BGSMath Graduate Course: Formalizing Mathematics in Lean with AI

Session 1: Lean and Mathlib

Michail Karatarakis

10 November 2026

Radboud University Nijmegen & Axiomatic AI

The course

	Session	Content
1	Lean and Mathlib	What formalizing means, and does not mean; first proofs; finding things in Mathlib; reading a Lean statement
2	From paper to Lean	Writing definitions and statements; the usual mistakes; drafting statements with AI and checking them
3	AI tools	What AI did this year; an agent proving a statement, live; checking what it hands back
4	Your own theorem	Statements sent in by participants, formalized live; how to do your own at home

Each session is a lecture, with me doing things live in Lean. You don't need a laptop here. At home, you redo what I did and try the exercises.

What formalizing means

Formalizing a piece of mathematics means writing it, definitions, statements and proofs, in a language a computer can read, and having the computer check every step.

- The computer here is Lean. It checks a proof all the way down to the axioms, and it won't accept a step it can't verify.
- You don't start from nothing. Mathlib, the shared library, already has most of the basic definitions and theorems, and quite a few advanced ones.
- What you end up with is a statement, and a proof of it that Lean accepts.

> A formal proof is a proof that a computer has checked, line by line.

What formalizing does not mean

I want to be clear about what this does *not* give you, because most misunderstandings start here.

- **It doesn't guarantee the theorem is the one you meant.** Lean checks the proof against the statement as written. If the statement differs from the paper, Lean has no way of knowing. Checking that is your job, and it's most of what we'll do today.
- **The computer doesn't find the proofs.** Automation and AI can propose them. Lean only checks.
- **It doesn't replace the paper.** A formal proof tells you that something is true, not why. The ideas stay on paper.
- **You don't need to be a programmer.** I'll assume no Lean and no programming. What you need is something you already do: read carefully.
- **It isn't all or nothing.** You can state a theorem without proving it, or prove it assuming a step you can't do yet. Both are useful.

Why now

Why now? Because in the last twelve months, AI has written a large part of the Lean code in several major formalizations. Here are four.

Fermat's Last Theorem	A complete Lean proof, produced by Claude and drawing on Kevin Buzzard's FLT project.
Sphere packing	Viazovska's theorems in dimensions 8 and 24, completed with Math Inc's Gauss from a project started by Hariharan and Viazovska. Gauss's part is disputed (session 3).
Unit distances	An AI-found disproof of Erdős's unit distance conjecture, formalized in full by Boris Alexeev with ChatGPT.
Navier–Stokes	OpenAI announced Navier–Stokes blowup with forcing, in Lean. Whether it matches the Millennium Prize statement needs a human check (Hales).

Anthropic (Sep 2026); Math, Inc. (Feb 2026); K. Buzzard, Xena blog (Jul 2026); OpenAI (Sep 2026).

The one idea of today

If you remember one sentence from today, make it this one:

> Lean checks the **proof**. You check the **statement**.

An AI can now hand you a proof that Lean accepts. Lean will tell you the proof is correct. It won't tell you whether the theorem is the one you meant. That part is yours, and it's what this course is about. So I'll spend less time on how to write Lean, and more on how to read it.

Tom Hales put it like this last month, on Terence Tao's blog

"A proof in Lean should not be accepted until a human audit is performed to ensure statement fidelity. [...] This task is generally massively easier than checking the proof itself."

The game

```
theorem fermat (n : ℕ) (hn : n ≥ 3) (a b c : ℕ)
    (ha : a ≠ 0) (hb : b ≠ 0) (hc : c ≠ 0) :
    a ^ n + b ^ n ≠ c ^ n := by
  sorry
```

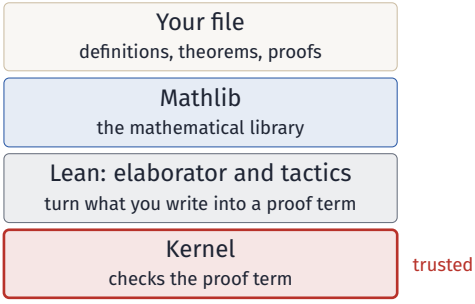
- This is Fermat's Last Theorem, in Lean.
- `sorry` is where the proof should be. Lean accepts the file, but warns you that something is missing.
- Have a look at the hypotheses. Why `a ≠ 0`? Because $0^n + b^n = b^n$, and we don't want that to count.

That's the whole game: write the statement, then replace `sorry` by a proof Lean accepts. Today is about the rules.

What Lean is

A proof assistant

- You write definitions, statements and proofs in a formal language, and Lean checks every step.
- At the bottom sits a small program, the **kernel**, which has the final word. Everything above it, tactics, automation, AI, can be as clever or as wrong as it likes: whatever it produces goes through the kernel.
- Lean is also a programming language, which is handy: you can compute with your definitions and try them on examples.



Mathlib

139,422 definitions

290,718 theorems

772 contributors

- One library, written by the community and reviewed by maintainers: algebra, number theory, analysis, measure theory and probability, topology, geometry, combinatorics, category theory, and so on.
- One thing to get used to: everything is stated as generally as is convenient. Where you would write $\mathbb{R}$, Mathlib often has an ordered field, or just a ring.

Mathlib statistics page, October 2026.

Getting started

In the browser

- live.lean-lang.org
- Mathlib included; nothing to install.
- Enough for sessions 1 and 2.

On your laptop

- VS Code with the *Lean 4* extension, and a project with Mathlib.
- The setup sheet has the steps; for sessions 3 and 4.

The course files

- The code on the slides, one file per session.
- Exercises with `sorry` to fill in, at home. Solutions after each session.

> The **infoview**, the panel next to your code, shows what Lean knows at the cursor. Keep it open.

All files: github.com/mkaratarakis/from-paper-to-proof. If opening a file makes Lean start building Mathlib, stop it and run `lake exe cache get` in the project folder.

First steps

First contact

live in Lean

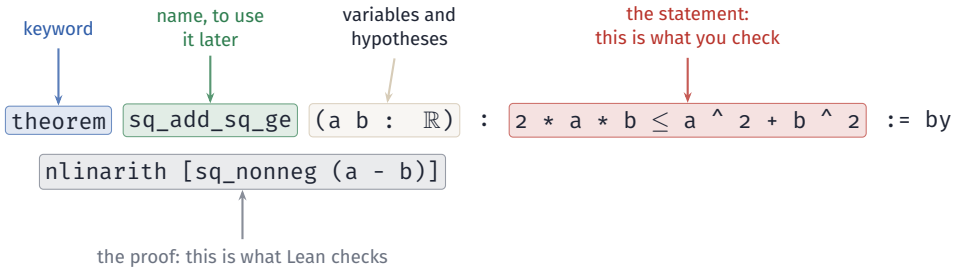
```
#eval 2 + 2                -- 4
#check (2 : ℝ)             -- 2 : ℝ
#check Nat.Prime           -- Nat.Prime (p : ℕ) : Prop
#check Nat.Prime 7         -- Nat.Prime 7 : Prop
```

- `#eval` computes something; `#check` tells you what type an expression has.
- `Nat.Prime 7` is a *statement*: its type is `Prop`. Lean knows what it means, but nothing has been proved yet.

```
example : 2 + 2 = 4 := rfl
```

- And this is a *proof*, of `2 + 2 = 4`. `rfl` says: compute both sides, they come out the same.

Anatomy of a theorem



- `example` is a theorem you don't bother to name.
- `by` says: what follows is a list of instructions, *tactics*, that build the proof step by step.

From paper to Lean

On paper

Claim. For all real a, b, $2ab \leq a^2 + b^2$.

Proof. We have $(a - b)^2 \geq 0$, and

$$\begin{aligned} 2ab &= a^2 + b^2 - (a - b)^2 \\ &\leq a^2 + b^2. \end{aligned}$$ □

In Lean

```
example (a b : ℝ) :
    2 * a * b ≤ a ^ 2 + b ^ 2 := by
  have h : 0 ≤ (a - b) ^ 2 := sq_nonneg _
  calc 2 * a * b
      = a ^ 2 + b ^ 2 - (a - b) ^ 2 := by ring
    _ ≤ a ^ 2 + b ^ 2 := by linarith
```

- `have` records an intermediate fact; `calc` chains (in)equalities, each step with its own justification.
- `ring` proves identities in commutative rings; `linarith` does linear arithmetic with the hypotheses.
- One-line version: `nlinarith [sq_nonneg (a - b)]`, with the same hint as on paper.

Tactic proofs

Reading the infoview

Lean infoview

```
m n : ℕ
hm : Even m
hn : Even n
⊢ Even (m + n)
```

- Above the line is what you **have**: the variables, and the hypotheses with their names.
- After ⊢ is what you still have to **prove**: the *goal*.
- Every tactic changes this picture a little. When no goals are left, you're done.

```
#check Even  -- Even {α : Type u_2} [Add α] (a : α) : Prop
#print Even  -- fun {α} [Add α] a => ∃ r, a = r + r
```

- In Mathlib, `Even a` *means* $\exists r,\ a = r + r$. Whenever you wonder what a name means, `#print` it.

A proof, step by step

live in Lean

```
example (m n : ℕ)
    (hm : Even m) (hn : Even n) :
    Even (m + n) := by
  obtain ⟨a, ha⟩ := hm
  obtain ⟨b, hb⟩ := hn
  use a + b
  rw [ha, hb]
  ring
```

Lean infoview

```
m n : ℕ
hm : Even m
hn : Even n
⊢ Even (m + n)
```

Where we start: two even numbers, and the goal.

A proof, step by step

live in Lean

```
example (m n : ℕ)
    (hm : Even m) (hn : Even n) :
    Even (m + n) := by
  obtain ⟨a, ha⟩ := hm
  obtain ⟨b, hb⟩ := hn
  use a + b
  rw [ha, hb]
  ring
```

Lean infoview

```
m n : ℕ
hn : Even n
a : ℕ
ha : m = a + a
⊢ Even (m + n)
```

`obtain` unpacks `hm : ∃ r, m = r + r`. We get a witness `a` and the equation `ha`.

A proof, step by step

live in Lean

```
example (m n : ℕ)
    (hm : Even m) (hn : Even n) :
    Even (m + n) := by
  obtain ⟨a, ha⟩ := hm
  obtain ⟨b, hb⟩ := hn
  use a + b
  rw [ha, hb]
  ring
```

Lean infoview

```
m n a : ℕ
ha : m = a + a
b : ℕ
hb : n = b + b
⊢ Even (m + n)
```

The same for `hn`.

A proof, step by step

live in Lean

```
example (m n : ℕ)
    (hm : Even m) (hn : Even n) :
    Even (m + n) := by
  obtain ⟨a, ha⟩ := hm
  obtain ⟨b, hb⟩ := hn
  use a + b
  rw [ha, hb]
  ring
```

Lean infoview

```
case h
m n a : ℕ
ha : m = a + a
b : ℕ
hb : n = b + b
⊢ m + n = a + b + (a + b)
```

To prove `∃ r, m + n = r + r`, we propose `r := a + b`. (`case h` is just a label; ignore it.)

A proof, step by step

live in Lean

```
example (m n : ℕ)
    (hm : Even m) (hn : Even n) :
    Even (m + n) := by
  obtain ⟨a, ha⟩ := hm
  obtain ⟨b, hb⟩ := hn
  use a + b
  rw [ha, hb]
  ring
```

Lean infoview

```
case h
m n a : ℕ
ha : m = a + a
b : ℕ
hb : n = b + b
⊢ a + a + (b + b) = a + b + (a + b)
```

`rw` rewrites `m` and `n` using the two equations.

A proof, step by step

live in Lean

```
example (m n : ℕ)
    (hm : Even m) (hn : Even n) :
    Even (m + n) := by
  obtain ⟨a, ha⟩ := hm
  obtain ⟨b, hb⟩ := hn
  use a + b
  rw [ha, hb]
  ring
```

Lean infoview

No goals

`ring` does the algebra. **No goals**: Lean has accepted the proof.

The short way

```
example (m n : ℕ) (hm : Even m) (hn : Even n) :
    Even (m + n) :=
  hm.add hn
```

- Mathlib already knows this, under the name `Even.add`. Writing `hm.add hn` applies it to our two hypotheses.
- Most short proofs look like this. The work isn't in writing the proof, it's in finding the right lemma. We'll get to that in a moment.

At home Type the step-by-step proof yourself and watch the infoview after each line. Then `Exercises1.lean`, parts 1 and 2.

What Lean actually checks

```
theorem small (x : ℝ) (h : 0 < x) :
    0 < x + 1 := by
  linarith

#print small
```

- A tactic is a program that *builds* a proof term. On the right: the one `linarith` built here.
- The kernel checks the term, not the tactic. So tactics, automation and AI can be as clever or as wrong as they like.

Lean says

```
theorem small : ∀ (x : ℝ), 0 < x → 0 < x + 1 :=
fun x h =>
  lt_of_not_ge fun a =>
    Mathlib.Tactic.Linarith.lt_irrefl
      (Eq.mp
        (congrArg (fun _a => _a < 0)
          (Mathlib.Tactic.Ring.of_eq
            (Mathlib.Tactic.Ring.Common.add_congr
              (Mathlib.Tactic.Ring.Common.add_congr
                (Mathlib.Tactic.Ring.Common.neg_congr
                  (Mathlib.Tactic.Ring.cast_pos (Mathlib.Meta.NormNum.isNat_ofNat ℝ
            Nat.cast_one))
                  (Mathlib.Tactic.Ring.Common.neg_add
                    (Mathlib.Meta.NormNum.IsInt.to_raw_eq
                      (Mathlib.Meta.NormNum.isInt_neg (Eq.refl Neg.neg)
                        (Mathlib.Meta.NormNum.IsNat.to_isInt (Mathlib.Meta.NormNum.IsNat.of_raw ℝ
             1))
                        (Eq.refl (Int.negOfNat 1))))
                    Mathlib.Tactic.Ring.Common.neg_zero))
                (Mathlib.Tactic.Ring.Common.sub_congr
                  (Mathlib.Tactic.Ring.cast_zero (Mathlib.Meta.NormNum.isNat_ofNat ℝ
            Nat.cast_zero))
                  (Mathlib.Tactic.Ring.Common.atom_pf x rfl
                    (Eq.mpr
                      (id
                        (congrArg (fun _a => x ^ Nat.rawCast 1 * Nat.rawCast 1 = x ^
            Nat.rawCast 1 * _a) (Eq.symm rfl)))
                        (Eq.refl (x ^ Nat.rawCast 1 * Nat.rawCast 1))))

  .
  .
  .  (68 lines in all)
```

Finding things in Mathlib

Ask Lean

```
example (a b : ℝ) :
    2 * a * b ≤ a ^ 2 + b ^ 2 := by
  exact?
```

Lean says

```
Try this:
  [apply] exact two_mul_le_add_sq a b
```

- `exact?` searches Mathlib for a lemma that closes the goal.
- `apply?` looks for lemmas whose conclusion fits; `rw?` looks for rewrites.
- Click the suggestion and it replaces the search.

Before you prove it, search for it Our `sq_add_sq_ge` from a few slides ago was already in Mathlib, as `two_mul_le_add_sq`. This will happen to you a lot.

When the search fails

live in Lean

Without a hypothesis

```
example (x : ℝ) :
    Real.sqrt x ^ 2 = x := by
  exact?
-- error: could not close the goal
```

```
example (x : ℝ) (hx : 0 ≤ x) :
    Real.sqrt x ^ 2 = x := by
  exact?
```

Lean says

```
Try this:
  [apply] exact Real.sq_sqrt hx
```

- The first statement is false: in Lean, $\sqrt{x} = 0$ when $x < 0$, so no lemma can prove it.
- A failed search tells you something. Before you search harder, ask whether the statement is actually true.

Search engines for Mathlib

Tool	What you give it	Example
`exact?`	a goal in your file	(previous slide)
Loogle	a pattern, with names and _ for holes	`Real.sqrt, _ * _ = _`
LeanSearch	a sentence in English	*square root of a product*
Mathlib docs	a name or a word	`two_mul_le_add_sq`

- loogle.lean-lang.org leansearch.net Mathlib documentation
- And in the editor: hover over a name to see its statement, or *Go to definition* to read its source.

Names follow the statement

```
#check mul_comm          -- a * b = b * a
#check mul_pos           -- 0 < a → 0 < b → 0 < a * b
#check add_le_add        -- a ≤ b → c ≤ d → a + c ≤ b + d
#check lt_of_le_of_lt    -- a ≤ b → b < c → a < c
#check Nat.Prime.two_le  -- p.Prime → 2 ≤ p
```

- A name reads the statement aloud: `mul_comm` is about `*` being commutative, `mul_pos` says a product is positive.
- For implications, the conclusion comes first: `lt_of_le_of_lt` gives you a `<` *of* a $\leq$ and a `<`.
- Dot notation: if `hp : p.Prime`, then `hp.two_le` is a proof of `2 ≤ p`.
- So you can often *guess* a name, and `#check` tells you whether you guessed right.

The library also has big theorems

```
#check irrational_sqrt_two          -- Irrational √2
#check Nat.exists_infinite_primes   -- ∀ n, ∃ p, n ≤ p ∧ p.Prime
#check Complex.exists_root          -- 0 < degree f → ∃ z, f.IsRoot z
```

- $\sqrt{2}$ is irrational, there are infinitely many primes, the fundamental theorem of algebra, and much more.
- Each of these is a name you can use in your own proofs, like any other lemma.

Reading a Lean statement

Three kinds of brackets

```
#check mul_comm
-- mul_comm.{u_1} {G : Type u_1} [CommMagma G] (a b : G) : a * b = b * a
```

`(a b : G)`	**explicit**: you supply them, `mul_comm x y`.
`{G : Type u_1}`	**implicit**: Lean works them out from the explicit ones.
`[CommMagma G]`	**instance**: a structure on `G`, found automatically. Here: a commutative multiplication.

So `mul_comm` works for $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$, polynomial rings, … but not for square matrices: Lean finds no `CommMagma` instance for them, and `mul_comm A B` is rejected. Which is what you'd want.

Do not panic at long signatures

Lean says

```
@two_mul_le_add_sq : ∀ {R : Type u_1} [inst : CommSemiring R]
  [inst_1 : LinearOrder R] [ExistsAddOfLE R] [MulPosStrictMono R]
  [AddLeftReflectLE R] [AddLeftMono R] (a b : R),
  2 * a * b ≤ a ^ 2 + b ^ 2
```

- Read it as: *for any ordered commutative semiring R and $a, b \in R$, $2ab \leq a^2 + b^2$.*
- The square brackets are the exact list of what the proof needs. $\mathbb{R}$ satisfies all of them, so you can ignore them.
- Mathlib states results like this so that one lemma works for $\mathbb{N}$, $\mathbb{Q}$, $\mathbb{R}$, … at once.

Junk values

```
example : (5 : ℕ) - 7 = 0 := by norm_num
example : (7 : ℕ) / 2 = 3 := by norm_num
example : (1 : ℝ) / 0 = 0 := by norm_num
example : Real.sqrt (-1) = 0 := by norm_num
example : Real.log 0 = 0 := by norm_num
```

- In Lean, every function is defined everywhere. Where mathematics says “undefined”, Lean quietly returns some value, usually 0.
- So subtraction on $\mathbb{N}$ stops at 0, and division on $\mathbb{N}$ rounds down.

In Mathlib’s theorems this is harmless: a lemma about a/b will ask you for $b \neq 0$. It bites when *you* write a statement and forget that hypothesis.

Junk values in analysis

```
-- The harmonic series diverges, so its sum is 0.
example : ∑' n : ℕ, (1 / n : ℝ) = 0 :=
  tsum_eq_zero_of_not_summable Real.not_summable_one_div_natCast

-- The empty set of natural numbers has infimum 0.
example : sInf (∅ : Set ℕ) = 0 := Nat.sInf_empty

-- Every function is periodic with period 0.
example (f : ℕ → ℕ) : Function.Periodic f 0 := by
  simp [Function.Periodic]
```

- A theorem about $\sum' f$ that forgets `Summable f` may be true only because the sum is 0.
- "The least n with property B" is 0 when there is no such n.
- "f has minimal period N", written as "no smaller period", fails at 0: every f has period 0.

Coercions

```
example : ((7 : ℕ) : ℝ) / 2 = 3.5 := by norm_num
example : (((7 : ℕ) / 2 : ℕ) : ℝ) = 3 := by norm_num
```

- `((7 : ℕ) : ℝ)` is the natural number 7, seen as a real number. Lean prints the coercion as an arrow: ↑n.
- Where the arrow sits changes the meaning. Divide in $\mathbb{R}$ and you get 3.5; divide in $\mathbb{N}$ first and you get 3.
- Whenever a statement mixes $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{R}$, look for the arrows.

Spot the problem (1)

Claim

For all real a, b, $\dfrac{a}{b} \cdot b = a$.

Spot the problem (1)

Claim

For all real a, b, $\dfrac{a}{b} \cdot b = a$.

```
-- Wrong as stated: take a = 1, b = 0.
example : ¬ ∀ a b : ℝ, a / b * b = a := by
  intro h
  have := h 1 0
  norm_num at this

-- The right statement needs a hypothesis.
example (a b : ℝ) (hb : b ≠ 0) : a / b * b = a :=
  div_mul_cancel₀ a hb
```

- As written, the claim is **false** in Lean, and Lean can prove that it's false.
- On paper, $b \neq 0$ usually goes without saying. In Lean it has to be said.

Spot the problem (2)

Claim

For every natural number n, $(n-1)+1=n$.

Spot the problem (2)

Claim

For every natural number n, $(n - 1) + 1 = n$.

```
-- Wrong as stated: n = 0 gives 0 - 1 + 1 = 1.
example : ¬ ∀ n : ℕ, n - 1 + 1 = n := by
  intro h
  have := h 0
  omega
```

- On $\mathbb{N}$, $0 - 1 = 0$, so the claim fails at $n = 0$.
- Over $\mathbb{Z}$ the same statement is true. Which one did the paper mean? Usually the paper doesn't say.

Spot the problem (3)

```
theorem everything_is_42 (x : ℝ) (h₁ : 0 < x) (h₂ : x < 0) :
    x = 42 := by
  linarith
```

Spot the problem (3)

```
theorem everything_is_42 (x : ℝ) (h₁ : 0 < x) (h₂ : x < 0) :
    x = 42 := by
  linarith
```

Lean accepts this The hypotheses $0 < x$ and $x < 0$ contradict each other, so *anything* follows from them. The theorem is true, and it says nothing.

- A theorem with inconsistent hypotheses is proved for free, and AI provers find such proofs very quickly.
- The check is simple: can the hypotheses hold at the same time? Try to give one example.

Spot the problem (4)

```
def Cont (f : ℝ → ℝ) : Prop :=
  ∀ x, ∀ ε > 0, ∃ δ > 0, ∀ y, |y - x| < δ → |f y - f x| < ε

def UnifCont (f : ℝ → ℝ) : Prop :=
  ∀ ε > 0, ∃ δ > 0, ∀ x y, |y - x| < δ → |f y - f x| < ε
```

Spot the problem (4)

```
def Cont (f : ℝ → ℝ) : Prop :=
  ∀ x, ∀ ε > 0, ∃ δ > 0, ∀ y, |y - x| < δ → |f y - f x| < ε

def UnifCont (f : ℝ → ℝ) : Prop :=
  ∀ ε > 0, ∃ δ > 0, ∀ x y, |y - x| < δ → |f y - f x| < ε
```

- The only difference is where `∀ x` sits.
- `Cont f` is continuity: δ may depend on x. `UnifCont f` is uniform continuity: one δ for every x.
- $x \mapsto x^2$ satisfies the first and not the second.

> Quantifier order is exactly the kind of thing a quick reading misses.

Quiz: which one is Goldbach's conjecture?

```
def Goldbach₁ : Prop :=
  ∀ n : ℕ, 2 < n → Even n → ∃ p q, p.Prime ∧ q.Prime ∧ n = p + q
def Goldbach₂ : Prop :=
  ∀ n : ℕ, Even n → ∃ p q, p.Prime ∧ q.Prime ∧ n = p + q
def Goldbach₃ : Prop :=
  ∀ n : ℕ, 2 < n → Even n → ∃ p q, p.Prime ∨ q.Prime ∧ n = p + q
```

Goldbach's conjecture

Every even integer greater than 2 is the sum of two primes.

Quiz: which one is Goldbach's conjecture?

```
def Goldbach₁ : Prop :=
  ∀ n : ℕ, 2 < n → Even n → ∃ p q, p.Prime ∧ q.Prime ∧ n = p + q
def Goldbach₂ : Prop :=
  ∀ n : ℕ, Even n → ∃ p q, p.Prime ∧ q.Prime ∧ n = p + q
def Goldbach₃ : Prop :=
  ∀ n : ℕ, 2 < n → Even n → ∃ p q, p.Prime ∨ q.Prime ∧ n = p + q
```

Goldbach$_2$ forgot $n > 2$, and is false:

```
-- Goldbach₂ is false: 0 is even, and not a sum of two primes.
example : ¬ Goldbach₂ := by
  intro h
  obtain ⟨p, q, hp, -, hpq⟩ := h 0 ⟨0, rfl⟩
  have := hp.two_le
  omega
```

Quiz: which one is Goldbach's conjecture?

```
def Goldbach₁ : Prop :=
  ∀ n : ℕ, 2 < n → Even n → ∃ p q, p.Prime ∧ q.Prime ∧ n = p + q
def Goldbach₂ : Prop :=
  ∀ n : ℕ, Even n → ∃ p q, p.Prime ∧ q.Prime ∧ n = p + q
def Goldbach₃ : Prop :=
  ∀ n : ℕ, 2 < n → Even n → ∃ p q, p.Prime ∨ q.Prime ∧ n = p + q
```

Goldbach$_3$ is true, for a silly reason: $\vee$ binds less tightly than $\wedge$, so it says "p is prime, or (q is prime and $n = p + q$)".

```
-- Goldbach₃ holds for a silly reason: `∨` binds weaker than `∧`.
example : Goldbach₃ := fun _ _ _ => ⟨2, 0, Or.inl Nat.prime_two⟩
```

Quiz: which one is Goldbach's conjecture?

```
def Goldbach₁ : Prop :=
  ∀ n : ℕ, 2 < n → Even n → ∃ p q, p.Prime ∧ q.Prime ∧ n = p + q
def Goldbach₂ : Prop :=
  ∀ n : ℕ, Even n → ∃ p q, p.Prime ∧ q.Prime ∧ n = p + q
def Goldbach₃ : Prop :=
  ∀ n : ℕ, 2 < n → Even n → ∃ p q, p.Prime ∨ q.Prime ∧ n = p + q
```

Goldbach$_1$ is the conjecture. Nobody knows a proof, so there's nothing for Lean to check. But we can *state* it, and make sure the statement is right. That's already worth something.

It happens in practice

This isn't a classroom worry. Boris Alexeev formalized a batch of Erdős problems with AI last year, and wrote up what went wrong:

Problem 480

"The formalization of Problem 480 said `m ≠ 0` *but meant to say* `n ≠ 0`*."* The AI found a counterexample that used the typo, together with a junk value.

Problem 56

"Aristotle found a completely trivial counterexample with tiny parameters like 2. It turns out that the conjecture was missing a hypothesis, somewhat implicitly."

> "Misformalization is a big issue and was more frequent than I had expected."

B. Alexeev, guest post on K. Buzzard's Xena blog, 5 December 2025.

Trust, but check: `sorry` and axioms

```
theorem not_proved : 1 = 2 := by sorry

#print axioms not_proved
-- depends on axioms: [sorryAx]
#print axioms sq_add_sq_ge
-- depends on axioms: [propext, Classical.choice, Quot.sound]
```

- `sorry` closes any goal, with a warning. A proof that contains one isn't a proof.
- `#print axioms` lists everything a theorem ultimately rests on.
- Three axioms are standard and fine: `propext`, `Classical.choice`, `Quot.sound`. `sorryAx`, or an axiom you've never heard of, is not.

> Whenever an AI hands you a proof: read the statement, then `#print axioms`.

Reading a statement: a checklist

1. **Types.** $\mathbb{N}$, $\mathbb{Z}$ or $\mathbb{R}$? Where are the coercion arrows?
2. **Junk values.** Is there a division, a subtraction on $\mathbb{N}$, a square root, a logarithm, an infinite sum, an infimum? Is the case where it is "undefined" excluded?
3. **Hypotheses.** Are they all needed? Can they hold together?
4. **Quantifiers.** In which order? What may depend on what?
5. **Precedence.** How do $\wedge$, $\vee$, $\to$ group? Add brackets if unsure.
6. **Definitions.** Does each name mean what you think? Use `#print` and hover.
7. **Proof.** No `sorry`, only standard axioms.

At home

At home

`Exercises1.lean`, in the browser or on your laptop. Four parts:

1. Logic: `intro`, `use`, `obtain`, …
2. Automation: pick the tactic from the table.
3. Search: find the Mathlib lemma.
4. Reading: true, false, or true for a silly reason? Decide, then prove it or its negation.

- Start by redoing the three demos; the code is in `Session1.lean`.
- The tactic tables are at the end of these slides.
- Stuck? Read the infoview first, then ask: on the Lean Zulip, or at the next session.
- Want more? The Natural Number Game.

> Every exercise has two ratings: how hard to *state*, and how hard to *prove*. With AI around, the first is what matters.

Next time: from paper to Lean

- Writing your own definitions and statements.
- The mistakes everyone makes at first, and how to catch them.
- Letting an AI draft a statement, and checking what it gives you.

> Before Thursday: think of a result from your own work that you'd like to see in Lean. Short and elementary is best. In session 4, I'll formalize some of them live.

Resources

- J. Avigad and P. Massot, *Mathematics in Lean*. The standard introduction for mathematicians.
- J. Avigad et al., *Theorem Proving in Lean 4*. How Lean works underneath.
- The Natural Number Game. Lean as a puzzle game, in the browser.
- The Mathlib documentation, Loogle and LeanSearch.
- The Lean Zulip chat, stream *#new members*. Questions of every level are welcome, really.

For the exercises

Logic: the tactics you need

	to prove it (in the goal)	**to use it** (a hypothesis `h`)
`P → Q`	`intro hp`	`apply h`, or `h hp`
`∀ x, P x`	`intro x`	`h a`, or `specialize h a`
`∃ x, P x`	`use a`	`obtain ⟨x, hx⟩ := h`
`P ∧ Q`	`constructor`, or `exact ⟨hp, hq⟩`	`obtain ⟨hp, hq⟩ := h`
`P ∨ Q`	`left` or `right`	`rcases h with hp | hq`
`¬ P`	`intro hp` (then prove `False`)	`exact absurd hp h`, `contradiction`
`P ↔ Q`	`constructor`	`rw [h]`, or `h.1` and `h.2`
`a = b`	`rfl`, `ring`, `simp`, …	`rw [h]`

And always: `exact h` when `h` is exactly the goal.

Logic: three small examples

```
example : ∀ n : ℕ, ∃ m, n < m := by
  intro n
  use n + 1
  omega
```

```
example (P Q : Prop) (h : P ∧ Q) :
    Q ∧ P := by
  obtain ⟨hp, hq⟩ := h
  exact ⟨hq, hp⟩
```

```
example (P Q : Prop) (h : P ∨ Q) :
    Q ∨ P := by
  rcases h with hp | hq
  · right
    exact hp
  · left
    exact hq
```

- `omega` decides linear arithmetic over $\mathbb{N}$ and $\mathbb{Z}$; here it proves `n < n + 1`.
- `⟨hq, hp⟩` builds a proof of a conjunction from its two halves.
- `·` focuses on one goal at a time. Indentation matters.

Automation

```
example (a b : ℝ) : (a + b) ^ 2 = a ^ 2 + 2 * a * b + b ^ 2 := by ring
example (x y : ℝ) (h₁ : x < y) (h₂ : y < 3) : x < 3 := by linarith
example (x : ℝ) : 0 < x ^ 2 + 1 := by positivity
example : Nat.Prime 97 := by norm_num
example (n : ℕ) (h : n % 4 = 3) : n % 2 = 1 := by omega
example : Continuous (fun x : ℝ => x ^ 2 + Real.sin x) := by fun_prop
example (a b c : ℝ) (h : a ≤ b) (hc : 0 ≤ c) : a * c ≤ b * c := by gcongr
example (xs : List ℕ) : (xs ++ []).length = xs.length := by simp
```

`ring`	identities in commutative rings	`omega`	linear arithmetic over $\mathbb{N}$, $\mathbb{Z}$
`linarith`	linear (in)equalities from hypotheses	`fun_prop`	continuity, measurability, …
`positivity`	goals `0 < e`, `0 ≤ e`	`gcongr`	“apply the same operation to both sides”
`norm_num`	numerical facts	`simp`	simplify using a large set of rewrite rules

Propositions are types

Mathematics	Lean	Read as
a statement P	`P : Prop`	P is a proposition
a proof of P	`h : P`	h is a proof of P
$P \Rightarrow Q$	`P → Q`	a function from proofs of P to proofs of Q
for all $x \in \mathbb{R}$, $P(x)$	`∀ x : ℝ, P x`	a function from x to proofs of $P(x)$
the theorem is true	`theorem t : P := proof`	Lean accepted a term of type P

You'll rarely need this explicitly. It explains why hypotheses are written `(h : x < 0)`: a hypothesis is just a named proof you've been given.