

02-logic.lean

```
import Mathlib
```

You may skip the materials tagged with [IGNORE] for the first runthrough. Most of them are here to illustrate the nature of inductive types, which may be too advanced for beginners.

Materials tagged with [EXR] are recommended for you to try before looking at the solution.

Materials tagged with [TODO] means that I'm still working on it, or I'm not sure about the content yet. Feel free to give your suggestions!

At the Very Beginning...

There are some basic notions you should be familiar with.

1 : and :=

3 : \mathbb{N} means that 3 is a term of type \mathbb{N} .

By the Curry–Howard correspondence, $hp : p$ means that hp is a proof of the proposition p .

```
#check 3
#check ℕ

#check ∀ x : ℝ, 0 ≤ x ^ 2
#check sq_nonneg
#check (sq_nonneg : ∀ x : ℝ, 0 ≤ x ^ 2)
```

:= is used to define terms.

```
def myThree : ℕ := 3

#check myThree
```

theorem is just a definition in the Prop universe By the Curry–Howard correspondence, for theorem, behind :, the theorem statement follows; behind :=, a proof should be given.

```
theorem thm_sq_nonneg : ∀ x : ℝ, 0 ≤ x ^ 2 := sq_nonneg

-- `example` is just an anonymous theorem
example : ∀ x : ℝ, 0 ≤ x ^ 2 := thm_sq_nonneg
```

Logic (Part I)

We shall work out the basic logic in Lean's dependent type theory.

In this part, we cover:

- Implication

- Syntax for defining functions / theorems
- Tactic Mode

[IGNORE] You may notice along the way that except \rightarrow , all other logical connectives are defined as inductive types. And they have their own self-evident introduction rules and elimination rules. We shall discuss inductive types later in this course. These logical connectives serve as good examples.

2 Implication \rightarrow

Implication \rightarrow is the most fundamental way of constructing new types in Lean's dependent type theory. It's one of the first-class citizens in Lean.

In the universe of `Prop`, for propositions p and q , the implication $p \rightarrow q$ means “if p then q ”.

```
section

variable (p q r : Prop) -- this introduces global variables within this section

#check p
#check q
#check p → q
```

\rightarrow is right-associative. In general, hover the mouse over the operators to see how they associate. so $p \rightarrow q \rightarrow r$ means $p \rightarrow (q \rightarrow r)$. You may notice that this is logically equivalent to $p \wedge q \rightarrow r$. This relationship is known as currification. We shall discuss this later.

modus ponens

```
theorem mp : p → (p → q) → q := by sorry -- `sorry` is a placeholder for unfinished proofs
```

By the Curry–Howard correspondence, $p \rightarrow q$ is also understood as a function that takes a proof of p and produces a proof of q .

We introduce an important syntax to define functions / theorems: When we define a theorem `theorem name (h1 : p1) ... (hn : pn) : q := ...`, we are actually defining a function `name` of type $(h1 : p1) \rightarrow \dots \rightarrow (hn : pn) \rightarrow q$. Programmingly, $h1, \dots, hn$ are the parameters of the function and q is the return type.

The significance of this syntax, compared to `theorem name : p1 → ... → pn → q := ...`, is that now $h1, \dots, hn$, proofs of $p1, \dots, pn$, are now introduced as hypotheses into the context, available for you along the way to prove q .

this proves a theorem of type $p \rightarrow p$

```
example (hp : p) : p := hp
```

modus ponens, with a proof

```
example (hp : p) (hq : p → q) : q := hq hp
```

A function can also be defined inline, using `fun` (lambda syntax): `fun (h1 : p1) ... (hn : pn) → (hq : q)` defines a function of type $(h1 : p1) \rightarrow \dots \rightarrow (hn : pn) \rightarrow q$

Some of the type specifications may be omitted, as Lean can infer them.

```

example : p → p := fun (hp : p) ↤ (hp : p)
example : p → p := fun (hp : p) ↤ hp
example : p → (p → q) → q := fun (hp : p) (hq : p → q) ↤ hq hp
example : p → (p → q) → q := fun hp hq ↤ hq hp

```

3 Tactic Mode

Construct proofs using explicit terms is called term-style proof. This can be tedious for complicated proofs.

Fortunately, Lean provides the tactic mode to help us construct proofs interactively.

`by` activates the tactic mode.

The tactic mode captures the way mathematicians actually think: There is a goal q to prove, and we have several hypotheses $h_1 : p_1, \dots, h_n : p_n$ in the context to use. We apply tactics to change the goal and the context until the goal is solved. This produces a proof of $p_1 \rightarrow \dots \rightarrow p_n \rightarrow q$.

```
example (hp : p) : p := by exact hp
```

tactic: `exact` If the goal is p and we have $hp : p$, then `exact hp` solves the goal.
`exact?` may help to close some trivial goals

```
example (hp : p) (hq : p → q) : q := by exact?
```

tactic: `intro` Sometimes a hypothesis is hidden in the goal in the form of an implication. If the goal is $p \rightarrow q$, then `intro hp` changes the goal to q and adds the hypothesis $hp : p$ into the context.

modus ponens, with a hidden hypothesis

```

example (hp : p) : (p → q) → q := by
  intro hq
  exact hq hp

example (hq : q) : p → q := by
  intro _ -- use '_' as a placeholder if the introduced hypothesis is not needed
  exact hq

```

modus ponens, with two hidden hypothesis

```

example : p → (p → q) → q := by
  intro hp hq -- you can `intro` multiple hypotheses at once
  exact hq hp

```

[EXR] transitivity of \rightarrow

```

example : (p → q) → (q → r) → (p → r) := by
  intro hq hqr hp
  exact hqr (hq hp)

```

tactic: `apply` If q is the goal and we have $hq : p \rightarrow q$, then `apply hq` changes the goal to p .

modus ponens

```
example (hp : p) (hpq : p → q) : q := by
  apply hpq
  exact hp
```

[EXR] transitivity of \rightarrow

```
example (hpq : p → q) (hqr : q → r) : p → r := by
  intro hp
  apply hqr
  apply hpq
  exact hp
```

[IGNORE] Above tactics are minimal and sufficient for simple proofs. When proofs went more complicated, you may want more tactics that suit your needs. Remember your favorite tactics and use them accordingly.

tactic: **specialize** If we have $hpq : p \rightarrow q$ and $hp : p$, then **specialize hpq hp** reassigned hpq to $hpq \, hp$, a proof of q .

```
example (hp : p) (hpq : p → q) : q := by
  specialize hpq hp
  exact hpq

example (hpq : p → q) (hqr : q → r) : p → r := by
  intro hp
  specialize hpq hp
  specialize hqr hpq
  exact hqr
```

tactic: **have** **have** helps you to state and prove a lemma in the middle of a proof. **have h : p := hp** adds the hypothesis $h : p$ into the context, where hp is a proof of p that you provide. **haveI** is similar to **have**, but it adds the hypothesis as **this**.

```
example (hpq : p → q) (hqr : q → r) : p → r := by
  intro hp
  have hq : q := hpq hp
  have hr : r := by -- combine with `by` is also possible
    apply hqr
    exact hq
  exact hr
```

tactic: **suffices** Say our goal is q , **suffices hp : p from hq** changes the goal to p , as long as you can provide a proof hq of q from a proof hp of p . You may also switch to the tactic mode by **suffices hp : p by ...**

```
example (hpq : p → q) (hqr : q → r) : p → r := by
  intro hp
  suffices hq : q from hqr hq
  exact hpq hp

example (hpq : p → q) (hqr : q → r) : p → r := by
  intro hp
  suffices hq : q by
```

```
apply hqr
exact hq
exact hpq hp
```

`show` (it is not a tactic!) Sometimes you want to clarify what exactly you are giving a proof for. `show p from h` make sure that `h` is interpreted as a proof of `p`. `show p by ...` switches to the tactic mode to construct a proof of `p`.

```
example (hpq : p → q) (hqr : q → r) : p → r := by
  intro hp
  exact hqr (show q by apply hpq; exact hp)

end
```