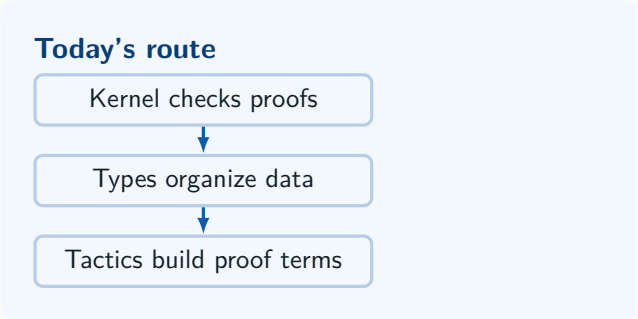
Lean Introduction

From formal proofs to basic tactics



July 15, Wednesday

Type judgments, propositions-as-types, basic tactics, and AI-assisted Lean work.

Learning Goals

Read Lean expressions
Use `a : A` and `#check`; separate type checking from evaluation.

Understand proofs as terms
Curry–Howard: propositions are types; proofs are terms.

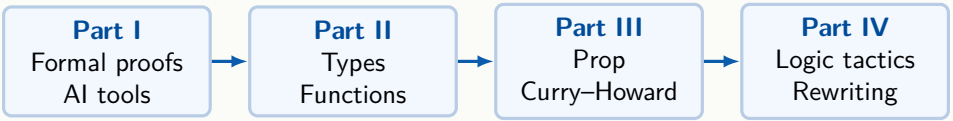
Write small programs
Define functions with `def`; inspect them with `#eval`, `#reduce`, and `#print`.

Build proof terms with tactics
Use `intro`, `exact`, `apply`, `constructor`, `cases`.

Use AI carefully
Let AI propose candidates; verify names, statements, and scripts in Lean.

Reason by equality
Use `rfl`, `rw`, and `simp` for rewriting.

Four-Part Outline



First: why a kernel can check proofs and why AI output still needs checking.

Then: the type-theoretic language in which Lean states and checks goals.

Six-Day Course Overview

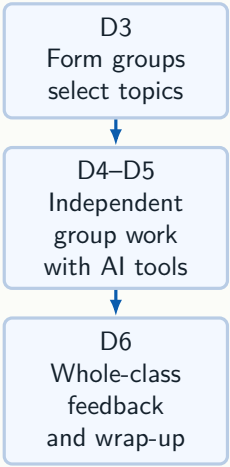
Day	Date	Topic
D1	Jul 15 (Wed)	Types, Prop, basic tactics, AI tools
D2	Jul 16 (Thu)	Inductive types, structures, type classes
D3	Jul 17 (Fri)	Instances, AI workflows, algebra, `SimpleGraph`
D4	Jul 18 (Sat)	Analysis, topology, filters, limits
D5	Jul 19 (Sun)	Functional programming, Monad
D6	Jul 20 (Mon)	Algorithm verification, termination, complexity

Tutorial and Project Schedule

Afternoon tutorials

D1	Environment, AI configuration, Mathlib search; `ring`/`linarith`/`decide`
D2	Inductive, structure, and class exercises
D3	Algebra/combinatorics exercises; project selection and grouping
D4–5	Project work; AI drafting, Mathlib search, Lean repair
D6	Feedback on specifications, invariants, termination, and complexity

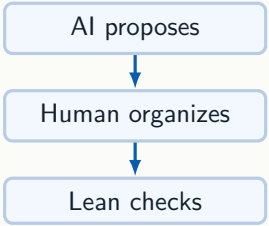
Project timeline



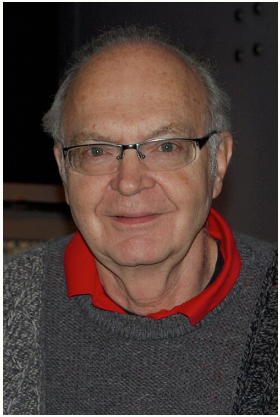
Part I: Formal Proofs and AI

Three questions

1. Why does Lean know whether a proof is correct?
2. Where does Lean fit in the history of mathematics?
3. How should we use AI tools alongside Lean?



Search, Proof, and Verification Working Together



Donald Knuth

THE PROBLEM

While preparing a future TAOCP section, Knuth studied the directed graph on m^3 vertices (i,j,k) modulo m. Each move increments one coordinate. The goal is to partition every arc into three Hamiltonian cycles.

CLAUDE OPUS 4.6

Ran 31 rounds of Python experiments and search. It discovered an explicit construction for every odd m in about one hour.

DONALD KNUTH

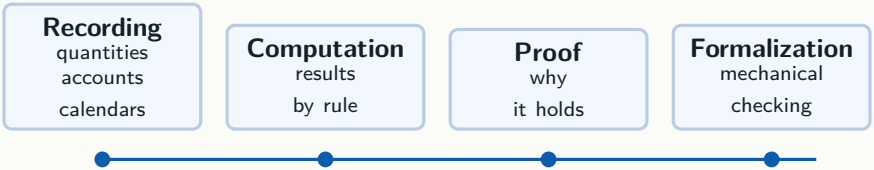
Formulated the problem, solved m = 3, then turned the candidate construction into a rigorous proof for all odd m.

STAPPERS + LEAN

Filip Stappers coached the search and tested odd m from 3 to 101. Kim Morrison formalized Knuth's proof; the Lean kernel checked the result.

Knuth, Claude's Cycles, 28 Feb 2026; revised 14 Apr 2026

From Recording to Checkable Proofs



Lean's position

A proof written in Lean is an object. The kernel checks that object step by step against explicit rules.

Ancient Mathematics: Recording and Computing

Record
quantities, land, debt, astronomy

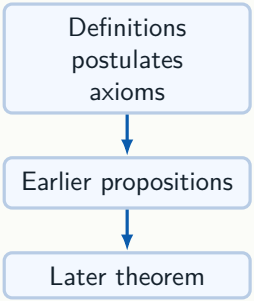
Compute
apply a rule to obtain a result

Explain
ask why the result must hold

Early mathematical traditions served memory and calculation first. Formalization inherits the same discipline:

- records must be accurate;
- computations must be reliable;
- reasons must be inspectable.

Euclid: Proofs as Dependency Networks



Elements (c. 300 BCE) established a model still in use:

- start from definitions and postulates;
- order propositions by dependency;
- use diagrams for intuition while proofs carry the conclusion.

Lean's dependency graph is a direct descendant.

Al-Khwarizmi: Algorithms and Knowledge Preservation

Algebra

Transform problems into symbols and rules.

Algorithm

Give steps that can be executed repeatedly.

Mathlib inherits the tradition of organizing steps and knowledge for reuse.

Modern Proofs Grow Long: Fermat to Wiles

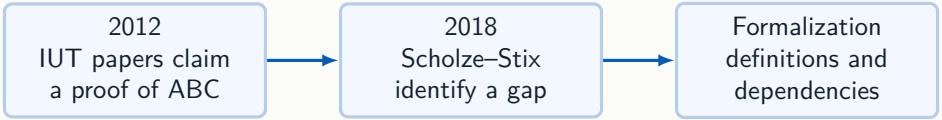
Fermat's Last Theorem:

$a^n + b^n = c^n, \quad n > 2$ has no positive integer solution.

- Short to state; centuries to prove.
- Wiles (1995): semi-stable case via modular forms and Galois representations.
- Complete modularity theorem completed by subsequent work.

As proofs grow longer, the question of *who checked each step* becomes pressing.

When Verification Becomes Part of the Story



The community needs not only answers, but a form of expression that can be jointly inspected.

Hilbert: Can Proof Checking Be Mechanized?

1900
Hilbert's program

David Hilbert (1862–1943) aimed to:

1. place mathematics on an explicit formal foundation;
2. express proofs in a symbolic language with fixed rules;
3. prove the consistency of those foundations.

Connection to Lean

A machine-checked proof makes every formal step answerable to explicit rules rather than authority or intuition.

Principia Mathematica

1910–13
Symbolic derivation

Russell and Whitehead aimed to derive mathematics from explicit symbolic rules.

- The derivation of $1 + 1 = 2$ became famously long.
- Formalization made dependencies explicit.
- It also exposed the high cost of writing every step by hand.

Next pressure

Formal detail is necessary but expensive. Useful systems must automate routine construction while preserving exact checking.

Gödel: Incompleteness

1931

Limits

Gödel's incompleteness theorems show that:

- a sufficiently strong consistent formal system cannot prove all true statements;
- such a system typically cannot prove its own consistency internally.

What remains possible

Within a fixed system, a kernel can still check each concrete proof. Lean checks given formal proofs; it does not prove all truths.

Two Routes: Proof Search vs. Proof Checking

Proof search

Machine tries to find a derivation.

- 1956: Logic Theorist
- Today: ATP solvers, AI provers

Proof checking

The proof object is supplied; the kernel verifies it.

- 1968: Automath
- Today: Coq, Lean kernel

In Lean, the two meet: tactics and AI help with search, but every proof must pass the kernel.

Automath: Proofs as Formal Objects

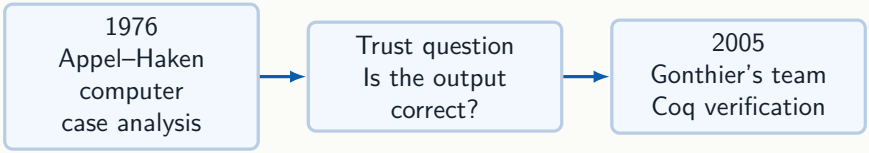
1968
Automath

Nicolaas de Bruijn's Automath made three design choices:

1. represent proofs as precisely typed objects;
2. check every line against a rule;
3. keep the trusted checking kernel small.

The shift
A proof becomes an object that a small program can check. Lean retains this architecture.

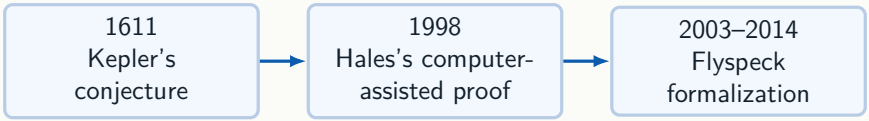
Four-Color Theorem: Computer-Assisted Proof



Two obligations

Check both the computation that handles the cases and the formal argument that connects those cases to the theorem.

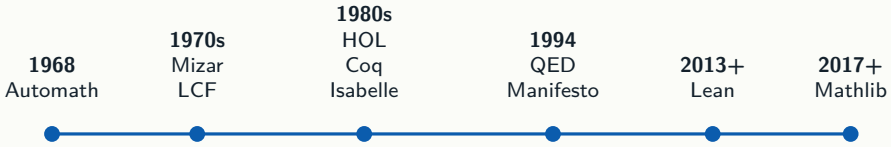
Flyspeck: Engineering a Complex Proof



Proof engineering

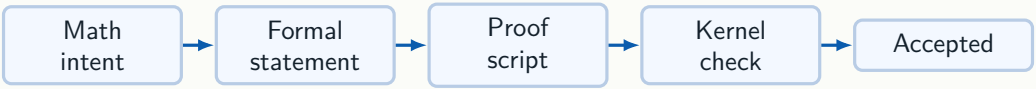
A large formal proof depends on reusable libraries, checked computations, and sustained maintenance in HOL Light and Isabelle.

Proof Assistants as Infrastructure



Proof assistants moved from individual checkers to shared mathematical infrastructure.

The Trust Chain of Formal Proofs



Lean checks whether the formalized statement has a valid proof. Mathematicians separately verify that the statement captures the intended mathematics.

Lean: Language, Checker, and Math Library

Language
write programs and definitions

Checker
kernel checks proof terms

Library
Mathlib stores reusable mathematics

The working mindset: enter a reusable, checkable mathematical knowledge base rather than proving the world from scratch.

Three Traditions Behind Lean

Alan Turing

Computation
programs evaluate

Per Martin-Löf

Type theory
proofs are terms

Philip Wadler

Functional style
functions compose

Lean is useful because these traditions meet in one language: compute, state, prove, and reuse.

Mathlib: A Reusable Math Library

Mathlib contains:

- **Definitions** — types, structures, functions;
- **Theorems** — results with machine-checked proofs;
- **Notation** — mathematical symbols;
- **Type classes** — algebraic, order, topological structures.

Library-centered workflow: find existing definitions and theorems first; then write your own proof on top.

Theorem Names Are Searchable Addresses

Lean theorem names follow a pattern:

- **prefix** — `Nat`, `Int`, `Set` (namespace / domain);
- **core word** — `add`, `mem`, `ext` (operation or relation);
- **modifier** — `comm`, `left` (direction or property).

```
#check Nat.add_comm    -- Nat.add_comm : a + b = b + a
#check Int.add_comm
#check Set.mem_union
#check Finset.mem_union
```

Live Lean: `codeD1/D1.lean`

Different Namespaces, Same Short Name

Many mathematical objects have an "extensionality" theorem named `ext`; namespaces keep them distinct.

```
#check Set.ext
-- (h : forall x, x in s <-> x in t) : s = t

#check Finset.ext
-- (h : forall x, x in A <-> x in B) : A = B
```

This is why namespaces matter: short names can repeat; full names remain unique.

Live Lean: `codeD1/D1.lean`

File Organization Commands

```
import Mathlib              -- load a file or library
namespace Demo
  def double (n : Nat) : Nat := n + n
end Demo
section
  variable (n : Nat)
  example : Demo.double n = n + n := by rfl
end
open Nat
#check zero_add             -- same as Nat.zero_add
```

`import`: load a library; `namespace`: add a name prefix; `section`: scope local variables; `open`: drop a prefix temporarily.

Live Lean: `codeD1/D1.lean`

Search Mathlib from Lean: Name, Type, or Goal

Start from the information you already have: a declaration name, a type shape, or the current goal in the Infoview.

```
#check Nat.add_comm
#print Nat.add_comm
-- #find <narrow type pattern>

example (a b : Nat) : a + b = b + a := by
  exact?
```

KNOWN NAME

#check shows the type; #print shows the stored declaration.

TYPE SHAPE

#find searches declarations whose type matches a pattern.

CURRENT GOAL

exact?, apply?, rw?, and simp? ask Lean to suggest a checked next step.

WORKFLOW Run the suggestion, inspect the generated theorem name, then keep the explicit proof in your file.

Live Lean: D1/D1.lean

Search Mathlib on the Web: Choose the Right Query

Use the web when you know the mathematics but not the declaration name. Different tools accept different kinds of clues.

Mathlib documentation

https://leanprover-community.github.io/mathlib4_docs/

Best for exact names, module APIs, signatures, source links, and nearby declarations.

```
Example: Nat.add_comm
```

LeanSearch

https://leansearch.net/

Best for a natural-language or mathematical description when you do not know Lean's name.

```
Example: dim(V)=dim(ker)+dim(im)
```

Loogle

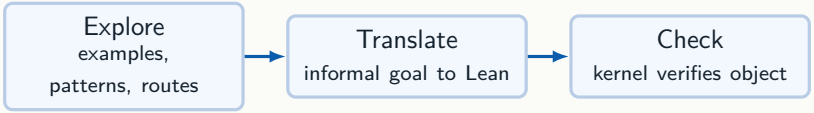
https://loogle.lean-lang.org/

Best for constants, name fragments, subexpressions, and theorem-type patterns.

```
Example: _ * (_ ^ _)   or   |- _ < _ -> _
```

VERIFY Copy the candidate name back into Lean and confirm it with #check in your installed Mathlib version.

AI for Math: Where Does Lean Fit?



Where this course fits

We learn the target language: the precise Lean statements and proof scripts that AI tools must eventually produce and that formalization projects share.

Choosing AI for Lean Work

Proof routes
Ask for lemmas and a proof outline; do not trust a complete script blindly.

Code repair
Provide the error, goal state, imports, and a narrow edit target.

Retrieval
Use search and citations; verify names, versions, and statements in Lean.

Local deploy
Use open-weight models for privacy or batch experiments.

Operational rule: AI proposes candidates; students decompose; Lean gives the final verdict.

What AI Is Good at in Math Learning

Explore
examples, conjectures, proof routes

Explain
definitions, analogies, small examples

Collaborate
drafts, code completion, gap finding

Use it as a co-pilot

AI broadens the search space and speeds up drafts. It does not replace the obligation to understand the statement and check the proof.

Hallucinations: The Dangerous Failure Mode

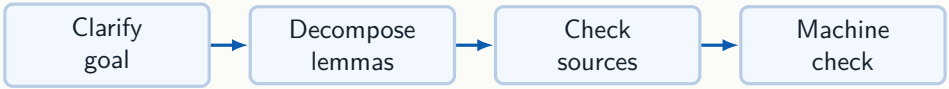
Looks plausible
Fluent proof text can contain fabricated theorem names, missing lemmas, or changed hypotheses.

Root cause
Language models optimize for plausible continuation, not for a global dependency graph.

Failure signs
Names fail under `#check`; goals silently change; key cases are skipped.

Response
Treat AI output as a draft candidate, not as mathematical evidence.

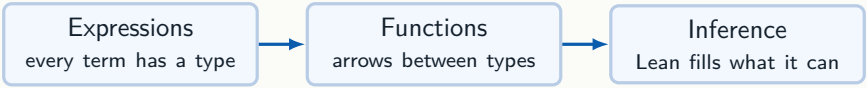
Making AI Output Checkable



Precise statements expose variables, assumptions, and quantifiers.

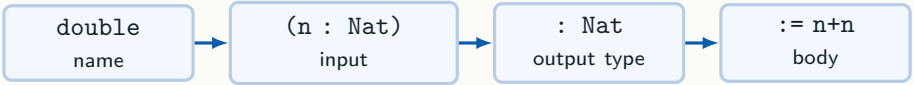
`#check`, `exact?`, Loogle, and Lean errors turn guesses into verifiable work.

Part II: Types and Functions



Read Lean from right to left when needed: first identify the type, then understand the expression inhabiting it.

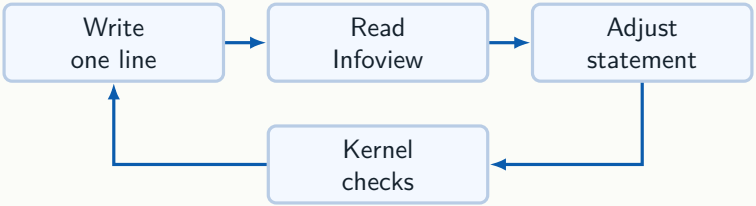
How to Read a Lean Line



Reading rule

First locate the colon. The expression on the left is being assigned a type on the right; after `:=`, Lean stores the actual definition.

The First Lean Feedback Loop



Beginners progress fastest when they make small edits and read the goal after every line.

Start Writing Lean

```
import Mathlib

#check Nat         -- Nat : Type
#eval 2 + 3        -- 5

example (P : Prop) : P -> P := by
  intro h
  exact h
```

Core habit: after every line, read the Infoview panel to see the current type or goal.

Live Lean: `codeD1/D1.lean`

Three Core Ideas of Type Theory

Judgment

`2 : Nat`

expression belongs to a type

Arrow

`f : A -> B`

input type to output type

Proof

`h : P`

evidence inhabits a proposition

Main shift

Lean proofs are not natural-language paragraphs. They are objects built in type theory and checked by the kernel.

Type Judgments: `a : A`

The basic sentence in Lean is a *type judgment*: `a : A`, read "`a` has type `A`."

```
#check 42           -- 42 : Nat
#check Nat          -- Nat : Type
#check Type         -- Type : Type 1
#check 42 + 1       -- 42 + 1 : Nat
-- #check 42 + "hi"  -- error: type mismatch
```

A type error does not mean `False`; it means the expression was rejected before truth or proof was even considered.

Live Lean: `codeD1/D1.lean`

Every Expression Has a Type

```
#check Nat          -- Nat : Type
#check 37           -- 37 : Nat
#check "hello"      -- "hello" : String
#check true         -- true : Bool
```

- `#check` asks for the type.
- `#eval` asks for the computed value.
- These are two different questions.

Live Lean: `codeD1/D1.lean`

Type Conversion: Same Value, Different Type

Functions and operators accept specific input types. Conversion is needed when a value must cross that type boundary without hiding what may change.

1. LOSSLESS COERCION: Nat -> Int

When the target type is known, Lean can insert a registered safe coercion.

```
def count : Nat := 5
#check (count : Int)
#eval (count : Int) - 7  -- -2
```

2. EXPLICIT CONVERSION: Int -> Nat

A negative Int has no exact Nat image, so the conversion must be explicit.

```
#eval Int.toNat (-3)  -- 0
#eval Int.toNat 5     -- 5
```

READING RULE Ask: What is the source type? What target type is expected? Is the conversion lossless, or can it discard information?

Live Lean: D1/D1.lean

#check, #reduce, #print

```
def onePlusOne : Nat := 1 + 1

#check  onePlusOne   -- onePlusOne : Nat
#reduce onePlusOne   -- 2
#print  onePlusOne   -- def onePlusOne : Nat := 1 + 1
```

`#check` What type does this name or expression have?

`#reduce` Reduce the expression to its normal form.

`#print` What is stored in the environment for this name?

Live Lean: `codeD1/D1.lean`

Types Also Have Types: Universes

```
#check Nat      -- Nat : Type
#check Prop     -- Prop : Type
#check Type     -- Type : Type 1
#check Type 1   -- Type 1 : Type 2
-- Sort 0 = Prop,  Sort 1 = Type,  Sort u : Type (u+1)
```

Why layering? If there were one "type of all types" `U : U`, self-reference would reintroduce paradoxes. Lean uses `Type u : Type (u+1)` to avoid them.

Live Lean: `codeD1/D1.lean`

Why Not `U : U`? Russell's Problem

The naive idea: a "set of all sets that do not contain themselves."

$$R = \{ x \mid x \notin x \} \qquad \text{Is } R \in R?$$

Both answers lead to contradiction. Lean avoids this with universe levels:

```
#check Type      -- Type : Type 1
#check Type 1    -- Type 1 : Type 2
-- Type u : Type (u + 1)
```

Live Lean: `codeD1/D1.lean`

Functions: Basic Building Blocks

```
def double (n : Nat) : Nat :=
  n + n

#check double    -- double : Nat -> Nat
#eval  double 21 -- 42
```

Parts of a definition:

- `def` — give the object a name;
- `(n : Nat)` — an explicit input parameter;
- `: Nat` — the return type;
- `n + n` — the function body.

Live Lean: `codeD1/D1.lean`

Anonymous Functions and Higher-Order Functions

```
#check fun x : Nat => x + 1

def applyTwice (f : Nat -> Nat) (x : Nat) : Nat :=
  f (f x)

#eval applyTwice (fun n => n + 1) 10   -- 12
```

A function can take another function as input. This is the core of functional programming.

Live Lean: `codeD1/D1.lean`

Multi-Argument Functions: Currying

```
def add3 (a b c : Nat) : Nat := a + b + c

#check add3         -- add3 : Nat -> Nat -> Nat -> Nat
#check add3 1       -- add3 1 : Nat -> Nat -> Nat
#check add3 1 2     -- add3 1 2 : Nat -> Nat
#check add3 1 2 3   -- add3 1 2 3 : Nat
```

`add3 1` is still a function waiting for two more arguments. Partial application comes for free in Lean.

Live Lean: `codeD1/D1.lean`

Implicit Arguments: Lean Infers Them

```
def myConst {A B : Type} (a : A) (_ : B) : A := a

#eval myConst 5 "ignored"   -- 5
```

Curly-bracket parameters `{A B : Type}` are inferred by Lean from the surrounding context.
When starting out, read implicit arguments as “type information that Lean fills in automatically.”

Live Lean: `codeD1/D1.lean`

Three Bracket Types

The bracket style on a parameter tells Lean *who provides it*.

`(x : α)` **Explicit.** You supply it. Used for the main mathematical objects.

`{α : Type}` **Implicit.** Lean infers it from later arguments or the goal.

`[Monoid α]` **Instance.** Lean searches the instance database (algebraic structure, order, topology, . . .).

```
#check List.length     -- hides inferred arguments
#check @List.length    -- shows all arguments
#check @Nat.add_comm   -- shows [AddCommMonoid] instance
```

Live Lean: `codeD1/D1.lean`

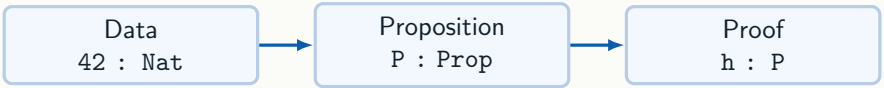
Calling Functions with Implicit Arguments

```
def same {alpha : Type} (x : alpha) : alpha := x

#check same 3           -- same 3 : Nat  (alpha inferred)
#check @same Nat 3      -- explicit: alpha = Nat

example {alpha : Type} [AddMonoid alpha] (x : alpha) :
    x + 0 = x := by simp

example : same (alpha := Nat) 3 = 3 := by rfl
```

- `@f` shows all hidden arguments.
- `f (alpha := Nat)` supplies a named implicit argument.
- `[AddMonoid alpha]` triggers instance search.

Live Lean: `codeD1/D1.lean`

Part III: Curry-Howard and Prop



What tactics do

A tactic script is a convenient way to construct a term whose type is the current proposition.

Dependent Functions: Π, $\forall$, and `fun`

```
#check ((n : Nat) -> Fin (n + 1))
-- the output type depends on the input n

#check (forall n : Nat, n = n)
-- universally quantified proposition

example : (forall n : Nat, n = n) := by
  intro n
  rfl
```

Read $\forall n, P\ n$ as "a function type: give any `n`, return a proof of `P n`." Then `intro` is simply the function-argument step.

Live Lean: `codeD1/D1.lean`

Propositions as Types

Data side

```
42 :  Nat
true :  Bool
```

a value inhabits a data type

Proof side

```
h :  2 + 2 = 4
h :  P
```

a proof inhabits a proposition

The same colon does both jobs: “has type” and “is a proof of”.

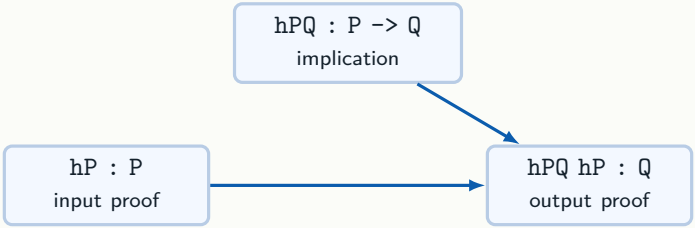
The Curry-Howard Correspondence

Logic	Type theory
proposition `P`	type `P`
proof of `P`	term `p : P`
proof of `P -> Q`	function `f : P -> Q`
use hypothesis `h : P`	use term `h`

Mental model

To prove an implication, build a function. To use an implication, apply that function to a proof of its premise.

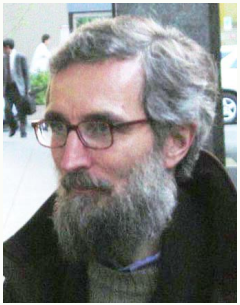
Implication Runs Like a Function



What `apply` does

If the goal is `Q` and Lean has `hPQ : P -> Q`, then `apply hPQ` changes the goal to `P`.

Per Martin-Löf (1942–)



Per Martin-Löf

Type theory as foundations
Martin-Löf Type Theory (1972–1984) is a direct ancestor of Lean's type system.

Propositions as types
This is not just an analogy: a proposition is treated as the type of its proofs.

Dependent types
Types may depend on values, e.g. vectors indexed by their length.

Curry-Howard Dictionary

Logic	Proof construction	What to do
`True`	`True.intro`	trivial; one proof
`False`	no constructor	nothing proves it
`P -> Q`	`fun hP => hQ`	give `P`, return `Q`
`P /\ Q`	`And.intro hP hQ`	supply both proofs
`P \/ Q`	`Or.inl hP` or `Or.inr hQ`	supply one side
`not P`	`fun h => absurd`	`P -> False`

Curry-Howard in Lean

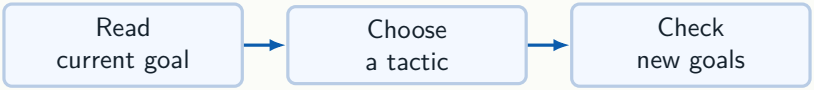
```
example (P Q : Prop) : P -> Q -> And P Q := by
  intro hP
  intro hQ
  exact And.intro hP hQ

#check And.intro
-- And.intro : P -> Q -> P /\ Q
```

- `intro hP` puts the proof of `P` into context.
- `And.intro hP hQ` assembles both proofs into a proof of `And P Q`.

Tactics are not magic commands; they construct a term whose type is the current goal.

Live Lean: `codeD1/D1.lean`

Part IV: Logic Tactics



What tactics are
Commands that construct a proof term step by step.

How to debug
Move cursor line by line and verify each goal transformation.

Goal Shape Suggests the Tactic

Goal shape	First tactic	Effect
`P -> Q` **or** `forall x, ...`	`intro`	move assumption into context
`P /\ Q`	`constructor`	split into two goals
`Exists x, P x`	`use`	provide a witness
equality goal	`rfl`, `rw`, `simp`	compare or rewrite both sides
arithmetic goal	`omega`, `decide`	let automation finish

Reading the Lean Infoview

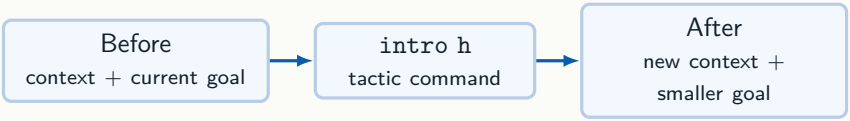
The Infoview has two regions: available facts above the line, target below.

```
P Q : Prop          -- context (hypotheses)
hP : P
hPQ : P -> Q
----- Goal -----
Q                   -- what you need to prove
```

Context
names and types you may use

Goal
the proposition still to prove

A Tactic Is a Goal Transformer



Good tactic choice

The goal becomes simpler and the context becomes more useful.

Bad tactic choice

Lean rejects it, or the new goal is not the one you expected.

`exact`: Closing a Goal

```
example (P : Prop) (h : P) : P := by
  exact h
```

Infoview reading:

- Context: `h : P`
- Goal: `P`
- `exact h` closes the goal because `h` has exactly type `P`.

`exact` means: I already have a term whose type matches the goal.

Live Lean: `codeD1/D1.lean`

`intro`: Proving Implications

```
example (P : Prop) : P -> P := by
  intro h
  exact h
```

Goal changes:

1. Before `intro h`: goal is `P -> P`
2. After `intro h`: context gains `h : P`; goal is `P`
3. `exact h` closes it.

`intro` moves an assumption from the goal into the context.

Live Lean: `codeD1/D1.lean`

`apply`: Using a Theorem Backwards

```
example (P Q : Prop) (hPQ : P -> Q) (hP : P) : Q := by
  apply hPQ
  exact hP
```

- Goal before `apply hPQ`: `Q`
- Goal after: `P`
- `exact hP` closes it.

`apply` reduces the current goal to the premises of a known implication.

Live Lean: `codeD1/D1.lean`

Build or Split the Logical Object

When the goal is compound

`constructor` builds an `And`; `use` builds an `Exists`; `Or.inl` and `Or.inr` choose a side.

When a hypothesis is compound

`cases h` decomposes an `Or`, an `And`, or an existential witness already in the context.

The same connective has two operations: build it when it is the goal, split it when it is a hypothesis.

`constructor`: Splitting a Conjunction

```
example (P Q : Prop) (hP : P) (hQ : Q) : And P Q := by
  constructor
  exact hP
  exact hQ
```

`constructor` on `And P Q` creates two subgoals:

1. After `constructor`: goal 1 is `P`, goal 2 is `Q`.
2. `exact hP` closes goal 1.
3. `exact hQ` closes goal 2.

Live Lean: `codeD1/D1.lean`

cases: Splitting a Disjunction

```
example (P Q : Prop) : Or P Q -> Or Q P := by
  intro h
  cases h with
  | inl hP => exact Or.inr hP
  | inr hQ => exact Or.inl hQ
```

`cases h` on `Or P Q` creates two branches:

- `| inl hP =>` branch: context has `hP : P`
- `| inr hQ =>` branch: context has `hQ : Q`

Each branch must close the goal independently.

Live Lean: `codeD1/D1.lean`

Negation, `False`, and `contradiction`

```
example (P : Prop) (hP : P) (hNotP : Not P) : False := by
  contradiction

example (P : Prop) (hP : P) (hNotP : P -> False) : False := by
  contradiction
```

- `Not P` is defined as `P -> False`.
- `False` has no constructor; nothing proves it.
- When context contains both `P` and `Not P`, `contradiction` finds the conflict automatically.

Live Lean: `codeD1/D1.lean`

`use`: Witness for Existence

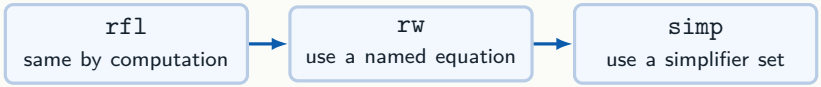
```
example (m : Nat) : Exists (fun n : Nat => 0 + n = m) := by
  use m
  simp
```

- `use m` provides the concrete witness `m`.
- The remaining goal is the property: `0 + m = m`.
- `simp` closes it.

`Exists` is written $\exists$ in Lean: $\exists$ `n, 0 + n = m`.
A proof of an existential $=$ a concrete witness $+$ a proof that it satisfies the property.

Live Lean: `codeD1/D1.lean`

Equality Proofs: Three Levels



Playbook

First ask whether both sides compute to the same expression. If not, rewrite with a specific equation. If the goal is routine cleanup, try `simp`.

Definitional Equality: What `rfl` Checks

```
def onePlusOne : Nat := 1 + 1

#reduce (fun n : Nat => n + 1) 3   -- 4
#reduce 2 + 3                      -- 5
#reduce onePlusOne                 -- 2
```

Two expressions are *definitionally equal* if the Lean kernel reduces both to the same normal form. `rfl` closes a goal exactly when both sides reduce the same way.

Live Lean: `codeD1/D1.lean`

`rfl`: Same After Unfolding

```
example (a : Nat) : a + 0 = a := by
  rfl                      -- works: definitionally equal

example (a : Nat) : 0 + a = a := by
  exact Nat.zero_add a     -- needs a theorem
```

- `a + 0 = a`: Lean reduces both sides to the same thing.
- `0 + a = a`: reduction does not equate them; the theorem `Nat.zero_add` is needed.

Live Lean: `codeD1/D1.lean`

When `rfl` Fails

```
example : 2 + 3 = 5 := by rfl    -- succeeds

-- example : 2 + 2 = 5 := by rfl  -- fails: not equal

example (a : Nat) : 0 + a = a := by
  exact Nat.zero_add a
```

When `rfl` fails:

- Use `#reduce` to see what each side computes to.
- Search for a theorem using the goal's shape.
- Use `rw` or `simp` to reduce it to something `rfl` can handle.

Read `rfl` as "please compare by definition," not "search for a proof."

Live Lean: `codeD1/D1.lean`

`rw`: Rewriting with an Equation

```
example (a b c : Nat) (h : a = b) : a + c = b + c := by
  rw [h]          -- replaces a with b in the goal

example (a b c : Nat) (h : a = b) : c + b = c + a := by
  rw [h.symm]     -- replaces b with a in the goal
```

- `rw [h]` rewrites left-to-right using `h`.
- `rw [h.symm]` or equivalently `rw [← h]` rewrites right-to-left (the arrow is the idiomatic Lean 4 form).
- `rw` gives precise control over exactly which equality is used.

Live Lean: `codeD1/D1.lean`

`simp`: Automatic Simplification

```
example (a : Nat) : a + 0 = a := by
  simp

def square (n : Nat) : Nat := n * n

example : square 3 = 9 := by
  simp [square]
```

- `simp` applies a library of standard simplification lemmas.
- `simp [f]` additionally unfolds the definition `f`.
- `simp` is automatic and fast; it may close goals that would need several `rw` steps.

Live Lean: `codeD1/D1.lean`

`rw` vs `simp`: When to Use Which

	`rw [h]`	`simp` / `simp [h]`
Choose when	One known equation should change the goal	Standard simplification lemmas should clean the goal
Control	One explicit rewrite	Automatic sequence of rewrites
Best use	Targeted transformation	Routine cleanup or closing a goal

Decision rule

Use `rw` when direction and location matter; use `simp` when the goal contains routine algebraic clutter.

Two More Useful Tactics: `omega` and `decide`

```
-- omega: linear arithmetic over Nat and Int
example (n : Nat) (h : n > 3) : n >= 4 := by omega

example (a b : Nat) : a + b = b + a := by omega

-- decide: any decidable proposition with finite cases
example : 2 + 3 = 5 := by decide

example : (3 : Nat) < 7 := by decide
```

- `omega` solves linear arithmetic goals automatically — addition, subtraction, inequalities over `Nat`/`Int`.
- `decide` evaluates a decidable proposition at compile time; works whenever the kernel can reduce the goal to `True`.
- Use these before reaching for `rw` or `simp` on numeric goals.

Live Demo Path

The session-opening code snippet follows this route:

```
import Mathlib
#check Nat         -- step 1: see a type
#eval 2 + 3        -- step 1: compute a value

def double (n : Nat) : Nat := n + n  -- step 2: write a function
#eval double 21

example (P : Prop) : P -> P := by   -- step 3: enter proof mode
  intro h
  exact h
```

First check types, then define functions, then prove.

Live Lean: `codeD1/D1.lean`

Nine Actions to Remember

Inspect

```
#check e
#eval e
```

Transform

```
intro h
apply h
rw [h]
```

Automate

```
simp
omega
decide
```

Close

```
exact h
rfl
```

Split

```
constructor
cases h
use t
```

Habit

read goal
choose action
confirm change

Suggested Exercises: Implication and Rewriting

Work through these in `lean-code/D1.lean`. After each tactic line, record how the Infoview goal changes.

```
-- 1. implication chaining  (intro, apply, exact)
example (P Q R : Prop) : (P -> Q) -> (Q -> R) -> P -> R := by
  sorry

-- 2. rewriting  (rw, omega)
example (a b : Nat) (h : a = b) : a + a = b + b := by
  sorry
```

1. Predict the first goal before typing a tactic.
2. After each line, say which hypothesis or expression changed.
3. Replace one tactic with a different tactic that still closes the goal.

Suggested Exercises: Cases and Witnesses

These exercises change the goal by exposing the structure of a proposition.

```
-- 3. And commutativity  (intro, cases, constructor, exact)
example (P Q : Prop) : P /\ Q -> Q /\ P := by
  sorry

-- 4. existential witness  (use, omega)
example : Exists (fun n : Nat => n > 10) := by
  sorry
```

For conjunction

First expose the hypothesis; then build the output conjunction in the opposite order.

For existence

Choose a concrete witness first; the remaining goal is an arithmetic fact.