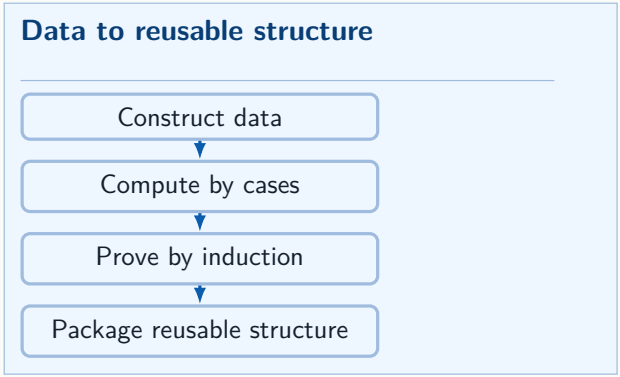
Inductive Types and Classes

Constructors, recursion, induction, structures, and type classes



July 16, Thursday

Inductive declarations, recursion, structures, type classes, deriving, diamonds, and expression trees.

Reading, Computing, Proving

Read declarations

Understand `inductive`, `structure`, and `class`.

Prove recursively

Connect recursion with induction and proof states.

Predict cases

Use constructors to predict pattern matching and proof branches.

Use type classes

Understand `instance` search, `deriving`, and notation.

Main fact

The shape of data determines how programs and proofs must be organized.

Modeling Choices and Classes

Use common types

Model with `Option`, `List`, `Fin`, `Multiset`, and `Finset`.

Know standard classes

Read `Repr`, `BEq`, `DecidableEq`, and `Inhabited`.

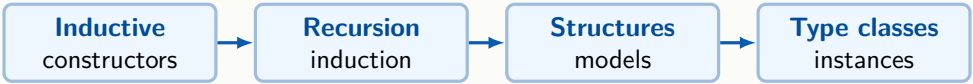
Recognize quotients

Read representatives modulo an equivalence relation.

Read math assumptions

Recognize `[Group G]`, `[Ring R]`, and `[Field K]`.

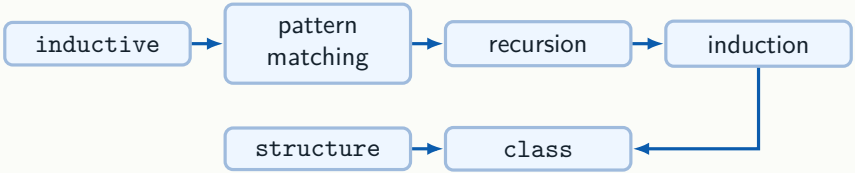
From Data to Reusable Structure



First: data constructors determine the legal cases.

Then: structures and classes make reusable mathematical APIs.

From Constructors to Search



Data shape question

How does the shape of data control the shape of programs, proofs, and reusable structure?

Why Inductive Types?

`Bool`

`true`, `false`

`Option alpha`

`none`, `some a`

`Nat`

`zero`, `succ`

`List alpha`

`[]`, `x :: xs`

Constructors determine behavior

If we know how values are built, we know how to compute with them and how to prove things about them.

A First Example

```
inductive Weekday where
  | mon | tue | wed | thu | fri | sat | sun
  deriving Repr, BEq, DecidableEq
```

- `Weekday` is a new type.
- The constructors are the only ways to build values of this type.
- `deriving` asks Lean to generate useful standard behavior.

How to Read an Inductive Declaration

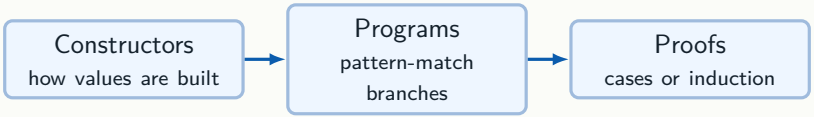
Ask four questions:

1. What is the new type?
2. What are the constructors?
3. Which constructors carry data?
4. Which constructors are recursive?

Reading rule

Constructor shape predicts program shape and proof shape.

Constructor Shape Controls Three Things



Finite constructors

Use case analysis: one branch per constructor.

Recursive constructors

Use recursion for programs and induction for proofs.

Weekday Constructors

```
inductive Weekday where
  | mon | tue | wed | thu | fri | sat | sun
```

- Every `Weekday` is one of these seven constructors.
- A function out of `Weekday` should account for these seven cases.
- A proof about all weekdays can also proceed by these seven cases.

Live Lean: `codeD2/D2_Inductive.lean`

Constructors Are Functions

```
inductive MyNat where
  | zero : MyNat
  | succ : MyNat -> MyNat
```

- `zero` is already a natural number.
- `succ` needs a smaller natural number before it produces a new one.
- Use `#check` to inspect constructor types.

Parameters vs. Constructor Arguments

```
inductive MyOption (alpha : Type) where
  | none : MyOption alpha
  | some : alpha -> MyOption alpha
```

- `alpha` is a parameter of the whole family of types.
- `some` stores one particular value of type `alpha`.
- Examples: `MyOption Nat`, `MyOption String`.

Recursive Constructors

```
inductive MyNat where
  | zero : MyNat
  | succ : MyNat -> MyNat

inductive MyList (alpha : Type) where
  | nil : MyList alpha
  | cons : alpha -> MyList alpha -> MyList alpha
```

- Recursive constructors mention the type being defined.
- The recursive argument is the smaller object.
- This is why structural recursion and induction follow the same shape.

Recursive Definition of Addition

```
def MyNat.add : MyNat -> MyNat -> MyNat
  | m, zero => m
  | m, succ n => succ (MyNat.add m n)
```

- The definition recurses on the second argument.
- The two equations follow the two constructors of `MyNat`.
- This is the same pattern as recursive definitions in mathematics.

Trees: Two Recursive Subobjects

```
inductive Tree (alpha : Type) where
  | leaf : alpha -> Tree alpha
  | node : Tree alpha -> Tree alpha -> Tree alpha
```

Remark

This teaching tree is nonempty and full: there is no empty constructor, and every internal node has exactly two children. It need not be balanced or perfect.

Mathlib's binary tree is different:

```
-- Mathlib.Data.Tree.Basic
inductive BinaryTree (alpha : Type) where
  | nil  : BinaryTree alpha
  | node : alpha -> BinaryTree alpha -> BinaryTree alpha -> BinaryTree alpha
```

Inductive Declarations

- An inductive type is specified by its constructors.
- Constructors may be plain values, functions with arguments, or recursive constructors.
- Type parameters describe a family of types; constructor arguments describe data stored in values.
- Natural numbers already show the bridge from constructors to recursive mathematical definitions.
- Constructor shape determines pattern matching, recursion, `cases`, and `induction`.

Recursion and Induction

$$\text{Data shape} \longrightarrow \text{program shape} \longrightarrow \text{proof shape}$$

We now use constructors in two ways:

- pattern matching to inspect data;
- recursion to compute over recursive data.

Pattern Matching

```
def isWeekend : Weekday -> Bool
  | Weekday.sat => true
  | Weekday.sun => true
  | _ => false
```

- Pattern matching eliminates an inductive value.
- Each branch handles one possible constructor shape.
- The wildcard pattern _ handles remaining cases.

Live Lean: codeD2/D2_PatternMatching.lean

Matching on Option

```
def getWithDefault {alpha : Type}
    (default : alpha) : Option alpha -> alpha
  | none => default
  | some a => a
```

- The `none` branch has no stored value.
- The `some` branch exposes the stored value.
- The return type is the same in both branches.

Recursive Definitions on Natural Numbers

```
def add (m : Nat) : Nat -> Nat
  | 0 => m
  | n + 1 => add m n + 1

def mul (m : Nat) : Nat -> Nat
  | 0 => 0
  | n + 1 => mul m n + m
```

- The recursion is on the second argument.
- The equations are ordinary recursive mathematical definitions.
- Powers can be defined in the same style.

Recursive Definitions on Lists

```
def myLength {alpha : Type} : List alpha -> Nat
  | [] => 0
  | _ :: xs => myLength xs + 1

def myMap {alpha beta : Type} (f : alpha -> beta) :
    List alpha -> List beta
  | [] => []
  | x :: xs => f x :: myMap f xs
```

- Both definitions branch on `[]` and `x :: xs`.
- The recursive call is made on `xs`, a smaller list.
- `map` preserves the list shape while changing stored data.

From Recursion to Induction

Data	Program	Proof
base constructor	base branch	base case
recursive constructor	recursive call	induction hypothesis

Reading rule

Programs and proofs follow the same shape as the data.

The Two Recursion Questions

Program question

If the input is a constructor, what output should the function return?

Proof question

If the value was built by a constructor, what must be proved in that case?

For recursive constructors, Lean gives either a recursive call or an induction hypothesis for the smaller subobject.

Proof by Cases

```
example (b : Bool) : b = true \/ b = false := by
  cases b <;> simp
```

- `cases` splits a value according to its constructors.
- `<;>` applies the following tactic to *all* remaining goals at once.
- `cases b <;> simp` means: split into `true` and `false`, then close both with `simp`.
- It is best for small or non-recursive inductive types.

Live Lean: `codeD2/D2_InductionProofs.lean`

Induction on Lists

```
theorem append_nil_right {alpha : Type} (xs : List alpha) :
    xs ++ [] = xs := by
  induction xs with
  | nil =>
      rfl
  | cons x xs ih =>
      rw [ih]
```

- The base case is the empty list.
- The step case is `x :: xs`.
- The induction hypothesis is the theorem already known for `xs`.

Induction on Addition

```
theorem zero_add (n : Nat) : add 0 n = n := by
  induction n with
  | zero =>
      rfl
  | succ n ih =>
      calc
        add 0 (n + 1) = add 0 n + 1 := rfl
        _ = n + 1 := by rw [ih]
```

- The base case proves the statement for `0`.
- The step case uses the induction hypothesis for `n`.
- `calc` chains equalities step by step; each `_` refers to the right-hand side of the previous line.
- This is the familiar proof of a recursive equation by induction.

The Proof State After Induction

In an induction step, focus on:

- the variables introduced by the constructor;
- the smaller recursive object;
- the induction hypothesis;
- the goal after simplification.

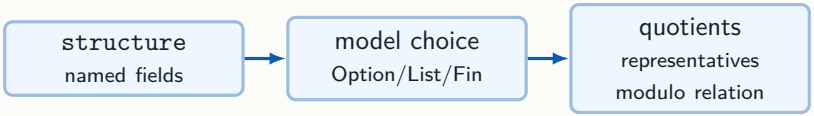
The local context is not decoration; it tells us exactly what we are allowed to use.

Proofs Follow Constructors

- Pattern matching uses the constructors of an inductive type as branches.
- Recursive definitions such as addition, multiplication, and powers follow recursive constructor arguments.
- `cases` is proof by constructor cases.
- `induction` proves recursive equations by following the same constructor shape.
- A good proof often begins by reading the local context carefully.

20-minute break

Structures and Models



Reading rule

Inductive types describe how values are built. Structures describe what fields a mathematical object carries.

Structures as Records

```
structure Vector2 where
  x : Int
  y : Int
```

- A structure bundles named fields.
- Values can be constructed with record syntax.
- Fields can be accessed with dot notation.

Live Lean: `codeD2/D2_Structure.lean`

Operations on a Structure

```
def Vector2.add (u v : Vector2) : Vector2 :=
  { x := u.x + v.x, y := u.y + v.y }

def Vector2.dot (u v : Vector2) : Int :=
  u.x * v.x + u.y * v.y
```

- `u.x` and `u.y` access coordinates.
- Record syntax constructs new vectors from field values.
- This mirrors ordinary coordinate definitions in linear algebra.

Bundling Data with a Property

```
structure PointOnDiagonal where
  x : Int
  y : Int
  onDiagonal : x = y
```

- Structures can contain data fields and proof fields.
- The field `onDiagonal` records a mathematical property.
- This pattern is common in formalized mathematics.

Inductive vs. Structure

Tool	Typical use	Shape
`inductive`	alternatives, recursive data	one or more constructors
`structure`	bundled fields	one constructor with named fields

Many mathematical objects in Lean are represented as structures.

Common Types as Modeling Choices

Type	Informal meaning	Typical use
`Option alpha`	maybe a value	partial lookup
`List alpha`	ordered finite data	sequences, recursion
`Fin n`	a natural number below `n`	finite indices
`Multiset alpha`	unordered, with multiplicity	combinatorics
`Finset alpha`	unordered, no duplicates	finite sets and sums

Live Lean: `codeD2/D2_CommonTypes.lean`

Option: Representing Possible Failure

```
def safeHead {alpha : Type} : List alpha -> Option alpha
  | [] => none
  | x :: _ => some x
```

- The type tells us that lookup may fail.
- Any consumer must handle both `none` and `some`.

Option: Supplying a Default

```
def getWithDefault {alpha : Type}
    (default : alpha) : Option alpha -> alpha
  | none => default
  | some x => x

#eval getWithDefault 0 (some 37)
#eval getWithDefault 0 (none : Option Nat)
```

- The caller explicitly provides the fallback value.
- The type `alpha` may be arbitrary.
- Without extra information, Lean cannot invent an `alpha`.

Option with an Inhabited Default

```
#check Inhabited
#check default
#synth Inhabited Nat

def getOrDefault {alpha : Type} [Inhabited alpha] :
    Option alpha -> alpha
  | none => default
  | some x => x

#eval getOrDefault (none : Option Nat)
#eval getOrDefault (none : Option Bool)
#eval getOrDefault (none : Option (List Nat))
```

- `[Inhabited alpha]` asks Lean to find a default.
- The square brackets mark a type class parameter.
- This is the same search mechanism used later for algebraic structure.

What Inhabited Does Not Mean

Reading rule

`[Inhabited alpha]` means: Lean, please find an inhabitant of `alpha`.

- It does not mean that the default is mathematically unique.
- It does not mean that the default is the best modeling choice.
- It only means that a value can be found automatically when requested.

List: Ordered Finite Data

For mathematicians, `List alpha` is often best read as a finite sequence.

Common operations:

- `map`: transform each element;
- `filter`: keep elements satisfying a test;
- `foldl`, `foldr`: accumulate values;
- `length`, append, membership.

Order matters, and repetitions are allowed.

Fin: Safe Bounded Indexing

```
#check Fin 3
#check (0 : Fin 3)
#check (2 : Fin 3)
```

- An element of `Fin n` is a natural number together with a proof it is less than `n`.
- Read it as the formal index set $\{0, \ldots, n-1\}$.
- Invalid indices are rejected by the type system.
- This is useful for vectors, matrices, and finite choices.

Finite Data: Three Choices

Type	Order	Repetition
`List alpha`	kept	kept
`Multiset alpha`	ignored	kept
`Finset alpha`	ignored	removed

- Use `List` for finite sequences and recursive traversals.
- Use `Multiset` when multiplicity matters but order does not.
- Use `Finset` for finite sets, finite sums, and finite products.

Multiset and Finset in Mathlib

```
import Mathlib

def orderedData : List Nat := [1, 2, 1]
def unorderedWithMultiplicity : Multiset Nat := [1, 2, 1]
def finiteSet : Finset Nat := {1, 2, 1}

#eval orderedData.length
#eval unorderedWithMultiplicity.card
#eval unorderedWithMultiplicity.count 1
#eval finiteSet.card
```

Live Lean: `codeD2/D2_Typeclass_Mathlib.lean`

Multiset as a Quotient

In Mathlib, this is not just an analogy:

```
-- Mathlib.Data.Multiset.Defs
def Multiset (alpha : Type u) : Type u :=
  Quotient (List.isSetoid alpha)

def Multiset.ofList : List alpha -> Multiset alpha :=
  Quot.mk _

theorem Multiset.coe_eq_coe {l1 l2 : List alpha} :
    (l1 : Multiset alpha) = l2 <-> l1 ~ l2 :=
  Quotient.eq
```

- `List.isSetoid alpha` is the permutation equivalence relation.
- The notation `l1 ~ l2` means that the two lists are permutations.
- So `Multiset alpha` is `List alpha` modulo permutation.

Quotient Types: Same Object, Many Names

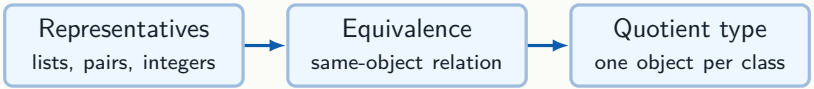
Many mathematical objects are defined by choosing representatives and then identifying equivalent representatives.

- Integers modulo n: different integers may represent the same residue.
- Rational numbers: different pairs may represent the same fraction.
- Multisets: different lists may represent the same unordered collection.
- Quotient groups, rings, and spaces: elements are equivalence classes.

Quotient type

A quotient type packages "representatives modulo an equivalence relation."

Quotient Workflow



Safe operation

Must give the same answer for equivalent representatives.

Lean's demand

Prove well-definedness before defining functions out of a quotient.

Setoid: The Equivalence Relation

Lean separates the representatives from the relation used to identify them.

```
class Setoid (alpha : Type) where
  r : alpha -> alpha -> Prop
  iseqv : Equivalence r
```

- `r a b` means that `a` and `b` represent the same object.
- The field `iseqv` says that `r` is reflexive, symmetric, and transitive.
- Given a `Setoid alpha`, Lean can form `Quotient s`.

A Small Quotient: Parity

```
def sameParity : Setoid Nat where
  r a b := a % 2 = b % 2
  iseqv := {
    refl := by intro a; rfl
    symm := by intro a b h; exact h.symm
    trans := by intro a b c h1 h2; exact h1.trans h2
  }

abbrev Parity := Quotient sameParity

def parityOf (n : Nat) : Parity :=
  Quotient.mk sameParity n
```

The quotient has two intended classes: even representatives and odd representatives.

Live Lean: `codeD2/D2_CommonTypes.lean`

Functions Out of a Quotient

```
def parityValue : Parity -> Nat :=
  Quotient.lift
    (fun n : Nat => n % 2)
    (by
      intro a b h
      exact h)
```

- To define a function out of a quotient, define it on representatives.
- Then prove that equivalent representatives give the same result.
- This proof is the well-definedness condition.

Why Well-Definedness Matters

```
-- This is well-defined on parity classes:
fun n : Nat => n % 2

-- This is not well-defined on parity classes:
fun n : Nat => n
```

- The numbers 2 and 4 represent the same parity class.
- But the representative function `fun n => n` gives different outputs.
- Lean asks for a proof precisely to prevent this mistake.

Reading rule

Quotient definitions force us to prove that the construction does not depend on the chosen representative.

Structures, Finite Data, Quotients

Structures

Named fields bundle data into one object.

Proof fields record mathematical properties.

Field access makes definitions explicit.

Partial data

`Option` makes failure explicit.

`Inhabited` supplies defaults through instance search.

Finite data

`List`: ordered finite data.

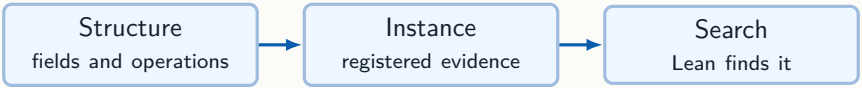
`Fin`: bounded finite indices.

`Multiset`/`Finset`: order-free finite data.

Quotients

Representatives modulo an equivalence relation become a new type.

From Structures to Classes



Class parameter

A type class parameter says: Lean, please find the required structure.

Why Type Classes?

Many types

`Nat`, `Int`, `Rat`, `Complex`

Shared notation

`+`, `*`, `0`, `1`

Generic theorems

state only the structure needed

Reading rule

Do not ask: is this exactly `Nat` or `Int`? Ask: does this type have the structure I need?

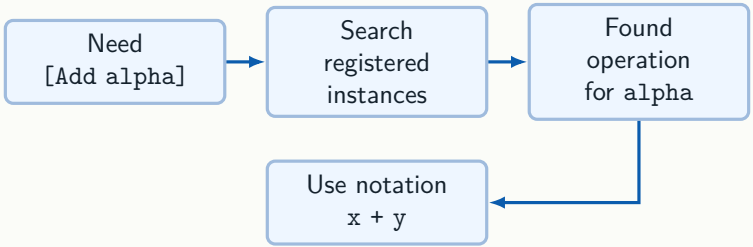
Class as Searchable Structure

A class is very close to a structure:

- it has fields;
- values of the class are pieces of structure;
- instances register those pieces of structure for later search.

The key extra behavior is not mathematical magic. It is automatic lookup.

Instance Search: What Lean Is Doing



Square brackets do not add new mathematics; they ask Lean to find existing mathematical structure.

A Small Type Class

```
class HasNorm (alpha : Type) where
  norm : alpha -> Nat

def doubleNorm [HasNorm alpha] (x : alpha) : Nat :=
  2 * HasNorm.norm x
```

- `[HasNorm alpha]` is a type class parameter.
- Lean tries to synthesize the instance automatically.

Live Lean: `codeD2/D2_Typeclass.lean`

What Is the Instance?

```
class HasNorm (alpha : Type) where
  norm : alpha -> Nat
```

An instance for `HasNorm alpha` is a value containing:

- a function `norm : alpha -> Nat`;
- registered so that Lean can find it later.

The class says what data is required. The instance supplies that data.

Instances

```
structure Vector2 where
  x : Int
  y : Int

instance : HasNorm Vector2 where
  norm v := intAbsNat v.x + intAbsNat v.y
```

- An instance provides the structure required by a class.
- Once registered, it can be found by instance search.
- `#synth` lets us inspect what Lean can find.

Checking Instance Search

```
#synth HasNorm Vector2
#synth HasNorm Nat

def natNormInstance : HasNorm Nat :=
  inferInstance

#eval natNormInstance.norm 9
```

- `#synth` checks instance search at command level.
- `inferInstance` performs instance search inside a term.
- Both are useful for understanding what Lean can find.

Explicit Structure Argument

```
def useExplicit {alpha : Type}
    (inst : HasNorm alpha) (x : alpha) : Nat :=
  inst.norm x

#eval useExplicit (inferInstance : HasNorm Nat) 5
```

- The instance is an ordinary explicit argument.
- The caller must supply it.
- This is honest, but quickly becomes verbose.

Type Class Argument

```
def useInstance {alpha : Type}
    [HasNorm alpha] (x : alpha) : Nat :=
  HasNorm.norm x

#eval useInstance (5 : Nat)
```

- The square brackets activate instance search.
- Lean infers `alpha = Nat` from the argument.
- Then it searches for `HasNorm Nat`.

Three Kinds of Parameters

Syntax	Meaning	Who supplies it?
`(x : alpha)`	explicit value	user writes it
`{alpha : Type}`	ordinary implicit parameter	elaborator infers it
`[HasNorm alpha]`	type class parameter	instance search finds it

Type class parameters are still parameters. They are just filled by a special search procedure.

Named Instance Parameters

```
def useInstanceVerbose {alpha : Type}
    [inst : HasNorm alpha] (x : alpha) : Nat :=
  inst.norm x
```

- We can name the instance if we want to use it directly.
- The brackets still request instance search.
- This is useful when several fields or projections are needed.

Notation Comes from Classes

```
#synth Add Nat
#synth Add Int
#synth Mul Nat
#synth Mul Int

def addThree [Add alpha] (a b c : alpha) : alpha :=
  a + b + c
```

- The symbol + is interpreted using an `Add` instance.
- The symbol * is interpreted using a `Mul` instance.
- To read `a + b`, Lean needs the type of `a`, `b`, and the relevant instance.
- The same notation can work over many mathematical types.

Generic Notation

```
def addThree {alpha : Type} [Add alpha]
    (a b c : alpha) : alpha :=
  a + b + c

#eval addThree (1 : Nat) 2 3
#eval addThree (1 : Int) 2 3
```

- The definition is generic in `alpha`.
- The operation comes from `[Add alpha]`.
- The same code runs at different concrete types.

Mathlib: Algebraic Instances

```
import Mathlib

#synth Add Nat
#synth Add Int
#synth Add Rat
#synth Add Complex

#synth AddCommMonoid Nat
#synth AddCommGroup Int
#synth Ring Int
#synth Field Rat
#synth Field Complex
```

- Mathlib provides a rich algebraic hierarchy.
- Lean can find instances for familiar number systems.
- This is why generic algebraic theorems can apply broadly.

Live Lean: `codeD2/D2_Typeclass_Mathlib.lean`

Classes for Mathematical Structure

Class	Purpose
`Add`, `Mul`	notation for operations
`Zero`, `One`	distinguished elements
`AddCommMonoid`	addition laws with 0 and +
`Group`, `AddGroup`	inverses and group notation
`Ring`, `CommRing`	ring structure
`Field`	field structure

Structure, Class, Instance

Word	Reading
`structure`	bundle a concrete object with named fields
`class`	bundle reusable structure on a type
`instance`	register that a type carries this structure
`[Group G]`	ask Lean to find a group structure on `G`

This is why theorems can be stated once and reused for many mathematical objects.

Multiplicative and Additive Notation

Class	Identity	Operation	Inverse
`[Group G]`	`1`	`g * h`	`g^{-1}`
`[AddGroup A]`	`0`	`a + b`	`-a`

The mathematical idea is similar, but the notation and class names are different.

Generic Group Theorems

```
example {G : Type} [Group G] (g h : G) :
    g * h * h⁻¹ = g := by
  simp [mul_assoc]

example {A : Type} [AddGroup A] (a b : A) :
    a + b + (-b) = a := by
  simp [add_assoc]
```

- The type is arbitrary.
- The class assumption supplies notation and algebraic facts.
- `simp` can use lemmas from the instance.

Operations vs. Laws

- `[Add alpha]` gives an addition operation.
- It does not by itself state associativity, commutativity, or identity laws.
- For laws, use stronger classes such as `AddCommMonoid alpha`, `Group G`, `Ring R`, or `Field K`.

Reading rule

A theorem with `[Ring R]` applies to any type for which Lean can find a ring instance.

Data, Property, and Mixed Classes

Kind	Example	What the instance contains
Data class	`HasNorm alpha`	function *norm* : $\alpha \to Nat$
Data class	`Add alpha`	operation *add* : $\alpha \to \alpha \to \alpha$
Property class	`MulCommLaw alpha`	a proof of commutativity
Mixed structure	`Group G`	operations plus laws

```
class MulCommLaw (alpha : Type) [Mul alpha] : Prop where
  mul_comm_law : forall a b : alpha, a * b = b * a
```

In mathlib, algebraic classes usually contain both operation fields and proof fields.

When a Math Structure Does Not Exist

```
-- Nat has addition, but no additive inverses:
-- #synth AddGroup Nat

-- Int is a ring, but not a field:
-- #synth Field Int
```

- These are useful failures for teaching instance search.
- The class names exist, but Lean cannot find the requested instances.
- This is different from an unknown identifier error.

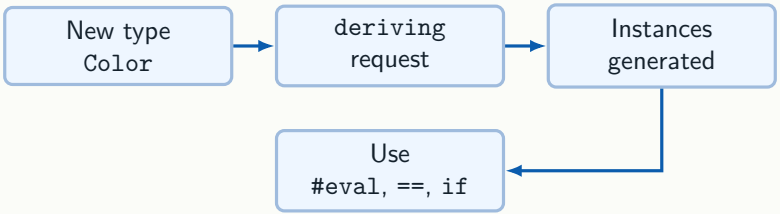
Deriving Standard Instances

```
inductive Color where
  | red | green | blue
  deriving Repr, BEq, DecidableEq
```

- `deriving` generates standard instances.
- It makes custom types usable in small examples immediately.

Live Lean: `codeD2/D2_Deriving.lean`

What `deriving` Automates



`deriving` is not a tactic; it asks Lean to synthesize routine type class instances from constructor data.

Repr, BEq, DecidableEq

Class	Purpose	Typical use
`Repr alpha`	printable representation	`#eval` output
`BEq alpha`	Boolean equality test	`x == y`
`DecidableEq alpha`	equality as proposition	`if x = y then ...`

These are ordinary type class instances, often generated automatically.

BEq: Boolean Equality

```
inductive Direction where
  | north | south | east | west
  deriving Repr, BEq

#eval Direction.north == Direction.north
#eval Direction.north == Direction.south
```

- BEq provides ==.
- The result is a `Bool`.
- This is computational equality testing.

BEq vs DecidableEq

Expression	Type	Meaning
`x == y`	`Bool`	computes a Boolean answer
`x = y`	`Prop`	states a proposition
`Decidable (x = y)`	`Type`	gives a decision procedure

In practice, both are useful, but they serve different layers of Lean.

If-Then-Else

```
def sameDirection (d e : Direction) : String :=
  if d == e then "same" else "different"

def sameColor (c d : Color) : String :=
  if c = d then "same" else "different"
```

- In the first example, `d == e :  Bool`; this uses BEq.
- In the second example, `c = d :  Prop`; Lean needs `Decidable (c = d)`.
- `DecidableEq Color` supplies decisions for equalities between colors.

What Deriving Does

- It inspects the constructors of an inductive type or the fields of a structure.
- It generates instance declarations.
- Those instances are then available to `#eval`, `==`, `decide`, and library APIs.

Constructor data

The constructor shape is enough for Lean to generate many standard behaviors.

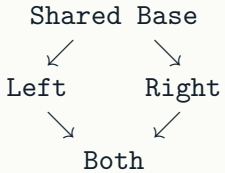
The `extends` Keyword

`extends` says that a class includes the fields and instance behavior of parent classes.

- It is how Lean builds class hierarchies.
- A child class can be used wherever a parent class is required.
- This is essential for algebraic hierarchies.
- It also creates design risks when parent data is duplicated.

A Diamond Warning

Instance search is powerful, but class hierarchies must be designed carefully.



A value of `Both` may carry structure through two paths.

What Is the Diamond Problem?

- Two parent classes may contain or inherit what is mathematically intended to be the same structure.
- Lean sees data, not intention.
- If two copies are stored, Lean will not silently identify them.
- Mathlib's hierarchy is carefully designed to avoid accidental duplication.

Hierarchy design

Prefer existing mathlib hierarchies over inventing parallel class towers.

Two Parent Structures with the Same Field Name

```
class BadSemigroup (alpha : Type) where
  op : alpha -> alpha -> alpha

class BadAddSemigroup (alpha : Type)
    extends BadSemigroup alpha

class BadMulSemigroup (alpha : Type)
    extends BadSemigroup alpha

instance : BadAddSemigroup Int where
  op := Int.add

instance : BadMulSemigroup Int where
  op := Int.mul
```

- $(\mathbb{Z}, +)$ and $(\mathbb{Z}, *)$ are both semigroup-like.
- Both parent classes inherit a field named op.
- The two inherited fields are mathematically different operations.

The Diamond Failure

```
class BadRing (alpha : Type)
    extends BadAddSemigroup alpha, BadMulSemigroup alpha

-- Failing attempt:
-- instance : BadRing Int where
--   toBadAddSemigroup := inferInstance
--   toBadMulSemigroup := inferInstance
```

- `BadRing` receives `BadSemigroup` structure through two parent paths.
- Lean sees two pieces of data named `op`, not one shared operation.
- The repair is to give addition and multiplication different field names.

Fixing the Diamond

```
class GoodAddSemigroup (alpha : Type) where
  add : alpha -> alpha -> alpha

class GoodMulSemigroup (alpha : Type) where
  mul : alpha -> alpha -> alpha

class GoodRingSkeleton (alpha : Type)
    extends GoodAddSemigroup alpha, GoodMulSemigroup alpha

instance : GoodRingSkeleton Int where
  add := Int.add
  mul := Int.mul

#eval (inferInstance : GoodRingSkeleton Int).add 2 3
#eval (inferInstance : GoodRingSkeleton Int).mul 2 3
```

- Use separate fields for mathematically different operations.
- Mathlib follows this idea: `AddSemigroup` uses $+$, `Semigroup` uses $*$.

Diamond: Practical Reading

- For ordinary use, trust mathlib's existing hierarchies.
- Avoid inventing parallel algebraic class towers in class projects.
- If two paths should share data, design the shared data as a common parent.
- If two fields should be equal, add that equality as a law.

The important lesson is not fear of type classes, but respect for hierarchy design.

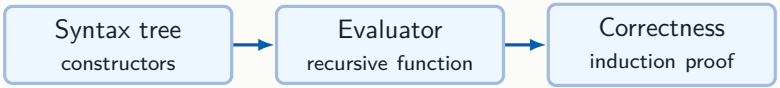
Expression Trees

```
inductive Expr where
  | const : Nat -> Expr
  | var : String -> Expr
  | add : Expr -> Expr -> Expr
  deriving Repr, BEq, DecidableEq
```

Syntax trees are naturally inductive.

Live Lean: `codeD2/D2_IntegratedExample.lean`

Expression Trees: Data, Program, Proof



Expression tree components

Expression trees reuse inductive syntax, recursive evaluation, environment structures, and induction principles for correctness.

Expression Trees Combine the Main Tools

`Expr` combines the main tools in this deck:

- constructors define a syntax tree;
- recursive constructors create recursive programs;
- `Option` handles failed lookup;
- `structure` packages environments;
- `deriving` gives printing and equality;
- induction can prove properties of recursive functions.

Expression trees are the first example of syntax represented as data.

An Environment

```
structure Env where
  lookup : String -> Option Nat

def sampleEnv : Env :=
  { lookup := fun name =>
      if name = "x" then some 10
      else if name = "y" then some 5
      else none }
```

- An environment interprets variable names.
- Lookup can fail, so the return type is `Option Nat`.
- This is a structure bundling one operation.

Evaluating Expressions

```
def Expr.eval (env : Env) : Expr -> Option Nat
  | const n => some n
  | var name => env.lookup name
  | add left right =>
      match eval env left, eval env right with
      | some m, some n => some (m + n)
      | _, _ => none
```

- Evaluation is recursive on the expression tree.
- Addition succeeds only if both subexpressions succeed.
- Failure propagates through `none`.

A Glimpse of Lean's Real Expr

```
inductive Lean.Expr where
  | bvar    : Nat -> Expr
  | fvar    : FVarId -> Expr
  | mvar    : MVarId -> Expr
  | sort    : Level -> Expr
  | const   : Name -> List Level -> Expr
  | app     : Expr -> Expr -> Expr
  | lam     : Name -> Expr -> Expr -> BinderInfo -> Expr
  | forallE : Name -> Expr -> Expr -> BinderInfo -> Expr
  | letE    : Name -> Expr -> Expr -> Expr -> Bool -> Expr
  | lit     : Literal -> Expr
  | mdata   : MData -> Expr -> Expr
  | proj    : Name -> Nat -> Expr -> Expr
```

Why Lean.Expr Matters

```
import Lean

#check Lean.Expr
#check Lean.Expr.bvar
#check Lean.Expr.const
#check Lean.Expr.app
#check Lean.Expr.lam
#check Lean.Expr.forallE
#check Lean.Expr.letE
```

- Our `Expr` is a small teaching language.
- Lean's real `Expr` represents terms in Lean itself.
- We only need the impression, not elaboration or metaprogramming.

Expression Trees in Programs

- Expressions can be represented as inductive syntax trees.
- Functional programs traverse and transform such trees.
- Correctness proofs use recursion and induction on the same syntax.
- Optimizers and interpreters rely on this data/program/proof pattern.

Expression Trees and Type Class Search

With the small `Expr` example, we see:

- `inductive` to define syntax trees;
- pattern matching to evaluate expressions;
- `Option` for failed variable lookup;
- `structure` to bundle an environment;
- `deriving` for printing and equality;
- induction as the proof principle for recursive syntax.

Type Class Search

Search

A `class` is a searchable structure.

An `instance` registers evidence.

`#synth` exposes the search result.

Algebra

`[Add alpha]` supplies notation.

`[Ring R]` supplies operations and laws.

Mathlib reuses these instances broadly.

Generated instances

`deriving` builds `Repr`, `BEq`, and `DecidableEq`.

Expression trees

Inductive syntax, evaluation, structures, and classes appear together.

Data Shape and Reuse

Data shape

Inductive declarations specify how values are built.

They drive pattern matching and recursion.

Cases and induction follow the same shape.

Reusable structure

Structures bundle named fields.

Classes and instances reuse structure by search.

Deriving generates standard behavior.

Syntax trees

`Expr` shows the next step: inductive data can represent syntax trees for programs and mathematics.

Exercises

- Complete the guided holes in `D2_Exercises.lean`.
- Use `Inhabited` to write a default-returning `Option` function.
- Use `inferInstance` to fill an explicit class argument.
- Experiment with `Repr`, `BEq`, and `DecidableEq`.
- Modify the simple `Expr` language.
- Extend the evaluator.
- Prove a small theorem by induction on expressions.

Live Lean: `codeD2/D2_Exercises.lean`