



*"OOP makes code understandable
by encapsulating moving parts. FP
does so by minimizing moving
parts."*

— Michael Feathers

Week 2 of 11

PROOF101 Week 2 Functional Programming

AUB Math Society
Spring 2027

Functional Programming

Daniel Dia & Guest Lecturers

American University of Beirut (AUB)

<https://danieldia.me/proof101/>

Section 1

Historical Exposition: How did we get here?

The Mathematical Lineage (Alonzo Church)

Lambda Calculus (1936)

Alonzo Church developed lambda calculus as a formal system for expressing computation through function abstraction and application.

Main Ideas:

- Functions as first-class values
- Computation by substitution (β -reduction)
- Everything is a function (even numbers and booleans!)
- No mutable state, no side effects

Legacy:

- Foundation of functional programming languages
- Influenced: LISP (1958), ML (1973), Haskell (1990), Lean (2013)
- Proved equivalent to Turing machines (Church-Turing thesis)

The Mechanical Lineage (Alan Turing)

Turing Machine (1936)

Alan Turing proposed a theoretical machine with an infinite tape, a head that reads/writes, and state transitions.

Main Ideas:

- Sequential execution of instructions
- Mutable state (tape contents, head position)
- Direct manipulation of memory
- Step-by-step computation

Legacy:

- Foundation of imperative programming
- Influenced: FORTRAN (1957), C (1972), C++ (1985), Java (1995)
- Directly inspired von Neumann architecture
- Dominated programming for 70+ years

The Era of "Spaghetti Code"

The "Software Crisis" (1960s)

As programs grew larger, they became impossible to understand and maintain.

The Problems:

- **GOTO statements** created incomprehensible control flow
- **Global state** meant any function could break anything
- **No abstraction** – code duplication everywhere
- Projects consistently over budget, late, or failed entirely

Dijkstra's Response (1968): "Go To Statement Considered Harmful"

He proposed **structured programming**: loops, conditionals, functions instead of arbitrary jumps.

The Dream: Smalltalk (1972)

Alan Kay's Vision: "Objects All the Way Down"

Pure OOP Principles:

- Everything is an object (even classes!)
- Objects communicate by sending messages
- Objects encapsulate state and behavior
- Late binding and polymorphism

The Promise of OOP:

- Modularity: compose complex systems from simple objects
- Reusability: objects as building blocks
- Natural modeling: objects $\leftrightarrow$ real-world entities

"I invented the term Object-Oriented, and I can tell you I did not have C++ in mind." — Alan Kay

The Reality: C++ (1985) and Java (1995)

The "Corruption" of OOP (okay, this is a *tiny* comedic exaggeration)

What went wrong:

- **Mutable state everywhere** – objects became bags hiding state
- **Deep inheritance hierarchies** – "fragile base class" problem
- **Side effects hidden in methods** – unpredictable behavior
- **Shared mutable state** – concurrency nightmares

Joe Armstrong (Erlang creator): *"The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana... and the entire jungle."*

The Null Pointer (1965)

Tony Hoare's "Billion Dollar Mistake"

The Main Issue:

- Every reference can be `null`
- Type system doesn't track which values might be `null`
- Result: `NullPointerException`, `SEGFAULT`, and so on

Hoare's Apology (2009): *"I call it my billion-dollar mistake. It was the invention of the null reference in 1965... This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."*

Functional Programming's Solution: Option/Maybe types – they explicitly handle absence!

The Era of "Concurrency Hell": The Death of OOP in Systems

The Multicore Revolution (2005+)

CPU speeds stopped increasing. To get faster, we needed *more cores*.

Shared Mutable State caused all sorts of issues:

- **Race conditions** – two threads modify the same object
- **Deadlocks** – threads wait for each other forever
- **Data races** – unpredictable ordering of operations
- **Heisenberg bugs** – disappear when you try to debug them

Why Functional Programming "Won":

- **Immutability** – no shared mutable state = no race conditions
- **Pure functions** – safe to run in parallel automatically
- **Referential transparency** – easier to reason about

Result: Erlang, Haskell, Clojure, Scala, Rust, and modern JavaScript all embrace FP.

Section 2

The Theory of Functional Programming

What is a Pure Function?

Definition: A function that always returns the same output for the same input and has no observable side effects.

Two requirements:

1. **Deterministic:** Same input → same output (always)
2. **No side effects:** Doesn't modify external state or perform I/O

```
const xs = [1, 2, 3, 4, 5];  
  
// PURE: Always returns same result for same input  
xs.slice(0, 3); // [1, 2, 3]  
xs.slice(0, 3); // [1, 2, 3]  
  
// IMPURE: Mutates array, different results  
xs.splice(0, 3); // [1, 2, 3]  
xs.splice(0, 3); // [4, 5]
```

Why Do We Care About Purity?

Pure functions are:

1. Predictable

- Behavior determined **entirely** by inputs
- No hidden dependencies on external state
- Easy to understand and reason about

2. Testable

- No setup or teardown needed
- Just provide input, check output

3. Composable

- Can compose pure functions safely
- Evaluation is independent (no shared or mutable state)
- Build complex behavior from simple parts

*Purity is the foundation of equational reasoning
in Lean 4!*

Characteristics of Pure Functions

Deterministic behavior:

```
-- Pure: output determined only by input
def square (n : Nat) : Nat := n * n

#eval square 5 -- 25
#eval square 5 -- 25 (always the same!)
```

No side effects:

- Doesn't modify arguments
- Doesn't change global variables
- Doesn't perform I/O (print, file, network)
- Doesn't throw exceptions (in FP, we return Result/Option)

Consequence: Functions become like mathematical functions – predictable, testable, and composable.

Purity and External State

Depending on external state breaks purity:

```
// IMPURE: Depends on external mutable variable
let minimum = 21;
const checkAge = age => age >= minimum;

checkAge(20); // false
minimum = 18; // (someone changed the global variable)
checkAge(20); // true
```

Pure version (self-contained):

```
// PURE: All dependencies explicit
const checkAge = (age, minimum) => age >= minimum;

checkAge(20, 21); // false
checkAge(20, 21); // false (always)
checkAge(20, 18); // true (different input, so different output is OK)
```

Main point: Make your dependencies explicit in parameters!

What Counts as a "Side Effect"?

Side effects change the world outside the function:

- Modifying a variable outside the function's scope
- Changing the file system (create, delete, modify files)
- Writing to a database or making network requests
- Printing to screen or writing to logs
- Obtaining user input
- Modifying the DOM in a web page
- Throwing exceptions
- Accessing system state (current time, random numbers)
- Modifying data structures (arrays, objects) passed as arguments

Key point: We don't completely forbid side effects, we just want to *contain* them and control when they happen!

Examples: Benefits of Purity (1)

Memoization: "Cache" (store) results of expensive function calls

```
const memoize = (f) => {  
  const cache = {};  
  return (...args) => {  
    const key = JSON.stringify(args);  
    if (!(key in cache)) {  
      cache[key] = f(...args); // compute once  
    }  
    return cache[key]; // return cached value  
  };  
};  
  
const expensiveSquare = memoize(x => {  
  console.log(`Computing ${x}^2...`);  
  return x * x;  
});  
  
expensiveSquare(4); // "Computing 4^2..." → 16  
expensiveSquare(4); // → 16 (from cache)
```

This only works for pure functions! Impure functions can't be safely cached.

Examples: Benefits of Purity (2)

Impure code hides dependencies:

```
const signUp = (attrs) => {  
  const user = saveUser(attrs);      // Hidden DB dependency  
  welcomeUser(user);                 // Hidden email service  
}; // IMPURE: Where do saveUser and welcomeUser come from?
```

Pure code makes dependencies explicit:

```
const signUp = (db, emailService, attrs) => {  
  const user = saveUser(db, attrs);  
  welcomeUser(emailService, user);  
  return user;  
}; // PURE: All dependencies are parameters
```

Testing benefits:

- No need to set up databases or email services
- Just pass mock objects as parameters (+ no cleanup needed after tests)

Examples: Benefits of Purity (3)

Referential transparency: Can replace function call with its value

Example: If $f(3) = 9$, then:

- $f(3) + f(3)$ equals $9 + 9$
- $2 * f(3)$ equals $2 * 9$
- This allows us to algebraically reason about code

Equational reasoning allows for some cool stuff:

- Substituting "equals for equals" (like algebra)
- Refactoring with confidence
- Optimizing code automatically (compiler)
- Understanding the code more easily

Contrast this with impure code: Can't replace `readFile()` with its value, since it might return different things!

Examples: Benefits of Purity (4)

Pure functions are inherently thread-safe:

- No shared mutable state to protect
- No race conditions possible
- No need for locks or synchronization
- Can run in parallel automatically!

Example: Parallel map

- `map f [1,2,3,4,5,6,7,8]` with a pure function `f`
- Each `f(i)` is independent
- Can compute all in parallel with zero coordination
- Guaranteed to give same result as sequential execution would

Very relevant for modern hardware: Multi-core processors need parallelism! **From this point forward, we will strive to write all functions in a pure way.**

What Does "First-Class" Mean?

"First-class citizen": A value that can be used anywhere other values can be used

In most languages, numbers are first-class:

- You can store them in variables: `x = 42`
- You can pass them to functions: `f(42)`
- You can return them from functions: `return 42`
- You can store them in data structures: `[42, 43, 44]`

First-class functions: Functions can do all of the above!

- Stored in variables: `f = fun x => x + 1`
- Passed to functions: `map f xs`
- Returned from functions: `return (fun x => x + n)`
- Stored in data structures: `[f1, f2, f3]`

The Motivation Behind First-Class Functions

They enable abstraction over computation patterns! Basically: you can chain them together in all sorts of ways for your convenience.

Without first-class functions:

- Write separate loops for each transformation
- `doubleList`, `squareList`, `incrementList`, ...
- Duplicate loop logic everywhere
- Hard to see the pattern

With first-class functions:

- Write `map` once
- `map double xs`, `map square xs`, `map increment xs`
- Abstract the pattern (transform each element)
- Separate "what" from "how"

This idea is the foundation of functional programming!

First-Class Functions in Lean: Storing in Variables

Functions are actual values, i.e. they can be stored in variables:

```
-- Let's store a function in a variable
def double : Nat → Nat := fun x => x * 2

-- We can use it like any other value
#eval double 5      -- 10

-- We can even create multiple "copies"
def myDouble := double
def alsoDouble := double

#eval myDouble 3    -- 6
#eval alsoDouble 3  -- 6
```

Main point: `double` is just a name for a value (that happens to be a function).

First-Class Functions: Passing to Functions

Functions can also take other functions as arguments:

```
-- Takes a function and applies it twice
def twice (f :  $\alpha \rightarrow \alpha$ ) (x :  $\alpha$ ) :  $\alpha$  :=
  f (f x)

#eval twice ( $\cdot + 1$ ) 5      -- 7 (5 + 1 + 1)
#eval twice ( $\cdot * 2$ ) 3      -- 12 (3 * 2 * 2)

-- Apply function n times
def applyN (f :  $\alpha \rightarrow \alpha$ ) : Nat  $\rightarrow$   $\alpha \rightarrow \alpha$ 
  | 0, x => x
  | n+1, x => f (applyN f n x)

#eval applyN ( $\cdot + 1$ ) 5 0    -- 5 (add 1 five times to 0)
#eval applyN ( $\cdot * 2$ ) 3 1    -- 8 (double three times: 1→2→4→8)
```

Pattern: The function f is just another parameter!

First-Class Functions: Returning Functions

Functions can return other functions as output:

```
-- Returns a function that adds n
def makeAdder (n : Nat) : Nat → Nat :=
  fun x => x + n

def add5 := makeAdder 5
def add10 := makeAdder 10

#eval add5 3    -- 8  (3 + 5)
#eval add10 3   -- 13 (3 + 10)

-- Returns a function that multiplies by n
def makeMultiplier (n : Nat) : Nat → Nat :=
  fun x => x * n

def double := makeMultiplier 2
def triple := makeMultiplier 3

#eval double 7   -- 14 (7 * 2)
#eval triple 7   -- 21 (7 * 3)
```

This is called a "function factory" or "higher-order function" (I prefer the latter name)

First-Class Functions: In Data Structures

Functions can be stored in lists, tuples, etc:

```
-- A list of functions
def operations : List (Nat → Nat) :=
  [(· + 1), (· * 2), (· * 5)]

-- Apply each function to a value with map
def applyAll (fs : List (Nat → Nat)) (x : Nat) : List Nat :=
  fs.map (fun f => f x)

#eval applyAll operations 5 -- [6, 10, 25]

-- A pair of functions
def mathOps : (Nat → Nat → Nat) × (Nat → Nat → Nat) :=
  ((· + ·), (· * ·))

#eval mathOps.1 3 4 -- 7 (addition)
#eval mathOps.2 3 4 -- 12 (multiplication)
```

This enables powerful patterns like the strategy pattern without OOP!

What is a Higher-Order Function?

Definition: A function that either:

- Takes one or more functions as arguments, OR
- Returns a function as its result, OR
- Both!

Examples of higher-order functions we've seen so far:

- `twice` – takes function, applies it twice
- `makeAdder` – returns a function
- `map` – takes function, applies to each element
- `filter` – takes predicate function
- `compose` – takes two functions, returns their composition

Why is this so powerful? They abstract over computation patterns!

Higher-Order Functions: The Power of Abstraction

Without higher-order functions:

- `doubleList` – loop through list, double each element
- `squareList` – loop through list, square each element
- `incrementList` – loop through list, add 1 to each
- Notice the amount of duplicated loop logic!

With **map** (higher-order function):

- Write the loop logic once in `map`
- Do `map double xs`
- Do `map square xs`
- Do `map increment xs`
- We abstracted away the pattern: "apply this function to each element"

Basically, higher-order functions let us separate the "what" from "how" in programming.

Map: Transform Every Element

Map Pattern: Apply a transformation to every element

```
def map (f :  $\alpha \rightarrow \beta$ ) : List  $\alpha \rightarrow$  List  $\beta$ 
| []      => []
| x :: xs => f x :: map f xs

-- Examples:
#eval map ( $\cdot$  * 2) [1, 2, 3, 4]      -- [2, 4, 6, 8]
#eval map ( $\cdot$  + 1) [1, 2, 3, 4]     -- [2, 3, 4, 5]
#eval map String.length ["hi", "hello", "hey"] -- [2, 5, 3]
```

Type signature: $(\alpha \rightarrow \beta) \rightarrow \text{List } \alpha \rightarrow \text{List } \beta$

- Takes function from α to β
- Takes list of α
- Returns list of β
- Each element transformed independently

Filter: Select Elements

Filter Pattern: Keep only elements that satisfy a condition (predicate)

```
def filter (p :  $\alpha \rightarrow \text{Bool}$ ) : List  $\alpha \rightarrow$  List  $\alpha$ 
| []      => []
| x :: xs => if p x then x :: filter p xs
           else filter p xs

-- Examples:
#eval filter (· > 5) [1, 8, 3, 9, 2, 7] -- [8, 9, 7]

#eval filter (· % 2 == 0) [1,2,3,4,5,6] -- [2, 4, 6]

#eval filter (fun s => s.length > 3) ["hi", "hello", "bye"] -- ["hello"]
```

Type: $(\alpha \rightarrow \text{Bool}) \rightarrow \text{List } \alpha \rightarrow \text{List } \alpha$

- Takes predicate function (returns Bool)
- Returns subset where predicate is true

Filter: Common Use Cases

Filter is everywhere in real code:

- **Data cleaning**, e.g. removing null/invalid values
- **Searching**, e.g. finding items matching criteria
- **(Input) Validation**, e.g. keeping only valid inputs
- **Filtering API results**, e.g. get only what you need
- **Permission checks**, e.g. show only authorized items

And it combines really well with map:

- You can filter, then transform: `map f (filter p xs)`
- You can transform, and then filter: `filter p (map f xs)`
- In some languages (e.g. Rust), this is called "method chaining"

Fold: Reduce List to a Single Value

Fold Pattern: Combines all elements using a certain binary operation

```
def foldr (f :  $\alpha \rightarrow \beta \rightarrow \beta$ ) (init :  $\beta$ ) : List  $\alpha \rightarrow \beta$ 
  | []      => init
  | x :: xs => f x (foldr f init xs)

-- Examples:
#eval foldr ( $\cdot + \cdot$ ) 0 [1, 2, 3, 4]      -- 10 (sum)
#eval foldr ( $\cdot * \cdot$ ) 1 [1, 2, 3, 4]      -- 24 (product)
#eval foldr ( $\cdot :: \cdot$ ) [] [1, 2, 3]      -- [1, 2, 3] (identity)
#eval foldr Nat.max 0 [3, 1, 4, 1, 5]      -- 5 (maximum)
```

Type signature: $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \text{List } \alpha \rightarrow \beta$

- The combining function: $\alpha \rightarrow \beta \rightarrow \beta$
- The initial or default value: β
- The input list: List α
- The single result: β

Understanding Fold

Fold replaces list constructors!

List structure:

- $[1, 2, 3] = 1 :: (2 :: (3 :: []))$
- Uses $::$ (cons) and $[]$ (nil)

Fold replaces constructors:

- `foldr f z` replaces $::$ with `f` and $[]$ with `z`
- $1 :: (2 :: (3 :: []))$ becomes `f 1 (f 2 (f 3 z))`

Example: Sum

- `foldr (+) 0 [1,2,3]`
- $1 :: (2 :: (3 :: [])) \rightarrow 1 + (2 + (3 + 0))$
- Result: 6

Fold: The Universal List Function

Many list operations are (*secretly*) just folds. Don't believe me? See for yourself:

- `sum`: `foldr (+) 0`
- `product`: `foldr (*) 1`
- `length`: `foldr (\_ acc => acc + 1) 0`
- `reverse`: `foldl (\acc x => x :: acc) []`
- `map f`: `foldr (\x acc => f x :: acc) []`
- `filter p`: `foldr (\x acc => if p x then x :: acc else acc) []`

Fold is incredibly powerful! It's the "mother of all list operations."

Fold Right vs Fold Left

Two ways to "fold":

```
-- Right fold: processes right to left
def foldr (f :  $\alpha \rightarrow \beta \rightarrow \beta$ ) (init :  $\beta$ ) : List  $\alpha \rightarrow \beta$ 
| []      => init
| x :: xs => f x (foldr f init xs)

-- Left fold: processes left to right
def foldl (f :  $\beta \rightarrow \alpha \rightarrow \beta$ ) (init :  $\beta$ ) : List  $\alpha \rightarrow \beta$ 
| []      => init
| x :: xs => foldl f (f init x) xs
```

The main difference is clear:

- **foldr**: $f \ 1 \ (f \ 2 \ (f \ 3 \ z))$, i.e. it is right-associative
- **foldl**: $f \ (f \ (f \ z \ 1) \ 2) \ 3$, i.e. it is left-associative
- Note: **foldl** is tail-recursive (which enables Tail-Call Optimization at the compiler level)

Fold Right Example: Subtraction

Right fold with subtraction:

```
-- foldr (-) 10 [1, 2, 3]
-- = 1 - (2 - (3 - 10))
-- = 1 - (2 - (-7))
-- = 1 - 9
-- = -8

#eval foldr (· - ·) 10 [1, 2, 3] -- -8
```

Execution trace:

1. We process the innermost operations first: $3 - 10 = -7$
2. Then, we do: $2 - (-7) = 9$
3. Finally, we get: $1 - 9 = -8$

It associates to the right: $1 - (2 - (3 - 10))$

Fold Left Example: Subtraction

Left fold with subtraction:

```
-- foldl (-) 10 [1, 2, 3]
-- = ((10 - 1) - 2) - 3
-- = (9 - 2) - 3
-- = 7 - 3
-- = 4

#eval foldl (. - .) 10 [1, 2, 3] -- 4
```

Execution trace:

1. We start with the accumulator: 10
2. We process left to right: $10 - 1 = 9$
3. Then, we do: $9 - 2 = 7$
4. Finally, we get: $7 - 3 = 4$

It associates to the left: $((10 - 1) - 2) - 3$

Summary: When to Use foldr vs foldl?

Use foldr when:

- Operation is naturally right-associative
- Building data structures (cons onto a list)
- Need to preserve order in certain operations
- Working with infinite lists (in lazy languages)
- Example: `foldr (:) [] xs = identity`

Use foldl when:

- Operation is naturally left-associative
- Need efficiency (tail recursion)
- Accumulating a result (sum, product, max)
- Building result incrementally
- Example: `foldl (+) 0 xs = sum` (very efficient)

For **commutative & associative operations** (e.g. `+`, `*`) it doesn't really matter, but `foldl` is more efficient.

Function Composition

Compose Pattern: Chains functions together

```
def compose (f :  $\beta \rightarrow \gamma$ ) (g :  $\alpha \rightarrow \beta$ ) :  $\alpha \rightarrow \gamma$  :=  
  fun x => f (g x)  
  
notation 90 f " ∘ " g => compose f g  
  
def addOne := (· + 1)  
def double := (· * 2)  
  
#eval (addOne ∘ double) 5      -- 11 ((5*2) + 1)  
#eval (double ∘ addOne) 5      -- 12 ((5+1)*2)
```

Type signature: $(\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$

Intuition: $(f \circ g)(x)$ means "first apply g, then apply f to the result"

Composition: Building Pipelines

Composition lets you build "transformation pipelines"!

Without composition:

- You'd write $f(g(h(x)))$, which is hard to read (inside-out)
- You also have to trace execution backwards
- Parentheses $()$ get unwieldy and chaotic with many functions

With composition:

- $(f \circ g \circ h)(x)$ – is much more natural (right-to-left)
- Or define some `pipeline = f ∘ g ∘ h`, and then use: `pipeline(x)`
- You can easily name intermediate transformations for convenience
- You can reuse composed functions

Function composition is associative: $(f \circ g) \circ h = f \circ (g \circ h)$

Composition Example: Data Processing Pipeline

Let's look at some examples of composing functions to form a data processing pipeline:

```
-- Individual transformations
def trim (s : String) : String := s.trim
def toLower (s : String) : String := s.toLowerCase
def replaceSpaces (s : String) : String :=
  s.replace " " "_"

-- Compose into pipeline
def slugify := replaceSpaces ◦ toLower ◦ trim

#eval slugify " Hello World "
-- "hello-world"

-- Can also chain with map
def slugifyAll := map slugify

#eval slugifyAll [" Hello ", " WORLD "]
-- ["hello", "world"]
```

Notice the pattern: We can build complex transformations from simple, reusable pieces.

What is Currying?

Currying: Transforms a function taking multiple arguments into chain of functions each taking one argument. Note: The attentive student will realize the direct connection with the λ -calculus.

Transform:

- From: $f : (\alpha \times \beta) \rightarrow \gamma$ (function taking pair)
- To: $f : \alpha \rightarrow \beta \rightarrow \gamma$ (function returning function)
- Notation: $\alpha \rightarrow \beta \rightarrow \gamma = \alpha \rightarrow (\beta \rightarrow \gamma)$

For culture: it is named after Haskell Curry (mathematician, 1900-1982)

- Though currying was actually invented by Gottlob Frege and Moses Schönfinkel
- Extremely common in logic and functional programming

In Lean: All multi-argument functions are automatically curried!

Why Currying Matters

Currying enables partial application.

Without currying:

- Function needs all arguments at once
- `add(3, 4)` gives 7
- Can't easily create "add 3 to something" function

With currying:

- `add : Nat → Nat → Nat`
- `add 3 : Nat → Nat` (partially applied - valid function!)
- `add 3 4 : Nat` (fully applied - gives result)
- Can create specialized functions easily

This is fundamental to functional programming style!

Currying in Action (Examples)

All the functions below are actually equivalent.

```
-- Explicit nested lambdas
def add1 : Nat → Nat → Nat :=
  fun x => fun y => x + y

-- Implicit currying (most common)
def add2 (x : Nat) (y : Nat) : Nat := x + y

-- Using operator
def add3 : Nat → Nat → Nat := (· + ·)

-- All have the same type
#check add1 -- Type signature: Nat → Nat → Nat
#check add2 -- Type signature: Nat → Nat → Nat
#check add3 -- Type signature: Nat → Nat → Nat
```

Understanding the type: $\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$ means $\text{Nat} \rightarrow (\text{Nat} \rightarrow \text{Nat})$

- It is a function taking Nat
- It is returning a function taking Nat
- Which itself returns Nat

Partial Application

Partial application: Supply some arguments, get back a function waiting for the rest

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

-- Fully applied (all arguments provided)
#eval add 3 4 -- 7 : Nat

-- Partially applied (only one argument provided)
def add3 := add 3
#check add3 -- Nat → Nat (this is a valid function!)

-- Use the partially applied function
#eval add3 4 -- 7
#eval add3 10 -- 13
#eval add3 100 -- 103
```

Notice the pattern: You fix some parameters, you get a specialized function

Partial Application: Creating "Specialized" Functions

```
-- General multiplication function
def multiply (a : Nat) (b : Nat) : Nat := a * b

-- Let's create specialized functions via partial application
def double := multiply 2
def triple := multiply 3
def quadruple := multiply 4

#eval double 7      -- 14
#eval triple 7      -- 21
#eval quadruple 7   -- 28

-- Use with higher-order functions
#eval map double [1, 2, 3, 4] -- [2, 4, 6, 8]
#eval map triple [1, 2, 3, 4] -- [3, 6, 9, 12]
#eval filter (· > 5) (map double [1, 2, 3, 4, 5]) -- [6, 8, 10]
```

This is incredibly powerful for code reuse!

Partial Application: Real-World Examples

Common programming patterns using partial application.

Configuration functions:

- `sendRequest = httpPost apiUrl authToken`
- `sendRequest data (use configured version)`

Validators:

- `isLongerThan min = ( $\lambda s \Rightarrow s.length > min$ )`
- `filter (isLongerThan 5) strings`

Comparators:

- `isGreaterThan x = ( $\lambda y \Rightarrow y > x$ )`
- `filter (isGreaterThan 10) numbers`

Common thread: Create families of related functions from one general function.

Curry and Uncurry: Converting Between Styles

curry: Convert from pairs to curried form

```
def curry {α β γ : Type} (f : (α × β) → γ) : α → β → γ :=  
  fun a b => f (a, b)  
  
-- Example: function taking pair  
def pairAdd (p : Nat × Nat) : Nat := p.1 + p.2  
  
-- Convert to curried form  
def curriedAdd := curry pairAdd  
  
#eval pairAdd (3, 4)      -- 7  
#eval curriedAdd 3 4      -- 7 (can partial apply now!)  
#eval (curry pairAdd) 3 4 -- 7 (inline)
```

Use case: Work with legacy code or APIs expecting pairs

Uncurry: Converting Curried to Pairs

uncurry: Convert from curried to pair form

```
def uncurry {α β γ : Type} (f : α → β → γ) : (α × β) → γ :=  
  fun (a, b) => f a b  
  
-- Example: curried function  
def add (a b : Nat) : Nat := a + b  
  
-- Convert to pair form  
def pairAdd := uncurry add  
  
#eval add 3 4           -- 7  
#eval pairAdd (3, 4)    -- 7  
#eval uncurry add (3, 4) -- 7 (inline)
```

Use cases: When you have pairs of data and want to apply a curried function

Note: $\text{uncurry } (\text{curry } f) = f$ and $\text{curry } (\text{uncurry } g) = g$ (isomorphism!)

Flip: Reverse Argument Order

flip: Swap the order of the first two arguments

```
def flip {α β γ : Type} (f : α → β → γ) : β → α → γ :=
  fun b a => f a b

-- Example: subtraction (order matters!)
def sub (a b : Nat) : Int := (a : Int) - (b : Int)

#eval sub 10 3          -- 7 (10 - 3)
#eval flipE sub 10 3    -- -7 (3 - 10, flipped!)

-- Useful for partial application
def subtractFrom10 := flip sub 10
#eval subtractFrom10 3  -- -7 (3 - 10)
#eval subtractFrom10 5  -- -5 (5 - 10)
```

Use cases: When you want to make partial application more convenient.

Flip: Why It's Useful

The Problem: Sometimes argument order is inconvenient (for us)

Example: append operation

- `append xs ys` appends `ys` to `xs`
- Want: `appendToXs = append xs` (partially applied)
- But often we want to append to a fixed list!
- Solution: `prependToYs = flip append ys`

Example: division operation

- `div a b` computes `a / b`
- Want: "divide something by 2"
- `divideBy2 = flip div 2`
- Now: `divideBy2 10 = 5`

Observation: Flip lets you partially apply the "wrong" argument!

Const: The Constant Function

const: Returns a function that always returns same value

```
def const {α β : Type} (a : α) : β → α :=  
  fun _ => a  
  
-- Ignores its argument, always returns a  
#eval (const 42) "hello"    -- 42  
#eval (const true) 100      -- true  
#eval (const "x") [1, 2, 3] -- "x"
```

Type signature: $\alpha \rightarrow \beta \rightarrow \alpha$

- Takes value of type α
- Returns function $\beta \rightarrow \alpha$
- That function ignores its argument
- Always returns the original α value

Const: Use Cases

1. Replace all elements in a list:

```
#eval map (const 0) [1, 2, 3, 4]      -- [0, 0, 0, 0]
#eval map (const "x") [1, 2, 3]      -- ["x", "x", "x"]

-- Create a function that replaces with a value
def replaceWith (value :  $\alpha$ ) : List  $\beta$   $\rightarrow$  List  $\alpha$  :=
  map (const value)

#eval replaceWith 7 ["a", "b", "c"]  -- [7, 7, 7]
```

2. Provide default values:

```
-- When you need a function but want constant output
def alwaysValid : String  $\rightarrow$  Bool := const true
def alwaysFalse : Nat  $\rightarrow$  Bool := const false

#eval filter alwaysValid ["a", "b"]  -- ["a", "b"]
#eval filter alwaysFalse [1, 2, 3]   -- []
```

Inductive Types: The Foundation

Recall from Week 2: Inductive types are the core building block in Lean

Main properties (and slogans):

- **"No Junk"**: only values from constructors exist
- **"No Confusion"**: different constructors $\neq$ different values
- **Structural induction**: pattern matching $\equiv$ proof by cases

Why important for FP:

- Exhaustive pattern matching (via compiler checks)
- Structural recursion (guaranteed termination)
- It is type-safe by construction
- We can compose data structures safely

Today: We'll see how these enable some very powerful functional patterns

Enumerated Types (Refresher)

Simplest inductive type: Finite list of elements (e.g. weekdays)

```
inductive Weekday where
  | sunday | monday | tuesday | wednesday
  | thursday | friday | saturday
  deriving Repr, BEq

-- Pattern match to define functions
def numberOfDay : Weekday → Nat
  | .sunday    => 1
  | .monday    => 2
  | .tuesday   => 3
  | .wednesday => 4
  | .thursday  => 5
  | .friday    => 6
  | .saturday  => 7

def isWeekend : Weekday → Bool
  | .saturday => true
  | .sunday  => true
  | _       => false -- "catch-all" pattern
```

Pattern Matching: Exhaustiveness Checking

Lean requires exhaustive patterns:

Without the catch-all pattern:

- You must handle every constructor
- The compiler checks if you have missed any
- Have to explicitly handle every case (which prevents bugs)

With catch-all (`_`):

- Handles "all other cases"
- Useful when most cases have same behavior
- But be careful – it might hide bugs if you add constructors later!

Best practice: Be explicit when possible, use catch-all when justified

The "No Junk, No Confusion" Principle

"No Junk": Only constructor-built values exist

```
-- For Weekday: only these 7 values exist
-- No "undefined", no "null", no special error values
def allWeekdays : List Weekday :=
  [.sunday, .monday, .tuesday, .wednesday,
   .thursday, .friday, .saturday]
```

"No Confusion": Different constructors build different values

```
-- Lean knows these are different
theorem monday_ne_tuesday : Weekday.monday ≠ Weekday.tuesday := by
  intro h
  cases h -- Contradiction! Different constructors

-- Can use in proofs and programs
def isSameDay (d1 d2 : Weekday) : Bool :=
  d1 == d2 -- Uses BEq derived from "no confusion"
```

Example: Color Mixing

```
inductive Color where
  | Red | Green | Blue
  deriving Repr, BEq

-- Complex pattern matching
def mixColors : Color → Color → Color
  | .Red, .Blue => .Green
  | .Blue, .Red => .Green
  | .Red, .Green => .Blue
  | .Green, .Red => .Blue
  | .Blue, .Green => .Red
  | .Green, .Blue => .Red
  | c, _ => c -- Same color or mixing with itself

#eval mixColors .Red .Blue -- Color.Green
#eval mixColors .Red .Red -- Color.Red
#eval mixColors .Green .Blue -- Color.Red
```

Pattern Matching: Define behavior explicitly for each case combination

Structures: Product Types

Structures: Group related values with named fields (like `struct` in C or Rust)

```
structure Point where
  x : Float
  y : Float
  deriving Repr

-- Create values
def origin : Point := { x := 0.0, y := 0.0 }
def p : Point := { x := 3.0, y := 4.0 }

-- Access fields
#eval origin.x -- 0.0
#eval p.y      -- 4.0

-- Pattern match on structure
def isOrigin : Point → Bool
| { x := 0.0, y := 0.0 } => true
| _ => false
```

Key: Structures are *immutable* – can't modify fields!

Operations on Structures

Let's consider a structure `Point` and define some functions on it.

```
structure Point where
  x : Float
  y : Float
  deriving Repr

def addPoints (p q : Point) : Point :=
  { x := p.x + q.x, y := p.y + q.y }

def scalePoint (k : Float) (p : Point) : Point :=
  { x := k * p.x, y := k * p.y }

def distance (p q : Point) : Float :=
  Float.sqrt ((p.x - q.x)^2 + (p.y - q.y)^2)

#eval addPoints { x := 1.0, y := 2.0 } { x := 3.0, y := 4.0 }
-- Result: { x := 4.0, y := 6.0 }

#eval scalePoint 2.0 { x := 3.0, y := 4.0 }
-- Result: { x := 6.0, y := 8.0 }
```

Notice the pattern: We implemented pure functions on immutable data.

Functional Update Syntax

Let's create some new functionality on this struct!

```
structure Point where
  x : Float
  y : Float
  deriving Repr

def p : Point := { x := 1.0, y := 2.0 }

-- Functional update: {struct with field := newValue}
def moveRight (p : Point) (dx : Float) : Point :=
  { p with x := p.x + dx }

def moveUp (p : Point) (dy : Float) : Point :=
  { p with y := p.y + dy }

#eval moveRight p 3.0    -- { x := 4.0, y := 2.0 }
#eval moveUp p 5.0      -- { x := 1.0, y := 7.0 }

-- Update multiple fields
def move (p : Point) (dx dy : Float) : Point :=
  { p with x := p.x + dx, y := p.y + dy }
```

Important observation: Original struct `p` is unchanged!

Why Immutability Matters

Immutable data structures:

Benefits:

- No accidental modifications (so safe to share between threads)
- Can reason about code locally
- History is preserved (time-travel debugging!)
- Easier to test (no hidden state changes)

Cost:

- Must copy data for updates
- More memory usage (but structural sharing helps!)
- Different mindset from imperative programming

Tradeoff: Safety and clarity vs. performance

- For most programs: Safety wins!
- For critical paths: Can use mutable structures carefully

Sum Types: "This OR That"

Recall sum types $A + B$: They contain a value of type A **OR** a value of type B

Constructors:

- `Sum.inl` : $A \rightarrow A + B$ (left injection)
- `Sum.inr` : $B \rightarrow A + B$ (right injection)

```
inductive Sum (α : Type) (β : Type) where
| inl : α → Sum α β -- "in left" - value of type α
| inr : β → Sum α β -- "in right" - value of type β

-- Example: String or Int
def value1 : Sum String Int := Sum.inl "hello"
def value2 : Sum String Int := Sum.inr 42

-- Must pattern match to use
def showSum : Sum String Int → String
| Sum.inl s => s!"Got string: {s}"
| Sum.inr n => s!"Got int: {n}"

#eval showSum value1 -- "Got string: hello"
#eval showSum value2 -- "Got int: 42"
```

Sum Types: Modeling Alternatives

Sum types model "either/or" situations:

Use cases:

- Result type: `Sum Error Success`
- Parsing: `Sum ParseError AST`
- User input: `Sum Cancel Submit`
- Multiple formats: `Sum JSON XML`
- Error handling: `Sum Exception Value`

Contrast with OOP inheritance hierarchies: fragile, implicit.

The compiler ensures you handle all alternatives!

Option Types: The "Billion Dollar Fix"

The Option type: Explicit handling of the absence of a value

```
inductive Option (α : Type) where
| none : Option α      -- No value present
| some : α → Option α  -- Value present

-- Safe list head
def head? {α : Type} : List α → Option α
| []      => none
| x :: _  => some x

#eval head? [1, 2, 3]      -- some 1
#eval head? ([] : List Nat) -- none

-- Type forces you to handle both cases!
def process (xs : List Nat) : Nat :=
  match head? xs with
  | none   => 0      -- Must handle empty case
  | some x => x + 1  -- Only here do we have value
```

No NullPointerException possible!

Working with the Option type

```
-- Get value or default
def getOrDefault {α : Type} (opt : Option α) (default : α) : α :=
  match opt with
  | none => default
  | some x => x

#eval getOrDefault (some 42) 0    -- 42
#eval getOrDefault none 0        -- 0

-- Map over Option
def mapOption {α β : Type} (f : α → β) : Option α → Option β
  | none => none
  | some x => some (f x)

#eval mapOption (· + 1) (some 5)  -- some 6
#eval mapOption (· + 1) none      -- none

-- Chain operations
def andThen {α β : Type} (opt : Option α) (f : α → Option β) : Option β :=
  match opt with
  | none => none
  | some x => f x
```

Option: Why It's Better Than Null

Problem with Null:

- `String s = maybeGetUser()` – is `s` null?
- Type doesn't say anything, so the check happens at runtime
- If we forget to check → `NullPointerException`

Solution with Option:

- `Option String` vs `String` – different types!
- Type tells you "might be absent"
- Can't use value without checking
- Forget to check → compile error (yay!)
- Null pointer errors literally impossible

This is "Tony Hoare's billion dollar fix"!

Recursive Types: Natural Numbers

Inductive types can be (and most of the time are...) recursive:

```
inductive Nat where
  | zero : Nat          -- Base case
  | succ : Nat → Nat    -- Recursive case

-- Representation:
-- 0 = zero
-- 1 = succ zero
-- 2 = succ (succ zero)
-- 3 = succ (succ (succ zero))

def add : Nat → Nat → Nat
  | n, Nat.zero    => n
  | n, Nat.succ m => Nat.succ (add n m)

#eval add 2 3 -- 5
```

Notice the pattern we are using: Defining operations by structural recursion

Recursive Types: Lists

```
inductive List (α : Type) where
| nil  : List α
| cons : α → List α → List α

-- Syntactic "sugar": [1, 2, 3] = cons 1 (cons 2 (cons 3 nil))

def length {α : Type} : List α → Nat
| []      => 0
| _ :: xs => 1 + length xs

def append {α : Type} : List α → List α → List α
| [],      ys => ys
| x :: xs, ys => x :: append xs ys

#eval length [1, 2, 3, 4]      -- 4
#eval append [1, 2] [3, 4, 5] -- [1, 2, 3, 4, 5]
```

Notice the pattern: Base case (nil) + recursive case (cons) → like induction!

Polymorphism in Inductive Types

Type parameters make structures "generic":

- `List  $\alpha$` works for any type α
- `Option  $\alpha$` can wrap any type α
- `Sum  $\alpha$   $\beta$` combines any two types α and β
- `BTree  $\alpha$` stores any type α in its nodes

Examples:

- `List Nat` – a list of numbers
- `List String` – a list of strings
- `List (List Nat)` – a list of lists
- `Option (List Nat)` – maybe a list
- `Sum String (List Nat)` – a string or a list

Write once, use for all types!

Deriving Instances

You can easily auto-generate useful functionality:

```
inductive Weekday where
  | sunday | monday | tuesday | wednesday
  | thursday | friday | saturday
deriving Repr, BEq, Ord, Inhabited

-- Repr: String representation
#eval Weekday.monday -- Weekday.monday

-- BEq: Boolean equality
#eval Weekday.monday == Weekday.tuesday -- false

-- Ord: Ordering (for sorting)
#eval compare Weekday.monday Weekday.friday -- Ordering.lt

-- Inhabited: Default value
#eval (default : Weekday) -- Weekday.sunday
```

Lean generates implementations automatically!

ZipWith: Combine Two Lists

ZipWith Pattern: Combine corresponding elements from two lists

```
def zipWith {α β γ : Type} (f : α → β → γ) :  
  List α → List β → List γ  
  | [], _ => []  
  | _, [] => []  
  | x :: xs, y :: ys => f x y :: zipWith f xs ys  
  
#eval zipWith (· + ·) [1, 2, 3] [4, 5, 6]  
-- [5, 7, 9] because (1+4, 2+5, 3+6)  
  
#eval zipWith (· * ·) [2, 3, 4] [5, 6, 7]  
-- [10, 18, 28] because (2*5, 3*6, 4*7)  
  
#eval zipWith (·, ·) [1, 2, 3] ["a", "b", "c"]  
-- [(1, "a"), (2, "b"), (3, "c")]
```

Stops at the shorter list of the two!

ZipWith: Use Cases

Common applications:

Vector operations:

- Add vectors: `zipWith (+) v1 v2`
- Dot product: `sum (zipWith (*) v1 v2)`

Data alignment:

- Merge two datasets: `zipWith combine data1 data2`
- Pair IDs with values: `zipWith (,) ids values`

Comparisons:

- Element-wise comparison: `zipWith (==) expected actual`
- Find differences: `filter (not . uncurry (==)) (zipWith (,) xs ys)`

General idea: Operating on aligned data from multiple sources

DropWhile: Skip Elements

DropWhile Pattern: Removes from the front of a list while a certain predicate holds

```
def dropWhile {α : Type} (p : α → Bool) : List α → List α
| [] => []
| x :: xs => if p x then dropWhile p xs
              else x :: xs

#eval dropWhile (· < 5) [1, 2, 3, 6, 4, 7]
-- [6, 4, 7] (stopped at 6)

#eval dropWhile (· % 2 == 0) [2, 4, 6, 1, 8]
-- [1, 8] (stopped at 1)

#eval dropWhile (· < 10) [1, 2, 3, 4]
-- [] (all dropped)

#eval dropWhile (· > 10) [1, 2, 3, 4]
-- [1, 2, 3, 4] (nothing dropped)
```

It is easy to see that it stops at the first element where the predicate is false!

DropWhile: Complementary Functions

Some worthwhile (pun intended) functions related to DropWhile:

```
-- takeWhile: opposite of dropWhile
def takeWhile {α : Type} (p : α → Bool) : List α → List α
| [] => []
| x :: xs => if p x then x :: takeWhile p xs
           else []

#eval takeWhile (· < 5) [1, 2, 3, 6, 4, 7]
-- [1, 2, 3] (before first ≥ 5)

-- drop: drop exactly n elements
def drop {α : Type} : Nat → List α → List α
| 0, xs => xs
| _, [] => []
| n+1, _ :: xs => drop n xs

#eval drop 2 [1, 2, 3, 4, 5]
-- [3, 4, 5]
```

Pattern: Different ways to remove elements from front

Partition: Split by Predicate

Partition Pattern: Split into (matching, non-matching) groups

```
def partition {α : Type} (p : α → Bool) :  
  List α → (List α × List α)  
| [] => ([], [])  
| x :: xs =>  
  let (matching, others) := partition p xs  
  if p x then (x :: matching, others)  
  else (matching, x :: others)  
  
#eval partition (· % 2 == 0) [1, 2, 3, 4, 5, 6]  
-- ([2, 4, 6], [1, 3, 5])  
  
#eval partition (· > 5) [1, 8, 3, 9, 2, 7]  
-- ([8, 9, 7], [1, 3, 2])
```

An important property of Partition:

No element of the original list is lost, but (without access to the original) order is!

Partition: Some Applications

Use cases: When you want one pass through list, but two different outputs.

Quicksort:

- Partition around pivot (`partition (< pivot) xs`)
- Recursively sort both partitions

Data filtering:

- Separate valid from invalid
- Process each group differently
- Keep both groups for analysis

User selection:

- Selected vs unselected items
- Process selected items
- Keep unselected for later

Interleave: Merge Alternating

Interleave Pattern: Alternates elements from two lists.

Note: The code below is incomplete, because it's left as an exercise for you in Programming Assignment 3 :)

```
def interleave {α : Type} : List α → List α → List α
| [], ys =>      -- Base case: first list empty
| xs, [] =>      -- Base case: second list empty
| x :: xs, y :: ys => -- Recursive: take from each, recurse

#eval interleave [1,3,5] [2,4,6]
-- [1, 2, 3, 4, 5, 6]

#eval interleave [1,2] [10,20,30,40]
-- [1, 10, 2, 20, 30, 40]

#eval interleave ["a", "b"] ["x", "y", "z"]
-- ["a", "x", "b", "y", "z"]
```

Use cases: When you want to merge two sorted sequences while preserving order

SplitAt: Split at Index

SplitAt Pattern: Split list at given position.

Note: The code below is incomplete, because it's left as an exercise for you in Programming Assignment 3 :)

```
def splitAt {α : Type} : Nat → List α → (List α × List α)
| 0, xs =>      -- Base case: split at 0
| _, [] =>      -- Base case: empty list
| n+1, x :: xs => -- Recursive case: split tail, add x to left part

#eval splitAt 2 [1,2,3,4,5]
-- ([1, 2], [3, 4, 5])

#eval splitAt 0 [1,2,3]
-- ([], [1, 2, 3])

#eval splitAt 10 [1,2,3]
-- ([1, 2, 3], [])
```

Property: `append (splitAt n xs).1 (splitAt n xs).2 = xs` In other words, concatenating results gives original list (order preserved!)

FindIndex: Locate Element

FinIndex Pattern: Find the position of the first match (for a predicate), but NOT the index.

```
def findIndexHelper {α : Type} (p : α → Bool) :  
  Nat → List α → Option Nat  
  | _, [] => none  
  | n, x :: xs =>  
    if p x then some n  
    else findIndexHelper p (n+1) xs  
  
def findIndex {α : Type} (p : α → Bool) : List α → Option Nat :=  
  findIndexHelper p []  
  
#eval findIndex (· > 5) [1, 3, 6, 2, 8]  
-- some 2 (found 6 at index 2)  
  
#eval findIndex (· > 10) [1, 3, 6, 2, 8]  
-- none (not found)
```

Helper pattern (aka Worker-Wrapper Pattern): Our function needs to “remember” how many elements it passed → second, helper function (accumulator) that updates the state by passing $n + 1$ to the next call.

FindIndex: Why Option?

Why return `Option Nat`?

Problem: Element might not exist

- Can't return `-1` (not a `Nat`)
- Can't return special "not found" value
- Could throw exception (but not FP style!)

Solution: `Option Nat`

- `some n` when found at index `n`
- `none` when not found
- Type system forces caller to handle both cases
- No special values, no exceptions!

This is the FP way!

GroupConsecutive: Group Adjacent Equals

GroupConsecutive Pattern: Groups together into a list consecutive equal elements.

Note: The code below is incomplete, because it's left as an exercise for you in Programming Assignment 3 :)

```
def groupConsecutive {α : Type} [BEq α] : List α → List (List α)
| [] => -- Base case: empty list
| x :: xs =>
  match xs with
  | [] => -- Single element: group of one
  | y :: ys =>
    if x == y then -- x equals y: add x to first group from recursion
    else -- x differs from y: start new group with x
-- Algorithm: Compare adjacent elements, build groups

#eval groupConsecutive [1,1,2,2,2,3,3]
-- [[1,1], [2,2,2], [3,3]]
```

The algorithm: Build groups by checking adjacent elements

Section 3

Binary Trees (Deep Dive)

Binary Trees: Definition

Recall: A binary tree is a recursive structure with at most two children

```
inductive BTree (α : Type) : Type where
| empty : BTree α
| node   : α → BTree α → BTree α → BTree α
deriving Repr

-- Example binary tree:
--      5
--     / \
--    3   7
--   /
--  1

def exampleTree : BTree Nat := -- Explicitly write out the binary tree using its constructors
  BTree.node 5
    (BTree.node 3
      (BTree.node 1 BTree.empty BTree.empty)
      BTree.empty)
    (BTree.node 7 BTree.empty BTree.empty)
```

Tree Size: Count All Nodes

Binary Tree Size Pattern: 1 (current node) + size of left + size of right

```
def size {α : Type} : BTree α → Nat
| BTree.empty => 0
| BTree.node _ l r => 1 + size l + size r

def tree1 : BTree Nat :=
  BTree.node 1 BTree.empty BTree.empty

def tree2 : BTree Nat :=
  BTree.node 2 tree1 tree1

#eval size (BTree.empty : BTree Nat) -- 0
#eval size tree1                      -- 1
#eval size tree2                      -- 3 (root + 2 children)
```

Time complexity: $O(n)$ – visits every node once

Tree Mirror: Swap Subtrees

Main property of Binary Tree Mirrors: `mirror (mirror t) = t` (involutive!)

```
def mirror {α : Type} : BTree α → BTree α
| BTree.empty => BTree.empty
| BTree.node a l r => BTree.node a (mirror r) (mirror l)

-- Original:      Mirror:
--      5          5
--     / \        / \
--    3   7       7   3
--   /       \   / \
--  1         1  1   1
--
#eval mirror exampleTree
```

Real-world use cases: Horizontal flip, RTL vs LTR displays, etc.

Tree Height: Maximum Depth

It is easy to see that the worst-case scenario will be an unbalanced tree.

```
def height {α : Type} : BTree α → Nat
| BTree.empty => 0
| BTree.node _ l r => 1 + Nat.max (height l) (height r)

#eval height (BTree.empty : BTree Nat) -- 0
#eval height tree1                      -- 1
#eval height tree2                      -- 2

-- Unbalanced tree (worst case):
--      1
--      |
--      2
--      |
--      3
def unbalanced : BTree Nat :=
  BTree.node 1 BTree.empty
    (BTree.node 2 BTree.empty
      (BTree.node 3 BTree.empty BTree.empty))

#eval height unbalanced -- 3
```

Observation: The BTree's height noticeably affects the performance of search operations!

Tree Height: Balanced vs Unbalanced

Height matters for performance:

Balanced tree (where height $\approx \log n$):

- The height grows slowly with the number of nodes
- Search, insert, delete operation all take $O(\log n)$ time
- Example: 1000 nodes $\rightarrow$ height 10

Unbalanced tree (where height $\approx n$):

- The height can equal the number of nodes
- Basically degrades to a linked list
- Search, insert, delete operations all take $O(n)$ time
- Example: 1000 nodes $\rightarrow$ height 1000

Self-balancing trees (AVL, Red-Black) maintain $O(\log n)$ height!

MapTree: Transform Values

MapTree Pattern: Essentially like the map function for lists, but for trees!

```
def mapTree {α β : Type} (f : α → β) : BTree α → BTree β
| BTree.empty => BTree.empty
| BTree.node a l r =>
  BTree.node (f a) (mapTree f l) (mapTree f r)

#eval mapTree (· + 1) tree1
-- node 2 empty empty

#eval mapTree (· * 2) tree2
-- node 4 (node 2 empty empty) (node 2 empty empty)

#eval mapTree toString exampleTree
-- Converts all values to strings
```

Observation: This preserves the tree's structure, but it transforms its values.

CountLeaves: Nodes Without Children

Definition: A *leaf* is a node where both subtrees are empty, i.e. it has no children.

CountLeaves Pattern: Special case for counting leaves, recurse (keep going) otherwise.

```
def countLeaves {α : Type} : BTree α → Nat
| BTree.empty => 0
| BTree.node _ BTree.empty BTree.empty => 1 -- Leaf!
| BTree.node _ l r => countLeaves l + countLeaves r

def leaf : BTree Nat :=
  BTree.node 1 BTree.empty BTree.empty

def branch : BTree Nat :=
  BTree.node 2 leaf leaf

#eval countLeaves (BTree.empty : BTree Nat) -- 0
#eval countLeaves leaf -- 1
#eval countLeaves branch -- 2
#eval countLeaves exampleTree -- 2 (nodes 1 and 7)
```

Trace Example: branch (Node 2 with two leaf children)

$\text{countLeaves branch} \rightarrow \text{countLeaves leaf} + \text{countLeaves leaf} \Rightarrow 1 + 1 = 2$

Contains: Search for Value

Contains Pattern: Search for a value in a binary tree.

```
def contains {α : Type} [BEq α] (x : α) : BTree α → Bool
| BTree.empty => false
| BTree.node a l r =>
  a == x || contains x l || contains x r

#eval contains 1 leaf      -- true
#eval contains 5 leaf      -- false
#eval contains 2 branch    -- true
#eval contains 1 branch    -- true (in children)
#eval contains 7 exampleTree -- true
#eval contains 4 exampleTree -- false
```

Time complexity: $O(n)$ worst case (without an ordering invariant, we may have to visit every node).

Binary search tree: If we enforce $\text{left} < \text{root} < \text{right}$, we can ignore half the tree at every step, reducing complexity to $O(\log n)$.

MaxElement: Find Maximum

MaxElement Pattern: Compare node with the max value of both subtrees.

```
def maxElement {α : Type} [Ord α] [Max α] : BTree α → Option α
| BTree.empty => none
| BTree.node a l r =>
  let maxL := maxElement l
  let maxR := maxElement r
  match maxL, maxR with
  | none, none => some a
  | some x, none => some (max a x)
  | none, some y => some (max a y)
  | some x, some y => some (max a (max x y))

#eval maxElement (BTree.empty : BTree Nat) -- none
#eval maxElement leaf                      -- some 1
#eval maxElement branch                    -- some 2
#eval maxElement exampleTree               -- some 7
```

Inorder Traversal

Order: Left subtree → Root → Right subtree

```
def inorder {α : Type} : BTree α → List α
| BTree.empty => []
| BTree.node a l r => inorder l ++ [a] ++ inorder r

-- Tree:
--      2
--     / \
--    1   3
def orderedTree : BTree Nat :=
  BTree.node 2
    (BTree.node 1 BTree.empty BTree.empty)
    (BTree.node 3 BTree.empty BTree.empty)

#eval inorder orderedTree -- [1, 2, 3]
#eval inorder exampleTree -- [1, 3, 5, 7]
```

Main Property: For a binary search tree, it returns a sorted list!

- Function mirrors tree's inductive structure, visiting each node exactly once ($O(n)$).
- By using ++, builds the sorted result from bottom up → clean, declarative.

Tree Traversals: The Three Orders

Three main traversal orders:

Inorder (left-root-right):

- For BST: gives sorted sequence
- This is used for: printing sorted values

Preorder (root-left-right):

- Process node before children
- This is used for: copying tree, expression evaluation

Postorder (left-right-root):

- Process node after children
- This is used for: deleting tree, postfix expressions

Different orders for different use cases!

Level-Order Traversal (Breadth-First)

Process nodes level by level:

```
def levelOrderHelper {α : Type} :  
  Nat → List (BTree α) → List (List α)  
  | 0, _ => []  
  | _, [] => []  
  | fuel+1, trees =>  
    let values := trees.filterMap (fun t =>  
      match t with  
      | BTree.empty => none  
      | BTree.node a _ _ => some a)  
    if values.isEmpty then []  
    else  
      let children := trees.flatMap (fun t =>  
        match t with  
        | BTree.empty => []  
        | BTree.node _ l r => [l, r])  
      values :: levelOrderHelper fuel children  
  
def levelOrder {α : Type} (t : BTree α) : List (List α) :=  
  levelOrderHelper 100 [t]
```

Level-Order: Example

```
-- Tree:
--       5
--      / \
--     3   7
--    /   \
--   1     9

#eval levelOrder exampleTree
-- [[5], [3, 7], [1]]

-- Each inner list is one level!
```

Practical use cases:

- Finding shortest path in tree
- Level-wise processing
- Pretty printing trees
- Serialization preserving structure

Level-Order Pattern: Queue of nodes to process (Breadth-First Search!)

Section 4

Pattern Matching (Deeper Dive)

Pattern Matching Expressions

Pattern Matching Syntax:

`match [the term] with | pattern => result`

```
-- Count elements satisfying predicate
def count {α : Type} (p : α → Bool) : List α → Nat
| []      => 0
| x :: xs =>
  match p x with
  | true  => 1 + count p xs
  | false => count p xs

#eval count (· > 5) [1, 8, 3, 9, 2, 7] -- 3

-- Multiple patterns
def describe (n : Nat) : String :=
  match n with
  | 0 => "zero"
  | 1 => "one"
  | 2 => "two"
  | _ => "many"
```

Pattern Matching on Structures

Let's go back to our `Point` structure and pattern match on it:

```
structure Point where
  x : Float
  y : Float

def isOrigin : Point → Bool
  if (p.x == 0.0) && (p.y == 0.0) then true else false
  -- Note: Lean has a hard time pattern matching on floats using '|'

#eval isOrigin {x := 0.0, y := 0.0} -- true
#eval isOrigin {x := 1.0, y := 0.0} -- false

-- Extract components
def describe : Point → String
  | {x := 0.0, y := 0.0} => "origin"
  | {x := x, y := 0.0} => s!"on x-axis at {x}"
  | {x := 0.0, y := y} => s!"on y-axis at {y}"
  | {x := x, y := y} => s!"at ({x}, {y})"

#eval describe {x := 3.0, y := 0.0}
-- "on x-axis at 3.000000"
```

Nested Pattern Matching

You can perform pattern matching on multiple structures:

```
-- Pattern match on multiple structures
def comparePoints : Point → Point → String
| {x := x1, y := y1}, {x := x2, y := y2} =>
  if x1 == x2 && y1 == y2 then "equal"
  else if x1 == x2 then "same x"
  else if y1 == y2 then "same y"
  else "different"

-- Match on Option in List
def getFirst {α : Type} : List (Option α) → Option α
| [] => none
| none :: xs => getFirst xs
| some x :: _ => some x

#eval getFirst [none, none, some 42, some 7] -- some 42
```

Notice the Pattern: We destructure nested data in one step!

Section 5

Mathematical Induction

Structural Induction on Lists

Principle: To prove $P[xs]$ for all lists, prove:

1. **Base case:** $P[[]]$
2. **Inductive step:** $\forall x xs, P[xs] \implies P[x :: xs]$

Why it works:

- All lists built from `[]` and `::`
- Base case handles empty list
- Inductive step handles cons
- Together: covers all lists!

This is pattern matching on steroids!

Example: Reverse is Involutive

Theorem: $\text{reverse} (\text{reverse } xs) = xs$

```
theorem reverse_reverse {α : Type} (xs : List α) :  
  reverse (reverse xs) = xs := by  
  induction xs with  
  | nil =>  
    rfl -- Base: reverse [] = []  
  | cons x xs ih =>  
    -- Inductive: assume reverse (reverse xs) = xs  
    -- Show: reverse (reverse (x :: xs)) = x :: xs  
    simp [reverse]  
    rw [ih] -- Use induction hypothesis  
  
-- This proof works because lists are inductive!
```

Pattern: Prove base case, use IH in inductive case

Structural Induction on Trees

Principle: To prove $P[t]$ for all trees, prove:

1. **Base case:** $P[\text{empty}]$
2. **Inductive step:** $\forall a \ l \ r, P[l] \implies P[r] \implies P[\text{node } a \ l \ r]$

Why it works:

- All trees built from empty and node
- Base case handles empty
- Inductive step: assume true for subtrees
- Prove true for node with those subtrees

Two induction hypotheses (one per subtree)!

Example: Mirror is Involutive

Theorem: $\text{mirror} (\text{mirror } t) = t$

```
theorem mirror_mirror {α : Type} (t : BTree α) :  
  mirror (mirror t) = t := by  
  induction t with  
  | empty =>  
    rfl -- Base: mirror empty = empty  
  | node a l r ih_l ih_r =>  
    -- Inductive: assume mirror (mirror l) = l  
    --               and mirror (mirror r) = r  
    -- Show: mirror (mirror (node a l r)) = node a l r  
    simp [mirror]  
    rw [ih_l, ih_r] -- Use both IHs!  
  
-- Two IHs because two recursive calls in definition!
```

Section 6

Summary

Summary

Functional Programming Core:

- Pure functions: deterministic, no side effects
- First-class functions: pass, return, store
- Higher-order functions: map, filter, fold, compose
- Currying and partial application

Data Structures:

- Inductive types: "No Junk", "No Confusion"
- Structures: immutable records
- Sum types and Option: explicit alternatives
- Lists and trees: recursive structures

Techniques:

- Pattern matching: exhaustive, safe
- Structural recursion: guaranteed termination
- Mathematical induction: prove correctness

Section 7

Assignments & Next Steps

This Week's Assignments

Readings (see the course website)

- Theorem Proving in Lean 4 (Chapter 4)
- Functional Programming in Lean 4 (Chapters 1-2-3 + Interlude 1)
- The Hitchhiker's Guide to Logical Verification (Chapter 5)

"Hand-in" Assignments (see the course website)

- PROOF101 Quiz 3 (due next time)
- Programming Assignment 3: Functional Programming (due next time)

Assignment covers: All concepts from today + Week 2

Questions & Discussion

Questions?

Join our community:

Zulip: Link on website

WhatsApp: Link on website

Website: <https://danieldia.me/proof101/>

Email: dmd13@mail.aub.edu



*"OOP makes code understandable
by encapsulating moving parts. FP
does so by minimizing moving
parts."*

— Michael Feathers

Week 2 of 11

PROOF101 Week 2 Functional Programming

AUB Math Society
Spring 2027

Functional Programming

Daniel Dia & Guest Lecturers

American University of Beirut (AUB)

<https://danieldia.me/proof101/>