Scheme-like interpreter**, following the staged project outline from today.
- Implement a **Stream** abstraction for productive infinite sequences.
- Add one proof about your evaluator, such as a simple preservation or determinism statement for a tiny fragment.

Books and URLs: Lean References I

- Functional Programming in Lean:
  `https://lean-lang.org/functional_programming_in_lean/`
- Theorem Proving in Lean 4:
  `https://docs.lean-lang.org/theorem_proving_in_lean4/`

Books and URLs: Lean References II

- Lean Language Reference: `https://lean-lang.org/doc/reference/latest/`
- Lean 4 Metaprogramming Book: `https://leanprover-community.github.io/lean4-metaprogramming-book/`

Books and URLs: Project and Type Theory

- Write Yourself a Scheme in 48 Hours:
  `https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours`
- Egbert Rijke, Introduction to Homotopy Type Theory:
  `https://arxiv.org/abs/2212.11082`
- TAPL companion page: `https://www.cis.upenn.edu/~bcpierce/tapl/`

Final Takeaway

"A monad is just a monoid in the category of endofunctors."

— Philip Wadler (paraphrasing Saunders Mac Lane)

Final Takeaway

"A monad is just a monoid in the category of endofunctors."

— Philip Wadler (paraphrasing Saunders Mac Lane)

- Don't be intimidated by the slogan.
  A monad is a design pattern for *sequencing effects* — and you already know how to use it.

Final Takeaway

"A monad is just a monoid in the category of endofunctors."

— Philip Wadler (paraphrasing Saunders Mac Lane)

- Don't be intimidated by the slogan.
  A monad is a design pattern for *sequencing effects* — and you already know how to use it.
- The category-theoretic view (D3) and the programming view (today) describe the same structure from different levels of detail.

Final Takeaway

"A monad is just a monoid in the category of endofunctors."

— Philip Wadler (paraphrasing Saunders Mac Lane)

- Don't be intimidated by the slogan.
  A monad is a design pattern for *sequencing effects* — and you already know how to use it.
- The category-theoretic view (D3) and the programming view (today) describe the same structure from different levels of detail.
- In Lean, advanced users can formalize these correspondences instead of treating them only as informal analogies.

Final Takeaway

"A monad is just a monoid in the category of endofunctors."

— Philip Wadler (paraphrasing Saunders Mac Lane)

- Don't be intimidated by the slogan.
  A monad is a design pattern for *sequencing effects* — and you already know how to use it.
- The category-theoretic view (D3) and the programming view (today) describe the same structure from different levels of detail.
- In Lean, advanced users can formalize these correspondences instead of treating them only as informal analogies.
- Next time, we will move from programming patterns to **verification**.