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

— Michael Feathers

Google Developer Groups
On Campus • American University of Beirut

MATHEMATICS SOCIETY
AMERICAN UNIVERSITY OF BEIRUT

PROOF101: Formal Verification & Proof Assistants

Google Developer Groups @ AUB
& AUB Math Society
Spring 2026

Week 3 of 10

Functional Programming

Daniel Dia & Guest Lecturers

https://danieldia-dev.github.io/proofs/

Section 1

Historical Exposition: How did we get here?

Section 1

Historical Exposition: How did we get here?

Subsection 1.1

The Great Divergence (1936): Logic vs. Machines

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.

Key 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

Impact:

- 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.

Key Ideas:

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

Impact:

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

Section 1

Historical Exposition: How did we get here?

Subsection 1.2

The Crisis of Complexity (1960s – 1970s)

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"*

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

Section 1

Historical Exposition: How did we get here?

Subsection 1.3

The Object-Oriented Dream & The "Billion Dollar Mistake" (1970s – 1990s)

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:

- 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

What went wrong:

- **Mutable state everywhere** - objects became bags of 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 Problem:

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

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."*

The Functional Solution: Option/Maybe types - explicitly handle absence!

Section 1

Historical Exposition: How did we get here?

Subsection 1.4

The Concurrency Wall (2005): The Return to Church

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*.

The Problem with Shared Mutable State:

- **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

Section 2

The Theory of Functional Programming

Subsection 2.1

Purity and Side Effects

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 key 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]  (same!)

// IMPURE: Mutates array, different results
xs.splice(0, 3); // [1, 2, 3]
xs.splice(0, 3); // [4, 5]     (different!)
```

Why Purity Matters

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
- No mocking complex dependencies

3. Composable

- Can combine pure functions safely
- Order doesn't matter (mathematically)
- 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)

Result: Functions become like mathematical functions - predictable, testable, 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!
checkAge(20);  // true (same input, different output!)
```

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)
```

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

The Benefits of Purity: Cacheable (Memoization)

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 4²..." → 16
expensiveSquare(4); // → 16 (from cache, no logging!)
```

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

The Benefits of Purity: Portable & Testable

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)

The Benefits of Purity: Reasonable (Equational Reasoning)

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`
- Can reason algebraically about code!

Equational reasoning:

- Substitute "equals for equals" (like algebra)
- Refactor with confidence
- Compiler can optimize automatically
- Humans can understand code more easily

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

The Benefits of Purity: Parallel (Automatic Parallelization)

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 pure `f`
- Each `f(i)` is independent
- Can compute all in parallel with zero coordination
- Guaranteed to give same result as sequential execution

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

Section 2

The Theory of Functional Programming

Subsection 2.2

First-Class Functions

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:

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

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

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

Why First-Class Functions Matter

Enables abstraction over computation patterns:

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 is the foundation of functional programming!

First-Class Functions in Lean: Storing in Variables

Functions are values - can be stored in variables:

```
-- Store function in a variable
def double : Nat → Nat := fun x => x * 2

-- Use it like any other value
#eval double 5  -- 10

-- Can create multiple "copies"
def myDouble := double
def alsoDouble := double

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

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

First-Class Functions: Passing to Functions

Functions can take other functions as arguments:

```
-- Takes a function and applies it twice
def twice (f : α → α) (x : α) : α :=
  f (f x)

#eval twice (· + 1) 5     -- 7  (5 + 1 + 1)
#eval twice (· * 2) 3     -- 12 (3 * 2 * 2)

-- Apply function n times
def applyN (f : α → α) : Nat → α → α
  | 0, x => x
  | n+1, x => f (applyN f n x)

#eval applyN (· + 1) 5 0    -- 5 (add 1 five times to 0)
#eval applyN (· * 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:

```
-- 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”

First-Class Functions: In Data Structures

Functions can be stored in lists, tuples, etc:

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

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

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

-- 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 strategy pattern without OOP!

Section 2

The Theory of Functional Programming

Subsection 2.3

Higher-Order Functions

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 we've seen:

- `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 it 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
- Lots of duplicated loop logic!

With `map` (higher-order function):

- Write loop logic once in `map`
- `map double xs`
- `map square xs`
- `map increment xs`
- Abstract the pattern: "apply function to each element"

Higher-order functions let us separate "what" from "how"!

Map: Transform Every Element

Pattern: Apply a transformation to every element

```
def map (f : α → β) : List α → List β
  | []      => []
  | x :: xs => f x :: map f xs

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

Type: `(α → β) → List α → List β`

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

Map: Why It Matters

Declarative vs Imperative:

Imperative (how to do it):

- Create empty result list
- Loop through input list
- For each element, apply function
- Append to result list
- Return result list

Functional (what to do):

- `map f xs`
- Clear, concise, declarative
- The "how" is hidden in `map`

Benefits: Less code, clearer intent, fewer bugs, easier to parallelize!

Filter: Select Elements

Pattern: Keep only elements that satisfy a condition

```
def filter (p : α → Bool) : List α → List α
  | []      => []
  | 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: `(α → Bool) → List α → List α`

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

Filter: Common Use Cases

Filter is everywhere in real code:

- **Data cleaning**: Remove null/invalid values
- **Search**: Find items matching criteria
- **Validation**: Keep only valid inputs
- **Filtering API results**: Get only what you need
- **Permission checks**: Show only authorized items

Combines well with map:

- Filter, then transform: `map f (filter p xs)`
- Transform, then filter: `filter p (map f xs)`
- Called ”method chaining” in some languages

Fold: Reduce to Single Value

Pattern: Combine all elements using a binary operation

```
def foldr (f : α → β → β) (init : β) : List α → β
  | []      => init
  | x :: xs => f x (foldr f init xs)

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

Type: `(α → β → β) → β → List α → β`

- Combining function: `α → β → β`
- Initial/default value: `β`
- Input list: `List α`
- 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 :: []))` → `1 + (2 + (3 + 0))`
- Result: 6

Fold: The Universal List Function

Many list operations are (secretly) just folds!

- **sum**: `foldr (+) 0`
- **product**: `foldr (*) 1`
- **length**: `foldr (λ_ acc => acc + 1) 0`
- **reverse**: `foldl (λ acc x => x :: acc) []`
- **map f**: `foldr (λx acc => f x :: acc) []`
- **filter p**: `foldr (λx 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 : α → β → β) (init : β) : List α → β
  | []      => init
  | x :: xs => f x (foldr f init xs)

-- Left fold: processes left to right
def foldl (f : β → α → β) (init : β) : List α → β
  | []      => init
  | x :: xs => foldl f (f init x) xs
```

Key difference:

- **foldr**: `f 1 (f 2 (f 3 z))`, i.e. right-associative
- **foldl**: `f (f (f z 1) 2) 3`, i.e. left-associative
- **foldl** is tail-recursive (more efficient!)

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. Process innermost first: `3 - 10 = -7`
2. Then: `2 - (-7) = 9`
3. Finally: `1 - 9 = -8`

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. Start with accumulator: `10`
2. Process left to right: `10 - 1 = 9`
3. Then: `9 - 2 = 7`
4. Finally: `7 - 3 = 4`

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 (efficient!)

For commutative operations (+ , *): Doesn't matter mathematically, but foldl is more efficient!

Function Composition

Pattern: Chain functions together

```
def compose (f : β → γ) (g : α → β) : α → γ :=
  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: $(\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$

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

Composition: Building Pipelines

Composition lets you build transformation pipelines:

Without composition:

- `f(g(h(x)))` - hard to read (inside-out)
- Have to trace execution backwards
- Parentheses get unwieldy with many functions

With composition:

- `(f ∘ g ∘ h)(x)` - reads naturally (right-to-left)
- Or define: `pipeline = f ∘ g ∘ h`, then use: `pipeline(x)`
- Can name intermediate transformations
- Reuse composed functions

Composition is associative: `(f ∘ g) ∘ h = f ∘ (g ∘ h)`

Composition Example: Data Processing Pipeline

```
-- Individual transformations
def trim (s : String) : String := s.trim
def toLower (s : String) : String := s.toLower
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"]
```

Pattern: Build complex transformations from simple, reusable pieces!

Section 2

The Theory of Functional Programming

Subsection 2.4

Currying and Partial Application

What is Currying?

Currying: Transform function taking multiple arguments into chain of functions each taking one argument

Transform:

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

Named after Haskell Curry (mathematician, 1900-1982)

- Though actually invented by Gottlob Frege and Moses Schönfinkel
- 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

All these are 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  -- Nat → Nat → Nat
#check add2  -- Nat → Nat → Nat
#check add3  -- Nat → Nat → Nat
```

Key insight: `Nat → Nat → Nat` means `Nat → (Nat → Nat)`

- Function taking `Nat`
- Returning function taking `Nat`
- Which returns `Nat`

Partial Application

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

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

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

-- Partially applied (one argument provided)
def add3 := add 3
#check add3  -- Nat → Nat (it's a function!)

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

Pattern: Fix some parameters, get specialized function

Partial Application: Creating Specialized Functions

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

-- 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 patterns using partial application:

Configuration functions:

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

Validators:

- `isLongerThan min = (λs => s.length > min)`
- `filter (isLongerThan 5) strings`

Comparators:

- `isGreaterThan x = (λy => y > x)`
- `filter (isGreaterThan 10) numbers`

Pattern: 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 case: When you have pairs of data and want to apply curried function

Note: `uncurry (curry f) = f` and `curry (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 flip sub 10 3    -- -7 (3 - 10, flipped!)

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

Use case: Make partial application more convenient!

Flip: Why It's Useful

Problem: Sometimes argument order is inconvenient

Example: append

- `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

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

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

Const: The Constant Function

const: Returns 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: $\alpha \to \beta \to \alpha$

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

Const: Use Cases

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 : α) : List β → List α :=
  map (const value)

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

Provide default values:

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

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

Section 2

The Theory of Functional Programming

Subsection 2.5

Inductive Types (Deep Dive)

Inductive Types: The Foundation

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

Key properties:

- **No junk**: Only values from constructors exist
- **No confusion**: Different constructors ≠ different values
- **Structural induction**: Pattern match = proof by cases

Why important for functional programming:

- Exhaustive pattern matching (compiler checks!)
- Structural recursion (guaranteed termination)
- Type-safe by construction
- Compose data structures safely

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

Enumerated Types (Refresher)

Simplest inductive type: Finite list of elements

```
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 catch-all:

- Must handle every constructor
- Compiler checks you haven't missed any
- Prevents bugs from unhandled cases

With catch-all (_):

- Handles "all other cases"
- Useful when most cases have same behavior
- But be careful - 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: Define behavior explicitly for each case combination

Structures: Product Types

Structures: Group related values with named fields

```
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

```
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 }
-- { x := 4.0, y := 6.0 }

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

Pattern: Pure functions on immutable data!

Functional Update Syntax

Create new struct with some fields changed:

```
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 }
```

Key: Original struct p is unchanged!

Why Immutability Matters

Immutable data structures:

Benefits:

- No accidental modifications
- 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"

Sum types: Value is one type OR another

```
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"
```

Type system forces you to handle both cases!

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:

- OOP: Inheritance hierarchy (fragile, implicit)
- FP: Sum types (explicit, exhaustive, safe)

Compiler ensures you handle all alternatives!

Option Types: The "Billion Dollar Fix"

Option type: Explicit absence of 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 Option

```
-- 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 - must check at runtime
- Forget to check → `NullPointerException`
- Costs billions in bugs and crashes

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 (safe!)
- Null pointer errors literally impossible

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

Recursive Types: Natural Numbers

Inductive types can be 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
```

Pattern: Define operations by structural recursion

Recursive Types: Lists

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

-- 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]
```

Pattern: Base case (nil) + recursive case (cons)

Polymorphism in Inductive Types

Type parameters make structures generic:

- `List α` works for any type α
- `Option α` can wrap any type
- `Sum α β` combines any two types
- `BTree α` stores any type in nodes

Examples:

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

Write once, use for all types!

Deriving Instances

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!

Section 2

The Theory of Functional Programming

Subsection 2.6

List Operations (Deep Dive)

ZipWith: Combine Two Lists

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] (1+4, 2+5, 3+6)

#eval zipWith (· * ·) [2, 3, 4] [5, 6, 7]
-- [10, 18, 28] (2*5, 3*6, 4*7)

#eval zipWith (·, ·) [1, 2, 3] ["a", "b", "c"]
-- [(1, "a"), (2, "b"), (3, "c")]
```

Stops at shorter list!

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)`

Pattern: Operate on aligned data from multiple sources

DropWhile: Skip Elements

Pattern: Remove from front while 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)
```

Key: Stops at first element where predicate is false!

DropWhile: Complementary Functions

Related functions:

```
-- 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

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

```
def partition {α : Type} (p : α → Bool) :
  List α → (List α × List α)
  | [] => ([], [])
  | x :: xs =>
    let (matches, others) := partition p xs
    if p x then (x :: matches, others)
    else (matches, 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])
```

Property: Concatenating results gives original list (order preserved!)

Partition: Applications

Use cases:

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

Pattern: One pass through list, two outputs!

Interleave: Merge Alternating

Pattern: Alternate elements from two lists

```
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 case: Merge two sorted sequences while preserving order

SplitAt: Split at Index

Pattern: Split list at given position

```
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`

FindIndex: Locate Element

Pattern: Find position of first match

```
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 0

#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: Track index with accumulator!

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

Pattern: Group consecutive equal elements

```
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]]
```

Algorithm: Build groups by checking adjacent elements

Section 3

Binary Trees (Deep Dive)

Binary Trees: Definition

Recall: 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 tree:
--       5
--      / \
--     3   7
--    /
--   1
def exampleTree : BTree Nat :=
  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

```
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)
```

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

Time complexity: O(n) - visits every node once

Tree Mirror: Swap Subtrees

```
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

#eval mirror exampleTree
```

Property: `mirror (mirror t) = t` (involutive!)

Use case: Horizontal flip, RTL vs LTR display

Tree Height: Maximum Depth

```
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
```

Height affects performance of search operations!

Tree Height: Balanced vs Unbalanced

Height matters for performance:

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

- Height grows slowly with number of nodes
- Search, insert, delete: O(log n)
- Example: 1000 nodes → height 10

Unbalanced tree (height $\approx n$):

- Height can equal number of nodes
- Degrades to linked list
- Search, insert, delete: O(n)
- Example: 1000 nodes → height 1000

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

MapTree: Transform Values

```
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
```

Like map for lists, but for trees!

Preserves structure, transforms values

CountLeaves: Nodes Without Children

Leaf node: No children (both empty)

```
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)
```

Pattern: Special case for leaves, recurse otherwise

Contains: Search for Value

```
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 (must check all nodes)

Better: Binary search tree can do O(log n)!

MaxElement: Find Maximum

```
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
```

Pattern: Compare node with max of both subtrees

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]
```

Property: For binary search tree, returns sorted list!

Tree Traversals: The Three Orders

Three main traversal orders:

Inorder (left-root-right):

- For BST: gives sorted sequence
- Used for: printing sorted values

Preorder (root-left-right):

- Process node before children
- Used for: copying tree, expression evaluation

Postorder (left-right-root):

- Process node after children
- 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!
```

Use cases:

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

Pattern: Queue of nodes to process (BFS!)

Section 4

Pattern Matching (Deep Dive)

Pattern Matching Expressions

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

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

def isOrigin : Point → Bool
  | {x := 0.0, y := 0.0} => true
  | _ => false

#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

```
-- 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
```

Pattern: 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: `reverse (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[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: `mirror (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

What We've Learned

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 inductive types

Questions & Discussion

Questions?

Join our community:

Discord: https://discord.gg/ZNGE8Xgd

Website: https://danieldia-dev.github.io/proofs/

Email: dmd13@mail.aub.edu

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

— Michael Feathers

PROOF101: Formal Verification & Proof Assistants

Google Developer Groups @ AUB
& AUB Math Society
Spring 2026

Week 3 of 10

Functional Programming

Daniel Dia & Guest Lecturers

`https://danieldia-dev.github.io/proofs/`