Lean 4 for Theorem Proving — A Crash Course

> **Goal**: Understand enough Lean 4 to read proof states, understand tactics, and know what your RL agent is doing. No prior Lean experience assumed.

Lesson 1: What Is Lean?

Lean is two things at once:

1. **A programming language** (like Haskell or OCaml — functional, typed)
2. **A proof assistant** (you write mathematical proofs and Lean checks them)

The key idea: in Lean, **a proof is a program** and **a theorem is a type**.

When you write:

```lean
theorem my_thm : 1 + 1 = 2 := by norm_num
```

You're saying:

- `my_thm` — the name of the theorem
- `1 + 1 = 2` — the **type** (what you're proving)
- `by norm_num` — the **proof** (a tactic that constructs the proof term)

Lean's kernel checks that the proof term actually has the claimed type. If it does, the theorem is accepted. If not, you get an error. There is no way to cheat — the kernel is the final judge.

Lesson 2: Types and Terms

Everything in Lean has a type. Here are the basics:

```lean
```

```
-- Natural numbers
#check 0          -- 0 : Nat
#check 42         -- 42 : Nat

-- Booleans
#check true       -- true : Bool
#check false      -- false : Bool

-- Propositions (things that can be true or false)
#check True       -- True : Prop
#check False      -- False : Prop
#check 1 + 1 = 2  -- 1 + 1 = 2 : Prop
```

Notice: `true` (lowercase) is a Bool value. `True` (uppercase) is a Prop — a proposition that is trivially provable.

Prop is the type of propositions. Your RL agent works with `Prop`-valued expressions — things like `∀ (n : Nat), n + 0 = n`.

Functions and arrows

```
-- A function from Nat to Nat
#check fun (n : Nat) => n + 1    -- Nat → Nat

-- In propositions, → means "implies"
-- "p → q" means "if p then q"
-- It's the SAME arrow — implication IS a function type
```

This is the Curry-Howard correspondence: **implication = function type**. A proof of `p → q` is a function that takes a proof of `p` and returns a proof of `q`.

Lesson 3: Logical Connectives

These are the building blocks your agent sees in goal targets:

Implication: →

```
```

```
-- "if p then q"
-- To prove p → q: assume p, then prove q
-- In tactics: use `intro h` to assume p (naming it h)
example : ∀ (p q : Prop), p → (p → q) → q := by
  intro p q hp hpq   -- assume p, q, hp : p, hpq : p → q
  apply hpq          -- goal becomes: prove p
  exact hp           -- we have hp : p, done
```

And: ∧

lean

```
-- "p and q"
-- To prove p ∧ q: prove p, then prove q (use `constructor`)
-- To use h : p ∧ q: extract h.1 (proof of p) and h.2 (proof of q)
example : ∀ (p q : Prop), p ∧ q → q ∧ p := by
  intro p q h     -- h : p ∧ q
  constructor     -- splits into two goals: prove q, prove p
  · exact h.2     -- h.2 is the proof of q
  · exact h.1     -- h.1 is the proof of p
```

Or: ∨

lean

```
-- "p or q"
-- To prove p ∨ q: prove p (use `left`) OR prove q (use `right`)
-- To use h : p ∨ q: case split (use `cases h`)
example : ∀ (p q : Prop), p ∨ q → q ∨ p := by
  intro p q h    -- h : p ∨ q
  cases h        -- two cases
  · right        -- in case h : p, we prove q ∨ p via right
    assumption   -- we have p in context
  · left         -- in case h : q, we prove q ∨ p via left
    assumption
```

Not: ¬

lean

```
-- ¬p is defined as p → False
-- To prove ¬p: assume p, derive a contradiction
-- To use h : ¬p with hp : p: apply h to hp, get False
example : ∀ (p : Prop), p → ¬¬p := by
  intro p hp hnp   -- hp : p, hnp : ¬p (which is p → False)
  exact hnp hp     -- apply hnp to hp, get False (contradiction)
```

For all: ∀

```lean
-- ∀ (x : T), P x means "for every x of type T, P x holds"
-- To prove it: use `intro x` to pick an arbitrary x
-- To use h : ∀ x, P x: apply h to a specific value
example : ∀ (n : Nat), n = n := by
  intro n     -- let n be an arbitrary Nat
  rfl         -- reflexivity: n = n is trivially true
```

Exists: ∃

```lean
-- ∃ (x : T), P x means "there exists an x of type T such that P x"
-- To prove it: provide a witness (use `exact ⟨witness, proof⟩`)
-- To use h : ∃ x, P x: destructure it (use `obtain ⟨x, hx⟩ := h`)
example : ∃ (n : Nat), n + n = 4 := by
  exact ⟨2, rfl⟩    -- witness is 2, and 2 + 2 = 4 by rfl
```

Equality: =

```lean
-- a = b
-- To prove it: rfl (if both sides compute to the same thing),
--              or rewrite using known equalities
-- To use h : a = b: rewrite goal using `rw [h]`
example : ∀ (n : Nat), n + 0 = n := by
  intro n
  simp    -- simplification knows n + 0 = n
```

Lesson 4: The Proof State (What Your Agent Sees)

When you write `by` and start a tactic proof, Lean shows you a **proof state**. This is exactly what Pantograph returns to your agent.

```lean
```

```
example : ∀ (p q : Prop), p ∧ q → q ∧ p := by
  -- PROOF STATE at this point:
  -- ⊢ ∀ (p q : Prop), p ∧ q → q ∧ p
  --
  -- The ⊢ symbol means "we need to prove"
  -- There are no hypotheses yet (nothing above the ⊢)
  intro p q h
  -- PROOF STATE:
  -- p q : Prop
  -- h : p ∧ q
  -- ⊢ q ∧ p
  --
  -- Now we have hypotheses (above ⊢) and a simpler goal (below ⊢)
  constructor
  -- PROOF STATE (two goals now):
  -- Goal 1:
  -- p q : Prop
  -- h : p ∧ q
  -- ⊢ q
  --
  -- Goal 2:
  -- p q : Prop
  -- h : p ∧ q
  -- ⊢ p
  exact h.2
  -- Goal 1 solved. Remaining:
  -- p q : Prop
  -- h : p ∧ q
  -- ⊢ p
  exact h.1
  -- All goals solved. Proof complete.
```

This is exactly the loop your RL agent runs:

Proof state component	RL equivalent
Hypotheses (above ⊢)	Part of the observation
Goal target (below ⊢)	Part of the observation
Number of remaining goals	Part of the observation
No goals remaining	Terminal state, reward = +1
Tactic	Action
Tactic fails	Invalid action, reward = -0.1

In Pantograph code:

```python
state = server.goal_start("∀ (p q : Prop), p ∧ q → q ∧ p")
# state.goals[0].target    = "∀ (p q : Prop), p ∧ q → q ∧ p"
# state.goals[0].variables = []

state = server.goal_tactic(state, 0, "intro p q h")
# state.goals[0].target    = "q ∧ p"
# state.goals[0].variables = [p : Prop, q : Prop, h : p ∧ q]

state = server.goal_tactic(state, 0, "constructor")
# state.goals[0].target    = "q"    (first subgoal)
# state.goals[1].target    = "p"    (second subgoal)
```

Lesson 5: Tactics Reference

These are the actions in your RL agent's action space. Each one transforms the proof state.

Introduction tactics (peel off the goal's outermost structure)

Tactic	When to use	What it does
`intro h`	Goal is `∀ x, ...` or `P → Q`	Names the variable/hypothesis `h`, goal becomes the body
`intros`	Same, but introduces everything at once	Introduces all leading ∀ and →

Example:

```
Before: ⊢ ∀ (p : Prop), p → p
After intro p h:  p : Prop,  h : p  ⊢ p
```

Closing tactics (finish a goal in one step)

Tactic	When to use	What it does
`exact h`	`h` has exactly the type of the goal	Closes goal with `h`
`assumption`	Some hypothesis matches the goal	Finds it automatically
`rfl`	Goal is `a = a`	Reflexivity

Tactic	When to use	What it does
`trivial`	Goal is `True` or very simple	Solves trivially
`decide`	Goal is decidable (finite check)	Brute force check

Example:

```
Before:  p : Prop,  h : p  ⊢ p
After exact h:  (goal closed)
```

Splitting tactics (break a goal into subgoals)

Tactic	When to use	What it does
`constructor`	Goal is `P ∧ Q` or `⟨...⟩`	Splits into: prove P, prove Q
`left`	Goal is `P ∨ Q`	Changes goal to: prove P
`right`	Goal is `P ∨ Q`	Changes goal to: prove Q

Example:

```
Before:  ⊢ p ∧ q
After constructor:  ⊢ p   AND   ⊢ q   (two subgoals)
```

Destructuring tactics (use a hypothesis)

Tactic	When to use	What it does
`cases h`	`h : P ∨ Q` or `h : ∃ x, ...`	Creates one branch per constructor
`obtain ⟨a, b⟩ := h`	`h : P ∧ Q` or `h : ∃ x, ...`	Extracts components
`induction n`	`n : Nat` (or any inductive type)	Creates base case + inductive step

Example:

```
Before:  h : p ∨ q  ⊢ q ∨ p
After cases h:
  Case 1:  h† : p  ⊢ q ∨ p
  Case 2:  h† : q  ⊢ q ∨ p
```

Rewriting tactics (transform the goal using equalities)

Tactic	When to use	What it does
`rw [h]`	`h : a = b`, goal contains `a`	Replaces `a` with `b` in goal
`rw [← h]`	`h : a = b`, goal contains `b`	Replaces `b` with `a` in goal
`simp`	Goal involves known simplification rules	Applies many rewrite rules

Example:

```
Before:  h : x = 5  ⊢ x + 1 = 6
After rw [h]:  ⊢ 5 + 1 = 6
```

Automation tactics (let Lean figure it out)

Tactic	What it solves
`simp`	Applies simplification lemmas
`omega`	Linear arithmetic over Nat and Int
`norm_num`	Numeric computations (1 + 1 = 2, etc.)
`ring`	Ring equalities (polynomial identities)
`tauto`	Propositional tautologies
`aesop`	General-purpose automation
`contradiction`	Finds contradictory hypotheses

Example:

```
Before:  ⊢ 2 * 3 + 1 = 7
After norm_num:  (goal closed)

Before:  ⊢ ∀ (n m : Nat), n + m = m + n
After intro n m; omega:  (goal closed)
```

Using hypotheses

Tactic	What it does
`apply h`	If `h : A → B` and goal is `B`, changes goal to `A`

Tactic	What it does
`exact h`	If `h` has exactly the goal type, closes goal
`exact h.1`	First component of `h : P ∧ Q` (the proof of P)
`exact h.2`	Second component of `h : P ∧ Q` (the proof of Q)
`exfalso`	Changes goal to `False` (proof by contradiction)

Example:

```
Before:  h : p → q  ⊢ q
After apply h:  ⊢ p
```

The · (focus dot)

When you have multiple goals, `·` focuses on one:

```lean
constructor
· exact h.2    -- solves first goal
· exact h.1    -- solves second goal
```

Your RL agent doesn't need this — Pantograph handles goals individually via `goal_id`.

Lesson 6: Complete Proof Walkthroughs

Proof 1: Identity (1 tactic)

```lean
-- "1 equals 1"
example : (1 : Nat) = 1 := by rfl
```

`rfl` works because both sides are literally the same expression.

Proof 2: Implication is reflexive (2 tactics)

```lean
-- "p implies p"
example : ∀ (p : Prop), p → p := by
  intro p h    -- assume p is a Prop, h is a proof of p
  exact h      -- h is exactly what we need
```

Step by step:

```
State 0: ⊢ ∀ (p : Prop), p → p
  ↓ intro p h
State 1: p : Prop, h : p ⊢ p
  ↓ exact h
(solved)
```

Proof 3: And is commutative (5 tactics)

```lean
-- "p and q implies q and p"
example : ∀ (p q : Prop), p ∧ q → q ∧ p := by
  intro p q h    -- h : p ∧ q
  constructor    -- split goal into ⊢ q and ⊢ p
  · exact h.2   -- h.2 is proof of q
  · exact h.1   -- h.1 is proof of p
```

Step by step:

```
State 0: ⊢ ∀ (p q : Prop), p ∧ q → q ∧ p
  ↓ intro p q h
State 1: p q : Prop, h : p ∧ q ⊢ q ∧ p
  ↓ constructor
State 2: [Goal 0] h : p ∧ q ⊢ q
         [Goal 1] h : p ∧ q ⊢ p
  ↓ exact h.2  (on goal 0)
State 3: [Goal 0] h : p ∧ q ⊢ p
  ↓ exact h.1  (on goal 0)
(solved)
```

Proof 4: Or is commutative (7 tactics)

```lean
example : ∀ (p q : Prop), p ∨ q → q ∨ p := by
  intro p q h
  cases h with       -- split on whether h is left or right
  | inl hp =>        -- case: hp : p
    right            -- prove q ∨ p via right side
    exact hp
  | inr hq =>        -- case: hq : q
    left             -- prove q ∨ p via left side
    exact hq
```

Simpler version (what your agent would do):

```lean
example : ∀ (p q : Prop), p ∨ q → q ∨ p := by
  intro p q h
  cases h
  · right; assumption
  · left; assumption
```

Proof 5: Modus ponens (3 tactics)

```lean
-- "if p, and p implies q, then q"
example : ∀ (p q : Prop), p → (p → q) → q := by
  intro p q hp hpq    -- hp : p,  hpq : p → q
  apply hpq           -- hpq : p → q, goal was q, now goal is p
  exact hp
```

Proof 6: Contrapositive (5 tactics)

```lean
-- "(p → q) → (¬q → ¬p)"
example : ∀ (p q : Prop), (p → q) → (¬q → ¬p) := by
  intro p q hpq hnq hp   -- hpq : p → q, hnq : ¬q, hp : p
  apply hnq              -- goal becomes q (since ¬q = q → False)
  apply hpq              -- goal becomes p
  exact hp
```

Proof 7: Natural number arithmetic (2 tactics)

```lean
example : ∀ (n : Nat), n + 0 = n := by
  intro n
  simp     -- simp knows that n + 0 = n

-- or equivalently:
example : ∀ (n : Nat), n + 0 = n := by
  intro n
  omega    -- omega handles linear arithmetic
```

Proof 8: Induction (multiple tactics)

```lean
```

```
-- Proving n + 0 = n by induction (the hard way, to show the structure)
example : ∀ (n : Nat), n + 0 = n := by
  intro n
  induction n with
  | zero =>
    -- Base case: 0 + 0 = 0
    rfl
  | succ k ih =>
    -- Inductive step: ih : k + 0 = k, ⊢ succ k + 0 = succ k
    simp [ih]
```

The proof state after `induction n`:

```
Goal 1 (base):    ⊢ 0 + 0 = 0
Goal 2 (step):    k : Nat,  ih : k + 0 = k  ⊢ k + 1 + 0 = k + 1
```

Lesson 7: Reading Pantograph Output

When your agent interacts with Pantograph, it sees `GoalState` objects. Here's how to read them:

```python
```

```
state = server.goal_start("∀ (p q : Prop), p ∧ q → q ∧ p")
print(state)
# ⊢ ∀ (p q : Prop), p ∧ q → q ∧ p

state = server.goal_tactic(state, 0, "intro p q h")
print(state)
# p q : Prop
# h : p ∧ q
# ⊢ q ∧ p

# Programmatic access:
state.goals[0].target      # "q ∧ p"
state.goals[0].variables   # [Variable(name="p", type="Prop"),
                           #  Variable(name="q", type="Prop"),
                           #  Variable(name="h", type="p ∧ q")]

state = server.goal_tactic(state, 0, "constructor")
len(state.goals)           # 2
state.goals[0].target      # "q"
state.goals[1].target      # "p"

state = server.goal_tactic(state, 0, "exact h.2")
len(state.goals)           # 1
state.goals[0].target      # "p"

state = server.goal_tactic(state, 0, "exact h.1")
state.is_solved            # True
```

Lesson 8: Common Patterns Your Agent Should Learn

Goal shape	Best tactic	Why
`⊢ ∀ (x : T), ...`	`intro x`	Peel off the quantifier
`⊢ P → Q`	`intro h`	Assume P, prove Q
`⊢ P ∧ Q`	`constructor`	Split into two subgoals
`⊢ P ∨ Q`	`left` or `right`	Choose which side to prove
`⊢ a = a`	`rfl`	Reflexivity
`⊢ numeric equation`	`norm_num` or `omega`	Automation
`⊢ same as hypothesis h`	`exact h` or `assumption`	Direct match

Goal shape	Best tactic	Why
`h : P ∨ Q` in context	`cases h`	Consider both cases
`h : P ∧ Q` in context	use `h.1` and `h.2`	Extract components
`h : P → Q`, goal is `Q`	`apply h`	Reduces goal to proving P
`n : Nat` and need induction	`induction n`	Base case + step

These patterns are exactly what your policy network needs to learn. The structural features in the encoder (`∧` flag, `∨` flag, `=` flag, etc.) give the network direct signal about which pattern applies.

Lesson 9: Why Tactics Fail

Understanding failures helps you understand the -0.1 reward your agent gets:

```
-- "rfl" fails when sides aren't definitionally equal
⊢ n + 0 = n       -- FAILS: Lean doesn't reduce n + 0 automatically
⊢ 0 + 0 = 0       -- WORKS: both sides reduce to 0

-- "exact h" fails when h's type doesn't match the goal
h : p, ⊢ q        -- FAILS: p ≠ q
h : p, ⊢ p        -- WORKS: exact match

-- "constructor" fails when goal isn't a structure/inductive
⊢ p               -- FAILS: p is just a Prop variable
⊢ p ∧ q           -- WORKS: And has a constructor

-- "cases h" fails when h isn't an inductive type
h : p → q         -- FAILS: function type, not inductive
h : p ∨ q         -- WORKS: Or is inductive

-- "omega" fails on non-linear arithmetic
⊢ n * n ≥ 0       -- FAILS: omega is linear only
⊢ n + 1 > n       -- WORKS: linear
```

Your agent will hit these failures constantly. That's normal and expected.
The -0.1 penalty teaches it to avoid tactics that don't match the goal shape.

Lesson 10: How This Maps to Your RL Agent

```
LEAN CONCEPT              RL CONCEPT
```

```
Theorem statement   →  Task / problem to solve
Proof state         →  Observation / state
Tactic              →  Action
Proof complete      →  Terminal state, reward +1
Tactic fails        →  Invalid action, reward -0.1
Lean kernel         →  Environment (ground truth)
Pantograph          →  Gym interface
```

The beauty of this setup: **Lean's kernel is a perfect verifier.** There are no false positives. If `state.is_solved` is True, the proof is correct. Period. This makes theorem proving an ideal RL environment — the reward signal is noise-free.

Quick Reference Card

```
SYMBOLS:
  ∀   for all        ∃   there exists
  →   implies        ↔   if and only if
  ∧   and            ∨   or
  ¬   not            ⊢   "we need to prove"
  :=  definition     by  start tactic proof

TYPES:
  Nat   natural numbers (0, 1, 2, ...)
  Int   integers
  Bool  true / false
  Prop  propositions (True, False, 1+1=2, ...)
  List  lists

TACTIC CHEAT SHEET:
  intro h       assume hypothesis
  exact h       close goal with h
  assumption    find matching hypothesis
  constructor   split ∧ goal
  left/right    choose ∨ side
  cases h       case split on h
  induction n   induction on n
  rw [h]        rewrite using h
  simp          simplify
  omega         linear arithmetic
  norm_num      numeric computation
  ring          polynomial identity
  rfl           reflexivity (a = a)
  apply h       backwards reasoning
```

contradiction find contradiction
exfalso switch to proving False
tauto propositional logic
decide decidable check