AI4Math Summer School · D3

# Lean and AI

## From algebraic instances to kernel-checked formalization

**Today's route**

Algebraic hierarchy → substructures → modules and choice

Categories and complexes → AI-assisted Lean

Finite counting and graphs

**Three-hour session**

Instances, Mathlib search, research-level interfaces, and AI failure analysis.

# Learning Goals

### Read algebraic interfaces

Explain what `AddGroup`, `Group`, `Ring`, `Field`, and `Module` require.

### Build and inspect instances

Separate operation fields from law fields; diagnose typeclass search with `#synth` and `#check`.

### Use substructures

Read subtype elements and construct subgroups, subrings, and submodules by closure proofs.

### See research-level interfaces

Recognize categories, functors, natural transformations, and homological complexes in Mathlib.

### Formalize finite mathematics

Use `Finset`, `SimpleGraph`, `simp`, `ring`, `linarith`, and `decide`.

### Use AI critically

Let AI propose statements and proofs, then verify semantics, API names, assumptions, and kernel acceptance.

# An Instance Is Lean's Entry Point to Structure

| Object | Structure | Theorems |
| --- | --- | --- |
| `G : Type*` | `[Group G]` | `mul_assoc` |
| only gives a type | asks typeclass search | becomes available |

```
variable (G : Type*) [Group G]

#synth Group G
#check mul_assoc
#check inv_mul_cancel
```

Parentheses introduce ordinary arguments. Square brackets introduce arguments that Lean should synthesize automatically.

**Environment:** Open and run these files inside a configured Mathlib Lake project. The distributed D3 folder alone is not a Lake project.

Live Lean: `D3_01_Inspect.lean`

# Writing an Instance: Data First, Laws Second

**Operation data**

- `mul`, `one`, `inv`
- `add`, `zero`, `neg`
- `Hom`, `id`, `comp`

These fields determine what notation means.

**Proof data**

- associativity, identities, inverses
- distributivity and commutativity
- category identity and associativity laws

Every law is a separate Lean goal.

**Practical inspection loop:** use `#print` to inspect fields; write a field followed by `:= by`; read the new goal; prove it; continue to the next field.

# `extends` Reuses a Parent Interface

```
class Add (G : Type u) where
  add : G → G → G

class AddSemigroup (G : Type u)
    extends Add G where
  protected add_assoc : ∀ a b c : G,
    a + b + c = a + (b + c)

variable (G : Type*) [AddSemigroup G]
#synth Add G
#check AddSemigroup.toAdd
```

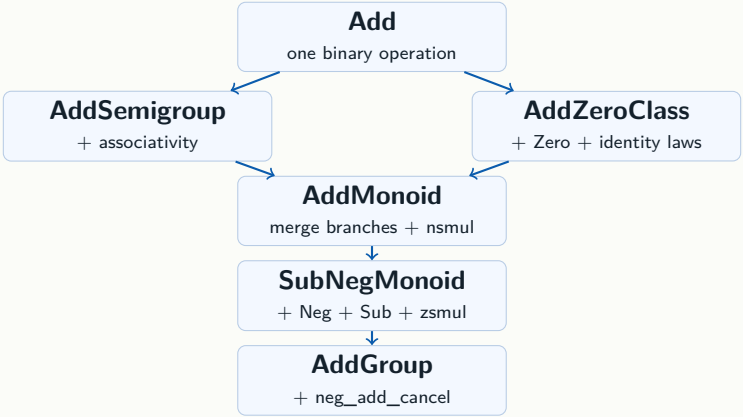
**What `extends` does**

- inherits the fields of `Add G`
- adds the new field `add_assoc`
- creates a parent projection
- lets typeclass search use the child as the parent
- does not prove the new law

**Multiple inheritance:** `AddMonoid G extends AddSemigroup G, AddZeroClass G`. An `AddMonoid G` can be used through either parent interface, but an `Add G` alone cannot be promoted to either child.

Live Lean: `D3_02_AddGroup_Group.lean`

# The Actual Mathlib Path from Add to AddGroup




**Two branches merge at AddMonoid.** Associativity comes from `AddSemigroup`; zero and its identity laws come from `AddZeroClass`. Negation does not become an additive inverse until the final `AddGroup` axiom is proved.

# Step 0: Add Supplies an Operation, Not a Law

```
class Add (G : Type u) where
  add : G → G → G

class AddSemigroup (G : Type u)
    extends Add G where
  protected add_assoc : ∀ a b c : G,
    a + b + c = a + (b + c)
```

**After Add**

- Lean can elaborate `a + b`.
- No theorem relates different sums.
- Even `(a+b)+c = a+(b+c)` is unavailable.

**AddSemigroup adds exactly one mathematical law:** associativity.

Core fields from the current Mathlib hierarchy; documentation, universes, and default proof bodies are omitted.

Live Lean: `D3_02_AddGroup_Group.lean`

# Step 1: Add Zero and Merge the Two Branches

```
class AddZero (G : Type u) extends Zero G, Add G

class AddZeroClass (G : Type u)
    extends AddZero G where
  protected zero_add : ∀ a : G, 0 + a = a
  protected add_zero : ∀ a : G, a + 0 = a

class AddMonoid (G : Type u)
    extends AddSemigroup G, AddZeroClass G where
  protected nsmul : ℕ → G → G
  protected nsmul_zero : ∀ x, nsmul 0 x = 0
  protected nsmul_succ : ∀ n x,
    nsmul (n + 1) x = nsmul n x + x
```

**Why three classes?**

- `AddZero`: data only
- `AddZeroClass`: zero laws
- `AddMonoid`: also associativity
- `nsmul`: interprets `n • a`

Natural scalar multiplication is an engineering field with a standard default implementation.

Core fields from the current Mathlib hierarchy; documentation, universes, and default proof bodies are omitted.

Live Lean: `D3_02_AddGroup_Group.lean`

# `nsmul`: Repeated Addition by a Natural Number

```
class AddMonoid (M : Type u) where
  protected nsmul : ℕ → M → M
  protected nsmul_zero : ∀ a,
    nsmul 0 a = 0
  protected nsmul_succ : ∀ n a,
    nsmul (n + 1) a = nsmul n a + a
```

```
variable (M : Type*) [AddMonoid M]
#synth SMul ℕ M
#check zero_nsmul
#check succ_nsmul

example : (3 : ℕ) • (5 : ℤ) = 15 := by
  norm_num
```

## Mathematical meaning

- `0 • a = 0`
- `(n+1) • a = n • a + a`
- hence `n • a` is (`n`) copies of (`a`)

## Why it is needed

- gives `SMul ℕ M` to every additive monoid
- enables `n • a` in algebraic statements
- stores one operation to avoid instance diamonds

Core fields from the current Mathlib hierarchy; documentation, universes, and default proof bodies are omitted.

Live Lean: `D3_02_AddGroup_Group.lean`

# Step 2: SubNegMonoid Adds Syntax for Negation and Subtraction

```
class SubNegMonoid (G : Type u)
    extends AddMonoid G, Neg G, Sub G where
  protected sub : G → G → G :=
    fun a b => a + -b
  protected sub_eq_add_neg : ∀ a b : G,
    a - b = a + -b
  protected zsmul : ℤ → G → G
  protected zsmul_zero' : ∀ a, zsmul 0 a = 0
  protected zsmul_succ' : ∀ n a,
    zsmul n.succ a = zsmul n a + a
  protected zsmul_neg' : ∀ n a,
    zsmul (Int.negSucc n) a = -zsmul n.succ a
```

**Important distinction**

- `Neg G` only defines `-a`.
- `Sub G` only defines `a-b`.
- `sub_eq_add_neg` connects the notations.
- `zsmul` gives integer scalar multiplication.
- No inverse law has been required yet.

Live Lean: `D3_02_AddGroup_Group.lean`

# `nsmul` and `zsmul`: Two Natural Interfaces

```
class NsmulData (α : Type u) where
  nsmul : ℕ → α → α

class ZsmulData (α : Type u) where
  zsmul : ℤ → α → α

-- two mathematically natural definitions
n • x := x + x + ⋯ + x       -- n times
n • x := (n : ℤ) • x          -- restrict zsmul to ℕ
```

**The mathematics**

- both definitions express repeated addition
- negative integers use `-x`
- under the intended laws, they should agree

**The Lean question** Which operation is the actual data behind `n • x`? If two instance paths construct it, the definitions may be mathematically equal without being definitionally equal.

**Not copy-paste Lean code; schematic extract only.** Universes, implicit parameters, and some inherited fields are omitted.

Live Lean: `D3_02_AddGroup_Group.lean`

# A Bad Hierarchy Creates Two `nsmul` Paths

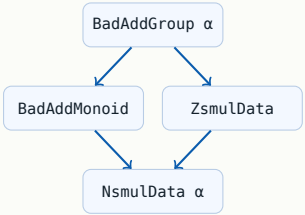
```
class BadAddMonoid (α : Type u)
    extends Add α, Zero α, NsmulData α

class BadAddGroup (α : Type u)
    extends Add α, Zero α, Neg α

instance groupToMonoid [BadAddGroup α] : BadAddMonoid α where
  nsmul := repeatedAddition

instance groupToZsmul [BadAddGroup α] : ZsmulData α where
  zsmul := integerRepeatedAddition

instance nsmulFromZsmul [ZsmulData α] : NsmulData α where
  nsmul n x := ZsmulData.zsmul (n : ℤ) x
```




There are two routes from `BadAddGroup α` to the same target instance. Priority can select one route, but it does not make the operations coherent.

**Not copy-paste Lean code; schematic extract only.** Universes, implicit parameters, and some inherited fields are omitted.

Live Lean: `D3_02_AddGroup_Group.lean`

# Why the Diamond Needs Definitional Equality

```
-- the same expression, reached by two paths
-- Path A: 2 • x  ↦  x + (x + 0)
-- Path B: 2 • x  ↦  (2 : ℤ) • x
--              ↦  zsmulRec (2 : ℤ) x

-- Mathlib's actual scalar operation
#synth SMul ℕ (Polynomial ℕ)
example (n : ℕ) (p : Polynomial ℕ) :
    AddMonoid.nsmul n p = n • p := by
  rfl
```

**Two notions of equality**

- propositional: needs a proof and possibly `rw`
- definitional: both terms reduce to the same expression; `rfl` works

**Mathlib's fix**

- `nsmul` is stored in `AddMonoid`
- `zsmul` is stored in `SubNegMonoid`, the parent of `AddGroup`
- `SMul` projects these fixed fields

**All paths reuse fixed fields.**

Core fields from the current Mathlib hierarchy; documentation, universes, and default proof bodies are omitted.

Live Lean: `D3_02_AddGroup_Group.lean`

# Step 3: AddGroup Adds the Inverse Axiom

```
class AddGroup (G : Type u)
    extends SubNegMonoid G where
  protected neg_add_cancel : ∀ a : G,
    -a + a = 0
```

```
#check neg_add_cancel
#check add_neg_cancel
#check sub_self
#check neg_add_rev
#check add_left_cancel
```

**Only one new axiom is stored.**

Mathlib derives:

- the right inverse law `a + -a = 0`
- cancellation laws
- `a - a = 0`
- involutive negation
- negation reversing addition

Typeclass search also synthesizes `AddCancelMonoid` and `SubtractionMonoid`.

Core fields from the current Mathlib hierarchy; documentation, universes, and default proof bodies are omitted.

Live Lean: `D3_02_AddGroup_Group.lean`

# Practical Construction: First Provide Add, Zero, and Neg

```
@[ext]
structure WrappedInt where
  val : ℤ

instance : Add WrappedInt where
  add a b := ⟨a.val + b.val⟩

instance : Zero WrappedInt where
  zero := ⟨0⟩

instance : Neg WrappedInt where
  neg a := ⟨-a.val⟩

#synth Add WrappedInt
-- #synth AddGroup WrappedInt  -- not yet
```

**At this point Lean knows only the operations.**

- `a + b` unwraps, adds integers, and wraps again.
- `0` wraps integer zero.
- `-a` wraps integer negation.
- No group theorem is available yet.

Live Lean: `D3_02_AddGroup_Group.lean`

# AddGroup.ofLeftAxioms Builds the Full Instance

```
instance : AddGroup WrappedInt :=
  AddGroup.ofLeftAxioms
    (fun a b c => by
      apply WrappedInt.ext
      exact add_assoc a.val b.val c.val)
    (fun a => by
      apply WrappedInt.ext
      exact zero_add a.val)
    (fun a => by
      apply WrappedInt.ext
      exact neg_add_cancel a.val)

#synth AddMonoid WrappedInt
#synth SubNegMonoid WrappedInt
#synth AddGroup WrappedInt
#synth AddCancelMonoid WrappedInt
```

Live Lean: `D3_02_AddGroup_Group.lean`

**Only three proofs**

1. associativity
2. left zero law
3. left inverse law

The constructor supplies default `sub`, `nsmul`, and `zsmul`, and derives the remaining group laws.

# AddGroup and Group: Same Pattern, Different Interface

| Structure | Notation and data | Core laws |
|---|---|---|
| `AddGroup A` | `0`, `+`, `-a` | additive associativity, zero laws, additive inverse |
| `Group G` | `1`, `*`, $g^{-1}$ | multiplicative associativity, identity laws, multiplicative inverse |

```
example (A : Type*) [AddGroup A] (a : A) :
    -a + a = 0 := neg_add_cancel a

example (G : Type*) [Group G] (g : G) :
    g⁻¹ * g = 1 := inv_mul_cancel g
```

They are distinct classes. Mathlib relates many parallel theorems through `to_additive`; notation still determines the interface Lean searches for.

Live Lean: `D3_02_AddGroup_Group.lean`

# The `Group` Class in Mathlib

```
class DivInvMonoid (G : Type u)
    extends Monoid G, Inv G, Div G where
  protected div_eq_mul_inv : ∀ a b : G,
    a / b = a * b⁻¹

class Group (G : Type u)
    extends DivInvMonoid G where
  protected inv_mul_cancel : ∀ a : G,
    a⁻¹ * a = 1
```

**Parent chain**

- `Monoid`: `*`, `1`, monoid laws
- `DivInvMonoid`: `⁻¹`, `/`, compatible division
- `Group`: stored law `a⁻¹*a = 1`

```
variable (G : Type*) [Group G]
#synth Monoid G
#synth DivInvMonoid G
#check Group.toDivInvMonoid
#check Group.inv_mul_cancel     -- stored field
#check inv_mul_cancel            -- public theorem
```

Core fields from the current Mathlib hierarchy; documentation, universes, and default proof bodies are omitted.

Live Lean: `D3_02_AddGroup_Group.lean`

# Why AddGroup Appears Before Linear Algebra

### Group-theoretic language

- `[Group G]` gives a multiplicative group.
- Theorems use `a * b` and $a^{-1}$.
- Order matters in noncommutative groups.

### Linear-algebra language

- `[AddCommGroup V]` gives vector addition.
- A module adds scalar multiplication `r • v`.
- Vector spaces use the additive interface.

```
#check add_assoc       -- additive interface
#check mul_assoc       -- multiplicative interface
#check neg_add_cancel  -- AddGroup
#check inv_mul_cancel  -- Group
```

# Example: Define the Operations of the Sign Group

```
inductive Sign where
  | pos
  | neg
  deriving DecidableEq, Repr

open Sign

instance : Group Sign where
  mul a b :=
    match a, b with
    | pos, x => x
    | neg, pos => neg
    | neg, neg => pos
  one := pos
  inv a := a
  -- law fields come next
```

Live Lean: D3_04_Group_Instance.lean

**Mathematical reading**

- `pos` is the identity.
- `neg * neg = pos`.
- Every element is self-inverse.

First define the table. Then let Lean expose one law goal at a time.

# Check the Group Laws One by One

```
mul_assoc := by
  intro a b c
  cases a <;> cases b <;> cases c <;> rfl

one_mul := by
  intro a
  cases a <;> rfl

mul_one := by
  intro a
  cases a <;> rfl

inv_mul_cancel := by
  intro a
  cases a <;> rfl
```

### What Lean checks

1. `intro` exposes arbitrary elements.
2. `cases` enumerates `pos`/`neg`.
3. `rfl` checks the reduced definitions.

```
#synth Group Sign

example (a : Sign) :
    a⁻¹ * a = 1 :=
  inv_mul_cancel a
```

# Subtype: A Value Packaged with a Proof

```
def NonzeroInt := {n : ℤ // n ≠ 0}

example : NonzeroInt :=
  ⟨1, by norm_num⟩

example (x : NonzeroInt) : ℤ :=
  x.1

example (x : NonzeroInt) : x.1 ≠ 0 :=
  x.property

#check Subtype.val
#check Subtype.property
```

Live Lean: D3_03_Subtype.lean

### How to read it

- `x.1` is the underlying value.
- `x.2` or `x.property` is the proof.
- `⟨value, proof⟩` constructs an element.

Subgroups, subrings, and submodules all reuse this pattern.

# Subgroup = Carrier Predicate + Closure Proofs

```
def oneSubgroup (G : Type*) [Group G] :
    Subgroup G where
  carrier := {g | g = 1}
  one_mem' := by
    rfl
  mul_mem' := by
    intro a b ha hb
    rw [ha, hb]
    simp
  inv_mem' := by
    intro a ha
    rw [ha]
    simp
```

Live Lean: D3_05_Subgroup.lean

**Field-by-field check**

1. Which `g : G` belong?
2. Is `1` included?
3. Is multiplication closed?
4. Is inversion closed?

Mathlib then provides a `Group` instance on the subtype.

# Concrete Example: A Subgroup of the Sign Group

```
def posSubgroup : Subgroup Sign where
  carrier := {g | g = Sign.pos}
  one_mem' := by
    rfl
  mul_mem' := by
    intro a b ha hb
    rw [ha, hb]
    rfl
  inv_mem' := by
    intro a ha
    rw [ha]
    rfl
```

```
#synth Group posSubgroup

example (x : posSubgroup) :
    (x : Sign) = Sign.pos := by
  exact x.property
```

Live Lean: D3_05_Subgroup.lean

**Closure check**

- identity: `pos`
- product: `pos * pos = pos`
- inverse: $\texttt{pos}^{-1} = \texttt{pos}$

The subtype `posSubgroup` automatically inherits the group operations from `Sign`.

# Group Homomorphisms in Mathlib: G →* H

```
variable {G H : Type*} [Group G] [Group H]

def conjugationHom (g : G) : G →* G where
  toFun x := g * x * g⁻¹
  map_one' := by
    simp
  map_mul' := by
    intro x y
    group
```

```
example (f : G →* H) (x : G) :
    f x⁻¹ = (f x)⁻¹ := by
  exact map_inv f x

example (f : G →* H) (x : G) (n : ℤ) :
    f (x ^ n) = f x ^ n := by
  exact map_zpow f x n
```

Live Lean: D3_05_GroupHom.lean

### A bundled map

`G →* H` is notation for a bundled `MonoidHom`. To construct one, provide:

- `toFun` of type $G \to H$;
- `map_one'`;
- `map_mul'`.

### Obtained automatically

Preservation of inverses, division, natural powers, and integer powers.

# Kernel, Range, and the General Theorem

```
variable {G H : Type*} [Group G] [Group H]
variable (f : G →* H)

example (x : G) :
    x ∈ f.ker ↔ f x = 1 := by
  exact MonoidHom.mem_ker

example (y : H) :
    y ∈ f.range ↔ ∃ x : G, f x = y := by
  rfl

example : (f.ker).Normal := by
  infer_instance
```

```
#check QuotientGroup.quotientKerEquivRange
-- quotient by f.ker ≃* f.range
```

The theorem targets `f.range`, so no surjectivity assumption is needed.

Live Lean: D3_05_GroupHom.lean

**First isomorphism theorem**

For every group homomorphism $f\colon G \to H$,

$$G/\ker(f) \cong \mathrm{range}(f).$$

- `f.ker` is a subgroup of $G$;
- `f.range` is a subgroup of $H$;
- `infer_instance` supplies normality of the kernel.

Normality makes $G/\ker(f)$ a group.

# Concrete Homomorphism: Reduction Modulo Five

```
def modFiveHom :
    Multiplicative ℤ →* Multiplicative (ZMod 5) :=
  (Int.castAddHom (ZMod 5)).toMultiplicative

example (m n : Multiplicative ℤ) :
    modFiveHom (m * n) =
      modFiveHom m * modFiveHom n := by
  exact map_mul modFiveHom m n
```

```
example (z : Multiplicative ℤ) :
    z ∈ modFiveHom.ker ↔
      Dvd.dvd ((5 : ℕ) : ℤ) z.toAdd := by
  change ((z.toAdd : ℤ) : ZMod 5) = 0 ↔
    Dvd.dvd ((5 : ℕ) : ℤ) z.toAdd
  exact ZMod.intCast_zmod_eq_zero_iff_dvd
    z.toAdd 5
```

Live Lean: D3_05_GroupHom.lean

**Mathematical reading** The type tag changes additive notation into multiplicative notation:

$$m * n \quad \text{means} \quad m + n.$$

Therefore `modFiveHom` is the familiar map

$$\mathbb{Z} \longrightarrow \mathbb{Z}/5\mathbb{Z}, \qquad z \longmapsto [z]_5.$$

Its kernel consists exactly of multiples of five: $\ker f = 5\mathbb{Z}$.

# First Isomorphism Theorem: Complete Lean Code

```
theorem modFiveHom_surjective :
    Function.Surjective modFiveHom := by
  change Function.Surjective
    ((Int.castAddHom (ZMod 5)).toMultiplicative)
  intro y
  obtain ⟨z, hz⟩ := ZMod.intCast_surjective y.toAdd
  refine ⟨Multiplicative.ofAdd z, ?_⟩
  apply Multiplicative.toAdd.injective
  simpa using hz

noncomputable def modFiveFirstIso :=
  QuotientGroup.quotientKerEquivRange modFiveHom

noncomputable def modFiveFirstIsoOntoCodomain :=
  QuotientGroup.quotientKerEquivOfSurjective
    modFiveHom modFiveHom_surjective
```

**Two APIs**

1. `quotientKerEquivRange` always targets the range.
2. The surjective variant targets the whole codomain.

Here Lean obtains

$$\mathbb{Z}/5\mathbb{Z} \cong \mathbb{Z}/5\mathbb{Z}.$$

The left side is a quotient group; the right side is the concrete finite group.

Live Lean: `D3_05_GroupHom.lean`

# Two Group Instances: Compare Their Operations

```
def IntegerGroup := Multiplicative ℤ
def RationalGroup := Multiplicative ℚ

instance group_1 : Group IntegerGroup :=
  Multiplicative.group

instance group_2 : Group RationalGroup :=
  Multiplicative.group

def intToRat : IntegerGroup →* RationalGroup :=
  show Multiplicative ℤ →* Multiplicative ℚ from
    (Int.castAddHom ℚ).toMultiplicative
```

```
example (m n : IntegerGroup) :
    intToRat (m * n) =
      intToRat m * intToRat n := by
  exact map_mul intToRat m n
```

Live Lean: D3_05_Subgroup.lean

**What multiplication means**

- in `IntegerGroup`: $m * n$ means $m + n$ in $\mathbb{Z}$;
- in `RationalGroup`: $x * y$ means $x + y$ in $\mathbb{Q}$.

Thus `map_mul` is exactly

$$\iota(m+n) = \iota(m) + \iota(n),$$

the additivity of the integer cast $\iota : \mathbb{Z} \to \mathbb{Q}$.

# Step 1: Prove the Homomorphism Is Injective

```
theorem intToRat_injective :
    Function.Injective intToRat := by
  change Function.Injective
    ((Int.castAddHom ℚ).toMultiplicative)
  intro m n h
  apply Multiplicative.toAdd.injective
  have h' : (m.toAdd : ℚ) = (n.toAdd : ℚ) := by
    simpa using congrArg Multiplicative.toAdd h
  exact_mod_cast h'
```

The mathematical core is

$$(m : \mathbb{Q}) = (n : \mathbb{Q}) \Longrightarrow m = n \qquad (m, n \in \mathbb{Z}).$$

Lean's extra steps only remove the multiplicative type tags.

**Read the proof in four steps**

1. Unfold the named homomorphism with `change`.
2. Assume `intToRat m = intToRat n`.
3. Use `toAdd` to expose equality of the rational casts.
4. `exact_mod_cast` returns equality in $\mathbb{Z}$.

Live Lean: D3_05_Subgroup.lean

# Step 2: The Range Is the Required Subgroup

```
def integerRange : Subgroup RationalGroup :=
  intToRat.range

#synth Group integerRange

noncomputable def group_1_as_subgroup :
    IntegerGroup ≃* intToRat.range :=
  MonoidHom.ofInjective intToRat_injective
```

## Instance versus subgroup

`group_1` supplies operations and laws on `IntegerGroup`. The actual `Subgroup RationalGroup` term is `integerRange`; the equivalence says that it is a copy of `IntegerGroup` inside `RationalGroup`.

Live Lean: `D3_05_Subgroup.lean`

### The precise statement

1. `intToRat` preserves the group operation.
2. Injectivity says no two integers become equal in $\mathbb{Q}$.
3. Its image `intToRat.range` is a `Subgroup RationalGroup`.
4. `ofInjective` identifies the source group with that image.

# Ring Combines Additive and Multiplicative Structure

| Layer | Mathematical requirement | Mathlib entry point |
|---|---|---|
| **Addition** | commutative group: $0, +, -$ | `AddCommGroup` |
| **Multiplication** | monoid: $1, *$ | `Monoid` |
| **Interaction** | left and right distributivity | `left_distrib`, `right_distrib` |
| **Commutative ring** | multiplication is commutative | `CommRing`, `mul_comm` |

**Inspection strategy:** identify the operation data `zero/add/neg/one/mul`, then check the additive, multiplicative, and distributive law goals.

# Transport a Ring Structure Along an Equivalence

```
@[ext]
structure PairInt where
  x : Int
  y : Int

def equivProd : PairInt ≃ Int × Int where
  toFun p := (p.x, p.y)
  invFun q := ⟨q.1, q.2⟩
  left_inv := by intro p; ext <;> rfl
  right_inv := by intro q; cases q; rfl

instance : CommRing PairInt :=
  equivProd.commRing
```

Live Lean: `D3_06_Ring_Instance.lean`

**Engineering principle**

- `Int × Int` already has a ring.
- `PairInt` is equivalent to it.
- Transfer avoids repeating every law proof.

# Check the Resulting Ring Interface

```
#synth CommRing PairInt
#check add_assoc
#check neg_add_cancel
#check left_distrib
#check right_distrib
#check mul_assoc
#check mul_comm

example (p q r : PairInt) :
    p * (q + r) = p * q + p * r := by
  exact mul_add p q r

example (p q : PairInt) :
    (p + q)^2 = p^2 + 2*p*q + q^2 := by
  ring
```

**Four-step check**

1. `#synth` confirms the instance.
2. `#check` reveals theorem assumptions.
3. Small lemmas test the interface.
4. `ring` normalizes polynomial identities.

# Subring Adds Both Additive and Multiplicative Closure

```
def diagonalSubring : Subring (ℤ × ℤ) where
  carrier := {p | p.1 = p.2}
  zero_mem' := by simp
  one_mem' := by simp
  add_mem' := by
    intro a b ha hb
    simp at ha hb ⊢
    rw [ha, hb]
  neg_mem' := by
    intro a ha
    simp at ha ⊢
    rw [ha]
  mul_mem' := by
    intro a b ha hb
    simp at ha hb ⊢
    rw [ha, hb]
```

Live Lean: D3_07_Subring.lean

**What changed from Subgroup?**

- constants: `0` and `1`
- additive closure
- negation closure
- multiplicative closure

An element still exposes its proof through `x.property`.

# Field = Commutative Ring + Inverses for Nonzero Elements

| Requirement | Mathematical meaning | Lean consequence |
|---|---|---|
| `CommRing K` | first, a commutative ring | polynomial normalization is available |
| `Nontrivial K` | at least two elements | $0 \neq 1$ |
| `inv, div` | total inverse and division operations | $x / y = x * y^{-1}$ |
| `mul_inv_cancel₀` | nonzero elements have inverses | theorem requires $x \neq 0$ |
| `inv_zero` | Lean defines $0^{-1} = 0$ | inverse remains a total function |

**Common AI error:** proposing $x / x = 1$ without the assumption $x \neq 0$. This is a false statement, not merely a failed tactic.

# Example: Wrap $\mathbb{Q}$ and Transfer Its Field Structure

```
@[ext]
structure WrappedQ where
  val : ℚ

def equivRat : WrappedQ ≃ ℚ where
  toFun x := x.val
  invFun q := ⟨q⟩
  left_inv := by intro x; ext; rfl
  right_inv := by intro q; rfl

instance : Field WrappedQ :=
  equivRat.field

example (x : WrappedQ) (hx : x ≠ 0) :
    x / x = 1 := by
  field_simp [hx]
```

Live Lean: D3_08_Field_Instance.lean

**What to notice**

- The source structure is the known field $\mathbb{Q}$.
- The equivalence transports all operations and laws.
- The theorem still needs a nonzero hypothesis.

# Module Is Mathlib's Interface for Vector Spaces

| Part | Mathematical meaning | Lean assumption |
|---|---|---|
| **Scalars** | coefficients that add and multiply | `[Semiring R]` or `[Field K]` |
| **Vectors** | an additive commutative group | `[AddCommGroup V]` |
| **Action** | scalar multiplication $r \cdot v$ | `[Module R V]` |
| **Compatibility** | distributivity, associativity, identity | `smul_add`, `add_smul`, `mul_smul`, `one_smul` |

```
variable {K : Type*} [Field K]
variable {V : Type*} [AddCommGroup V] [Module K V]

#check smul_add
#check add_smul
#check mul_smul
#check one_smul
```

Live Lean: D3_09_Module_Submodule.lean

# Submodule = Subtype + Linear Closure

```
def diagonalSubmodule : Submodule ℚ (ℚ × ℚ) where
  carrier := {p | p.1 = p.2}
  zero_mem' := by
    simp
  add_mem' := by
    intro a b ha hb
    simp at ha hb ⊢
    rw [ha, hb]
  smul_mem' := by
    intro c p hp
    change c * p.1 = c * p.2
    rw [hp]
```

Live Lean: D3_09_Module_Submodule.lean

**Field-by-field check**

1. define the carrier
2. include zero
3. prove addition closure
4. prove scalar closure

Mathlib supplies an additive group and module structure on the subtype.

# Model a Symplectic Vector Space in Layers

```
variable (F W : Type*)
variable [Field F] [Fintype F]
variable [AddCommGroup W] [Module F W]
variable [FiniteDimensional F W]

def IsAlternating
    (ω : W →ₗ[F] W →ₗ[F] F) : Prop :=
  ∀ w : W, ω w w = 0

def IsNondegenerate
    (ω : W →ₗ[F] W →ₗ[F] F) : Prop :=
  ∀ w : W, (∀ v : W, ω w v = 0) → w = 0
```

Live Lean: D3_10_VectorSpace_Choose.lean

## Modeling order

1. choose scalar and vector types
2. add algebraic instances
3. define the bilinear form
4. state mathematical properties in `Prop`

# CompletePolarization Stores a Decomposition Hypothesis

```
def TotallyIsotropic
    (ω : W →ₗ[F] W →ₗ[F] F)
    (U : Submodule F W) : Prop :=
  ∀ u : W, u ∈ U →
    ∀ v : W, v ∈ U → ω u v = 0

structure CompletePolarization
    (ω : W →ₗ[F] W →ₗ[F] F) where
  X : Submodule F W
  Y : Submodule F W
  isotropic_X : TotallyIsotropic F W ω X
  isotropic_Y : TotallyIsotropic F W ω Y
  disjoint : Disjoint X Y
  sup_eq_top : sup X Y = ⊤
```

**Each field is reusable**

- X, Y: subspaces
- isotropy proofs
- trivial intersection
- X + Y = W

Later proofs call fields such as `P.sup_eq_top` directly.

# Classical.choose Turns Existence into a Chosen Object

```
#check Classical.choose
-- (h : ∃ x, p x) → α

#check Classical.choose_spec
-- p (Classical.choose h)

lemma exists_decomposition
    (P : CompletePolarization F W ω) (w : W) :
    ∃ x : W, x ∈ P.X ∧
      ∃ y : W, y ∈ P.Y ∧ w = x + y := by
  -- derive membership in sup X Y from P.sup_eq_top
  ...

noncomputable def xComponent (w : W) : W :=
  Classical.choose (exists_decomposition F W P w)
```

**Choice pattern**

1. prove $\exists x,\ p(x)$
2. use `choose` for a witness
3. use `choose_spec` for its property
4. mark definitions `noncomputable` when needed

The ellipsis is presentation shorthand; the linked Lean file contains the complete proof.

Live Lean: D3_10_VectorSpace_Choose.lean

# `Classical.choose_spec` Recovers the Guarantee

```
noncomputable def yComponent
    (P : CompletePolarization F W ω) (w : W) : W :=
  Classical.choose
    ((Classical.choose_spec
      (exists_decomposition F W P w)).2)

lemma decompose_by_components
    (P : CompletePolarization F W ω) (w : W) :
    w = xComponent F W ω P w +
        yComponent F W ω P w := by
  dsimp [xComponent, yComponent]
  exact
    (Classical.choose_spec
      ((Classical.choose_spec
        (exists_decomposition F W P w)).2)).2
```

**Nested choice**

- first choose `x`
- its specification contains $\exists y$
- choose `y`
- recover `w = x + y`

Choice proves existence-based mathematics; it does not provide an algorithm.

# Category: Morphism Data Plus Three Laws

```
class CategoryStruct (obj : Type u)
    extends Quiver obj where
  id : ∀ X : obj, Hom X X
  comp : ∀ {X Y Z : obj},
    Hom X Y → Hom Y Z → Hom X Z

class Category (obj : Type u)
    extends CategoryStruct obj where
  id_comp : ∀ {X Y} (f : Hom X Y),
    comp (id X) f = f
  comp_id : ∀ {X Y} (f : Hom X Y),
    comp f (id Y) = f
  assoc : ∀ f g h,
    comp (comp f g) h = comp f (comp g h)
```

**Instance checklist**

1. define Hom
2. define identities
3. define composition
4. prove left identity
5. prove right identity
6. prove associativity

**Not copy-paste Lean code; schematic extract only.** Universes, implicit parameters, and some inherited fields are omitted.

Live Lean: D3_11_Category_Homological.lean

# Functor: Map Objects and Morphisms, Preserve Structure

```
structure Functor (C : Type u₁) [Category C]
    (D : Type u₂) [Category D] where
  obj : C → D
  map : ∀ {X Y : C},
    Hom X Y → Hom (obj X) (obj Y)
  map_id : ∀ X,
    map (id X) = id (obj X)
  map_comp : ∀ f g,
    map (comp f g) = comp (map f) (map g)
```

**Notation**

- `F : Functor C D`
- `F.obj X`
- `F.map f`
- `F.comp G`

A functor is not merely an object-level function: it transports morphisms and proves compatibility.

**Not copy-paste Lean code; schematic extract only.** Universes, implicit parameters, and some inherited fields are omitted.

# Natural Transformation: Components Plus Naturality

```
structure NatTrans (F G : Functor C D) where
  app (X : C) : Hom (F.obj X) (G.obj X)
  naturality {X Y : C} (f : Hom X Y) :
    comp (F.map f) (app Y) =
      comp (app X) (G.map f)
```

For every $f : X \to Y$, the two routes around the naturality square agree:

$$F(X) \xrightarrow{F(f)} F(Y) \xrightarrow{\eta_Y} G(Y) \quad = \quad F(X) \xrightarrow{\eta_X} G(X) \xrightarrow{G(f)} G(Y).$$

**How to read it**

- component: `η.app X`
- law: `η.naturality f`
- morphism notation: $\eta : F \Rightarrow G$

**Not copy-paste Lean code; schematic extract only.** Universes, implicit parameters, and some inherited fields are omitted.

# HomologicalComplex: Objects, Differentials, and $d^2 = 0$

```
structure HomologicalComplex
    (c : ComplexShape ι) where
  X : ι → V
  d : ∀ i j, Hom (X i) (X j)
  shape : ∀ i j,
    ¬ c.Rel i j → d i j = 0
  d_comp_d' : ∀ i j k,
    c.Rel i j → c.Rel j k →
      comp (d i j) (d j k) = 0
```

**Mathematical meaning**

- `X i`: the object in degree `i`
- `d i j`: a differential
- `shape`: disallowed arrows vanish
- `d_comp_d'`: consecutive differentials compose to zero

**Not copy-paste Lean code; schematic extract only.** Universes, implicit parameters, and some inherited fields are omitted.

# Example: A One-Object, One-Morphism Category

```
open CategoryTheory

inductive OneObj where
  | star

instance : Category OneObj where
  Hom _ _ := PUnit
  id _ := PUnit.unit
  comp _ _ := PUnit.unit
  id_comp := by
    intro X Y f
    cases f
    rfl
  comp_id := by
    intro X Y f
    cases f
    rfl
  assoc := by
    intro W X Y Z f g h
    cases f; cases g; cases h; rfl
```

Live Lean: D3_11_Category_Homological.lean

**Why the laws are easy**

- only one object
- every hom type is `PUnit`
- only one possible identity
- only one possible composition
- case analysis reduces each law to `rfl`

# Example: The Zero Homological Complex

```
open CategoryTheory
open CategoryTheory.Limits

noncomputable def tinyZeroComplex :
    HomologicalComplex
      (Discrete PUnit) (ComplexShape.up ℕ) where
  X _ := Discrete.mk PUnit.unit
  d _ _ := 0
  shape := by
    intro i j hij
    rfl
  d_comp_d' := by
    intro i j k hij hjk
    simp
```

Live Lean: D3_11_Category_Homological.lean

**Why it works**

- the category `Discrete PUnit` is tiny
- every differential is zero
- illegal arrows are already zero
- zero composed with zero is zero

This is a small construction that exposes the research-level interface.

# A Proposition Can Also Be an Instance

**Data-valued class**

- usually lives in `Type`
- carries operations and computational data
- examples: `Group G`, `Field K`
- changes notation and available theorems

**Prop-valued class**

- lives in `Prop`
- carries a proof for automatic reuse
- examples: `Fact P`, `Decidable P`
- passes mathematical properties through search

The mechanism is the same: Lean searches for an instance. The payload is now evidence rather than algebraic operations.

# Fact P Places a Proposition in Typeclass Search

```
instance : Fact (2 ≤ 5) :=
  ⟨by norm_num⟩

example : 2 ≤ 5 := by
  exact Fact.out

example (n : Nat) [Fact (n ≠ 0)] :
    0 < n := by
  exact Nat.pos_of_ne_zero
    (Fact.out : n ≠ 0)

example : 2 ≤ 5 := by
  haveI : Fact (2 ≤ 5) := ⟨by norm_num⟩
  exact Fact.out
```

Live Lean: `D3_12_Prop_Instance.lean`

**Key operations**

- `Fact P` wraps a proof of P.
- `Fact.out` retrieves it.
- `[Fact P]` passes it implicitly.
- `haveI` installs a local instance.

# A Custom Prop-Valued Class: IsEven

```
class IsEven (n : Nat) : Prop where
  witness : ∃ k, n = 2 * k

instance : IsEven 8 where
  witness := (4, by norm_num)

example [h : IsEven n] :
    ∃ k, n = 2 * k := by
  exact h.witness

example : ∃ k, 8 = 2 * k := by
  exact (inferInstance : IsEven 8).witness
```

**Design pattern**

- class name describes a property
- instance proves that an object has it
- `inferInstance` asks Lean to recover the proof

Use global proposition instances selectively: too many can make search opaque.

# Three Tools for Seeing Instance Search

| `#synth` | `infer_instance` | `haveI` |
| --- | --- | --- |
| Check whether an instance can be synthesized. | Invoke typeclass search inside a proof. | Install a local structure or proposition instance. |

```
#synth Group Sign

example : Group Sign := by
  infer_instance

set_option trace.Meta.synthInstance true in
#synth Group Sign
```

The trace shows candidate instances, recursive obligations, and the successful search path.

Live Lean: `D3_13_Deeper_Instance_Search.lean`

# Read the Weakest Interface a Theorem Needs

```
#check mul_assoc
-- only [Semigroup G] is needed

#check inv_mul_cancel
-- requires [Group G]

example {G : Type*} [Group G] (a b : G) :
    (a * b)⁻¹ = b⁻¹ * a⁻¹ := by
  group
```

**Why weaker is better**

- broader reuse
- fewer unnecessary assumptions
- clearer mathematical dependency
- better alignment with Mathlib style

AI often overspecifies $\mathbb{R}$, `CommGroup`, or `Field` when a weaker class would suffice.

# Structure Transfer Is a General Mathlib Pattern

```
#check equivProd

instance : CommRing PairInt :=
  equivProd.commRing

#check equivRat

instance : Field WrappedQ :=
  equivRat.field
```

**When to look for transfer**

- wrapper types
- products and re-encodings
- subtypes with known closure
- objects related by equivalence or isomorphism

The hard problem becomes proving the equivalence, not replaying every algebraic law.

# AI Error 1: The Statement Is False

### AI draft

```
example (K : Type*) [Field K] (x : K) :
    x / x = 1 := by
  field_simp
```

### Corrected statement

```
example (K : Type*) [Field K]
    (x : K) (hx : x ≠ 0) :
    x / x = 1 := by
  field_simp [hx]
```

Live Lean: `D3_15_AI_Examples.lean`

### Diagnosis

- `x = 0` is a counterexample.
- Lean will not invent a missing hypothesis.
- More automation cannot prove a false statement.
- Repair semantics before tactics.

# AI Error 2: The Algebraic Level Is Wrong

**AI draft**

```
example (G : Type*) [Group G] (a b : G) :
    (a * b)⁻¹ = a⁻¹ * b⁻¹ := by
  group
```

**Correct in every group**

```
example (G : Type*) [Group G] (a b : G) :
    (a * b)⁻¹ = b⁻¹ * a⁻¹ := by
  group
```

**Diagnosis**

- inversion reverses multiplication order
- the original statement needs `CommGroup`
- `group` only proves valid group identities

# AI Error 3: A Plausible API Name Does Not Exist

### Hallucinated theorem

```
example : (Finset.range 5).card = 5 := by
  exact Finset.card_range_five
```

### Search and verify

```
#check Finset.card_range

example : (Finset.range 5).card = 5 := by
  exact Finset.card_range 5
```

### Required evidence

- ask AI to include #check
- use editor completion
- search Mathlib, LeanSearch, or Loogle
- compile every candidate name

# AI Error 4: decide Is Not General Classical Reasoning

## AI draft

```
example (P : Prop) : P ∨ ¬ P := by
  decide
```

## Two valid repairs

```
example (P : Prop) [Decidable P] : P ∨ ¬ P := by
  by_cases h : P
  · exact Or.inl h
  · exact Or.inr h

example (P : Prop) : P ∨ ¬ P := by
  classical
  exact em P
```

## Distinction

- `decide` evaluates a `Decidable` proposition
- finite propositions are good candidates
- unrestricted excluded middle is classical
- computability and classical truth are not identical

# A Good AI Draft: Algebraic Normalization with ring

```
example {R : Type*} [CommRing R]
    (a b c d : R)
    (h1 : c = d * a + b)
    (h2 : b = a * d) :
    c = 2 * a * d := by
  rw [h1, h2]
  ring
```

**Why this is robust**

1. hypotheses expose the substitutions
2. `rw` performs semantic rewriting
3. `ring` handles routine polynomial normalization
4. the final proof is short and kernel checked

AI is most useful when the statement is precise and the remaining subgoal belongs to a well-supported decision procedure.

# A Good AI Draft: Linear Arithmetic with linarith

```
example (x y : ℝ)
    (h1 : x + y = 10)
    (h2 : x - y = 4) :
    x = 7 := by
  linarith
```

**Division of labor**

- the human chooses variables and assumptions
- AI may recognize a linear-arithmetic fragment
- `linarith` produces a proof term
- the kernel checks that proof term

**Do not ask automation to choose the mathematics.** Ask it to close a well-specified local goal after definitions and assumptions are already correct.

# Search Tactics Help During Development

```
example (n : Nat) :
    (Finset.range n).card = n := by
  exact?

example {G : Type*} [Group G] (a : G) :
    a⁻¹ * a = 1 := by
  apply?

example (n : Nat) :
    (Finset.range (n + 1)).card = n + 1 := by
  simp?
```

Live Lean: `D3_15_AI_Examples.lean`

**Recommended workflow**

1. run the suggestion tactic
2. inspect the theorem or simplification it found
3. replace the exploratory command with a stable proof
4. rerun Lean

# A Statement-First AI Workflow

| Stage | Ask AI for | Check in Lean |
|---|---|---|
| **Statement** | variables, typeclasses, necessary assumptions | does it match the intended claim? |
| **Search** | candidate definitions and theorem names with `#check` evidence | do the names exist with the right types? |
| **Proof** | a short proof and local lemmas | does it compile; is the proof stable? |
| **Review** | counterexamples and stronger/weaker variants | are assumptions too strong or too weak? |

**AI proposes. Lean's kernel checks. The mathematician owns the semantics.**

# A Prompt That Constrains the Formalization Task

```
Formalize the following statement in Lean 4 with Mathlib.
First give only the theorem statement, not the proof.

Requirements:
1. List all variables and typeclass assumptions.
2. Identify nonzero, finiteness, or decidability assumptions.
3. Give #check evidence for relevant API names.
4. Do not invent theorem names.
5. If several formal statements are possible, compare them.
```

**Order of work**

1. align objects
2. align structures
3. align assumptions
4. only then prove

# Current Bottlenecks and Practical Repairs

| Problem | Typical symptom | Repair action |
|---|---|---|
| Missing hypothesis | `x / x = 1` lacks `x ≠ 0` | ask for assumptions before proof code |
| Wrong hierarchy | a group theorem needs `mul_comm` | inspect the weakest required class |
| API hallucination | a plausible theorem name is absent | demand `#check` or search evidence |
| Overspecialization | a general theorem is written only over $\mathbb{R}$ | compare nearby Mathlib theorem types |
| Brittle proof | one long tactic breaks after small changes | split lemmas; use local automation |

The main difficulty is usually semantic alignment between natural-language mathematics and the selected formal objects.

# Lean's Trusted Boundary

**Lean checks**

- the formal theorem statement
- whether the proof term has that type
- whether required instances can be synthesized
- definitional equality and kernel reduction

**Lean does not decide for us**

- whether the formal statement matches the original prose
- whether the chosen Mathlib definition is the best model
- whether assumptions are mathematically natural
- whether the result is interesting or useful

The kernel gives a strong guarantee about formal derivation, not an automatic guarantee of informal intent.

# Finset Is the Basic Container for Finite Counting

### Core objects

- `Finset α`: finite elements with no duplicates
- `s.card`: cardinality
- `Finset.range n`: $0, \ldots, n-1$
- `s.filter p`: elements satisfying `p`

### Proof habits

- inspect APIs with `#check`
- use `simp` for structural counting facts
- use `native_decide` for small finite computations
- use `sum_range_succ` to split the final summand

Unlike an arbitrary `Set`, a `Finset` is designed for computation, enumeration, finite sums, and finite products.

# Finset Theorems: Cardinality, Filtering, and Sums

```
open BigOperators
open Finset

#check Finset.card_range
#check Finset.filter
#check Finset.sum_range_succ

example : (Finset.range 5).card = 5 := by
  exact Finset.card_range 5

example :
    ((Finset.range 6).filter
      fun n => n % 2 = 0).card = 3 := by
  native_decide

example (n : Nat) :
    ∑ _i ∈ Finset.range (n + 1), (1 : Nat) = n + 1 := by
  simp
```

Live Lean: D3_14_Finset_SimpleGraph.lean

**Choose the proof mode**

- library theorem for a general fact
- computation for a concrete finite fact
- simplification for a structurally obvious sum

# Finset Theorem: Sum the First $n$ Natural Numbers

```
theorem sum_id (n : Nat) :
    ∑ i ∈ Finset.range (n + 1), i =
      n * (n + 1) / 2 := by
  symm
  apply Nat.div_eq_of_eq_mul_right
    (by norm_num : 0 < 2)
  induction n with
  | zero =>
      simp
  | succ n ih =>
      rw [Finset.sum_range_succ, mul_add 2, ← ih]
      ring

example : ∑ i ∈ Finset.range 5, i = 10 := by
  norm_num [sum_id]
```

**Proof rhythm**

1. handle natural-number division
2. induct on `n`
3. split the final summand
4. rewrite with the induction hypothesis
5. use `ring`

# SimpleGraph = Vertex Type + Adjacency Proposition

| Object | Lean form | Reading |
|---|---|---|
| **Graph** | `G : SimpleGraph V` | `V` is the vertex type |
| **Edge** | `G.Adj u v` | `u` and `v` are adjacent |
| **Undirectedness** | `symm` field | adjacency is symmetric |
| **No loops** | `loopless` field | never `G.Adj v v` |
| **Small finite graph** | `V = Fin n` | vertices can be enumerated by `fin_cases` |

Formalizing a graph starts by specifying its vertex type and adjacency predicate, not by drawing a picture.

# Define a Three-Vertex Path Graph

```
def path3 : SimpleGraph (Fin 3) where
  Adj i j :=
    (i = 0 ∧ j = 1) ∨
    (i = 1 ∧ j = 0) ∨
    (i = 1 ∧ j = 2) ∨
    (i = 2 ∧ j = 1)
  symm := by
    unfold Symmetric
    intro i j h
    fin_cases i <;> fin_cases j <;> simp_all
  loopless := by
    constructor
    intro i h
    fin_cases i <;> simp_all
```

**Mathematical graph**

$$0 — 1 — 2$$

- adjacency lists both directions
- `symm` proves undirectedness
- `loopless` rules out `i -- i`

Live Lean: D3_14_Finset_SimpleGraph.lean

# Small Graph Theorems: Unfold, Enumerate, Simplify

```
example : path3.Adj 0 1 := by
  simp [path3]

example : path3.Adj 1 2 := by
  simp [path3]

example : ¬ path3.Adj 0 2 := by
  simp [path3]

example (i : Fin 3) : ¬ path3.Adj i i := by
  fin_cases i <;> simp [path3]
```

**Proof pattern**

1. inspect `Adj`
2. unfold the concrete graph
3. enumerate finite vertices if needed
4. let `simp` close propositional cases

# More SimpleGraph Theorems: Abstract API and decide

```
example (i j : Fin 3) (h : path3.Adj i j) :
    i ≠ j := by
  exact SimpleGraph.ne_of_adj path3 h

example : path3.Adj 0 1 ∧ ¬ path3.Adj 0 2 := by
  constructor
  · simp [path3]
  · simp [path3]

example : (⊤ : SimpleGraph (Fin 4)).Adj 0 3 := by
  decide
```

**Three proof styles**

- abstract theorem: `ne_of_adj`
- concrete definition: `simp [path3]`
- finite decidable fact: `decide`

Graph proofs move between the library interface and concrete adjacency definitions.

# A Classroom Checklist for Writing Instances

1. Identify the object type: what are `G`, `R`, `K`, `V`, or `C`?
2. Choose the intended interface: `AddGroup`, `Group`, `CommRing`, `Field`, `Module`, or `Category`?
3. Use `#print`, `#check`, and elaboration errors to discover missing fields.
4. For a substructure, define the carrier and prove closure under every relevant operation.
5. For a structure, define operations first and prove laws second.
6. Confirm success with `#synth`; test the interface with a small theorem.
7. Read theorem types to avoid assumptions stronger than necessary.
8. Let AI draft candidates, but finish with Lean compilation and human semantic review.

# Live Demonstration Map

**Algebra and linear structure**

1. `D3_02`: AddGroup vs Group
2. `D3_03--05`: subtype and subgroup
3. `D3_06--08`: ring, subring, field
4. `D3_09--10`: module, submodule, choice

**Advanced interfaces and AI**

1. `D3_11`: categories and complexes
2. `D3_12--13`: proposition instances and search
3. `D3_14`: Finset and SimpleGraph
4. `D3_15`: AI drafts and repairs

Slides explain the mathematical architecture; the Lean files show every line accepted by the kernel.

`Open the integrated classroom file: D3_Demo.lean`