BIYSC 2021 - Notes

Release 0.1

Marc Masdeu
Roberto Rubio

Jul 21, 2021

CONTENTS

CHAPTER ONE

INTRODUCTION

1.1 What is Lean?

Lean is an open source proof-checker and a proof-assistant. One can *explain* mathematical proofs to it and it can check their correctness. It also simplifies the proof writing process by providing `goals` and `tactics`.

Lean is built on top of a formal system called type theory. In type theory, the basic notions are "terms" and "types" — compare to "elements" and "sets" in set theory. Every term has a type, and types are just a special kind of term. Terms can be interpreted as mathematical objects, functions, propositions, or proofs. The only two things Lean can do is *create* terms and *check* their types. By iterating these two operations, we can teach Lean to verify complex mathematical proofs. For instance, below you can see the statement of Fermat's Last Theorem coded in Lean, which states that the equation $x^n + y^n = z^n$ has no nontrivial solutions in the integers when $n \in \mathbb{N}$ is greater than 2.

```
def x := 2 + 2                                   -- a natural number
def f (x : ℕ) := x + 3                           -- a function
def easy_theorem_statement := 2 + 2 = 4          -- a proposition
def fermats_last_theorem_statement               -- another proposition
  :=
  ∀ n : ℕ,
  n > 2
  →
  ¬ (∃ x y z : ℕ, (x^n + y^n = z^n) ∧ (x ≠ 0) ∧ (y ≠ 0) ∧ (z ≠ 0))

theorem
easy_proof : easy_theorem_statement              -- proof of easy_theorem
:=
begin
  exact rfl,
end

theorem
my_hard_proof : fermats_last_theorem_statement     -- cheating!
:=
begin
  sorry,
end

#check x
#check f
#check easy_theorem_statement
#check fermats_last_theorem_statement
#check easy_proof
#check hard_proof
```

1.2 How to use these notes

Every once in a while, you will see a code snippet like this:

```
#eval "Hello, World!"
```

Clicking on the `try it!` button in the upper right corner will open a copy in a window so that you can edit it, and Lean provides feedback in the `Lean Infoview` window. We use this feature to provide exercises inline in the notes. We recommend attempting each exercise as you go along.

These notes are based a 5-day Lean crash course at Mathcamp 2020. We have adapted them to BIYSC 2021.

These notes provide a sneak-peek into the world of theorem proving in Lean and are by no means comprehensive. It is recommended that you simultaneously attempt the Natural Number Game. It is a fun (and highly addictive!) game that proves same basic properties of natural numbers in Lean.

1.3 Acknowledgments.

These notes are based on work of Apurva Nakade and Jalex Stark. Large chunks of these notes are taken directly from <https://apurvanakade.github.io/courses/lean_at_MC2020/>`__.

1.4 Useful Links.

1. Formalizing 100 theorems
2. Formalizing 100 theorems in Lean
3. **Articles, videos, blog posts, etc.**
    1. The Xena Project
    2. The Mechanization of Mathematics
    3. The Future of Mathematics
4. Lean Zulip chat group

CHAPTER

TWO

LOGIC IN LEAN - PART 1

Lean is built on top of a logic system called *type theory*, which is an alternative to *set theory*. In type theory, instead of elements we have *terms* and every term has a *type*. When translated to math, terms can be either mathematical objects, functions, propositions, or proofs. The notation `x : X` stands for "`x` is a term of type `X`" or "`x` is an inhabitant of `X`". For the most part, you can think of a type as a set and terms as elements of the set.

2.1 Propositions as types

In set theory, a **proposition** is any statement that has the potential of being true or false, like `2 + 2 = 4`, `2 + 2 = 5`, "Fermat's last theorem", or "Riemann hypothesis". In type theory, there is a special type called `Prop` whose inhabitants are propositions. Furthermore, each proposition `P` is itself a type and the inhabitants of `P` are its proofs!

```
P : Prop      -- P is a proposition
hp : P        -- hp is a proof of P
```

As such, in type theory "producing a proof of `P`" is the same as "producing a term of type `P`" and so a proposition `P` is `true` if there exists a term `hp` of type `P`.

Notation. Throughout these notes, `P, Q, R, ...` will denote propositions.

2.1.1 Implication

In set theory, the proposition `P ⇒ Q` ("P implies Q") is true if either both `P` and `Q` are true or if `P` is false. In type theory, a proof of an implication `P ⇒ Q` is just a function `f : P → Q`. Given a function `f : P → Q`, every proof `hp : P` produces a proof `f hp : Q`. If `P` is false then `P` is *empty*, and there exists an empty function from an empty type to any type. Hence, in type theory we use $\to$ to denote implication.

2.1.2 Negation

In type theory, there is a special proposition `false : Prop` which has no proof (hence is *empty*). The negation of a proposition `¬ P` is the implication `P → false`. Such a function exists if and only if `P` itself is empty (empty function), hence `P → false` is inhabited if and only if `P` is empty which justifies using it as the definition of `¬ P`.

To summarize:

1. Proving a proposition `P` is equivalent to producing an inhabitant `hp : P`.
2. Proving an implication `P → Q` is equivalent to producing a function `f : P → Q`.
3. The negation, `¬ P`, is defined as the implication `P → false`.

2.1.3 Propositions in Lean

In Lean, a proposition and its proof are written using the following syntax.

```
theorem fermats_last_theorem
  (n : ℕ)
  (n_gt_2 : n > 2)
  :
  ¬ (∃ x y z : ℕ, (x^n + y^n = z^n) ∧ (x ≠ 0) ∧ (y ≠ 0) ∧ (z ≠ 0))
:=
begin
  sorry,
end
```

Let us parse the above statement.

- `fermats_last_theorem` is the name of the theorem.
- `(n : ℕ)` and `(n_gt_2 : n > 2)` are the two *hypotheses*. The former says `n` is a natural number and the latter says that `n_gt_2` is a proof of `n > 2`.
- **:** is the delimiter between hypotheses and targets
- `¬ (∃ x y z : ℕ, (x^n + y^n = z^n) ∧ (x ≠ 0) ∧ (y ≠ 0) ∧ (z ≠ 0))` is the *target* of the theorem.
- `:= begin ... end` contains the proof. When you start your proof, Lean opens up a goal window for you to keep track of hypotheses and targets. **Your goal is to produce a term that has the type of the target**.

```
-- example of Lean goal window
n : ℕ, -- hypothesis 1
n_gt_2 : n > 2 -- hypothesis 2
⊢ ¬∃ (x y z : ℕ), x ^ n + y ^ n = z ^ n ∧ x ≠ 0 ∧ y ≠ 0 ∧ z ≠ 0 -- target
```

- The commands you write between `begin` and `end` are called *tactics*. `sorry,` is an example of a tactic. **Very Important:** All tactics must end with a comma (,) .

Even though they are not explicitly displayed, all the theorems in the Lean library are also hypotheses that you can use to close the goal.

2.2 Implications in Lean

We'll start learning tactics by proving implications in Lean. In the following sections, there are tables describing what a tactic does. Solve the following exercises to see the tactics in action.

The first two tactics we'll learn are `exact` and `intros`.

`exact`	If `P` is the target of the current goal and `hp` is a term of type `P`, then `exact hp,` will close the goal. Mathematically, this saying "this is *exactly* what we were required to prove".
`intro`	If the target of the current goal is a function `P → Q`, then `intro hp,` will produce a hypothesis `hp : P` and change the target to `Q`. Mathematically, this is saying that in order to define a function from `P` to `Q`, we first need to choose an arbitrary element of `P`.

```
/-----------------------------------------------------------------------

``exact``

  If ``P`` is the target of the current goal and
  ``hp`` is a term of type ``P``, then
  ``exact hp,`` will close the goal.


``intro``

  If the target of the current goal is a function ``P → Q``, then
  ``intro hp,`` will produce a hypothesis
  ``hp : P`` and change the target to  ``Q``.

Delete the ``sorry,`` below and replace them with a legitimate proof.

-----------------------------------------------------------------------/

theorem tautology (P : Prop) (hp : P) : P :=
begin
  sorry,
end

theorem tautology' (P : Prop) : P → P :=
begin
  sorry,
end

example (P Q : Prop): (P → (Q → P)) :=
begin
  sorry,
end

-- Can you find two different ways of proving the following?
example (P Q : Prop) : ((Q → P) → (Q → P)) :=
begin
  sorry,
end
```

The next two tactics are `have` and `apply`.

`have`	`have` is used to create intermediate variables. If `f` is a term of type `P → Q` and `hp` is a term of type `P`, then `have hq := f(hp),` creates the hypothesis `hq : Q.`
`apply`	`apply` is used for backward reasoning. If the target of the current goal is `Q` and `f` is a term of type `P → Q`, then `apply f,` changes target to `P`. Mathematically, this is equivalent to saying "because `P` implies `Q`, to prove `Q` it suffices to prove `P`".

Often these two tactics can be used interchangeably. Think of `have` as reasoning forward and `apply` as reasoning backward. When writing a big proof, you often want a healthy combination of the two that makes the proof readable.

```
/-----------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
``have``

  If ``f`` is a term of type ``P → Q`` and
  ``hp`` is a term of type ``P``, then
  ``have hq := f(hp),`` creates the hypothesis ``hq : Q`` .


``apply``

  If the target of the current goal is ``Q`` and
  ``f`` is a term of type ``P → Q``, then
  ``apply f,`` changes target to ``P``.

Delete the ``sorry,`` below and replace them with a legitimate proof.

-------------------------------------------------------------------------/

example (P Q R : Prop) (hp : P) (f : P → Q) (g : Q → R) : R :=
begin
  sorry,
end

example (P Q R S T U: Type)
(hpq : P → Q)
(hqr : Q → R)
(hqt : Q → T)
(hst : S → T)
(htu : T → U)
: P → U :=
begin
  sorry,
end
```

For the following exercises, recall that `¬ P` is defined as `P → false`, `¬ (¬ P)` is `(P → false) → false`, and so on.

```
/-------------------------------------------------------------------------

Recall that
  ``¬ P`` is ``P → false``,
  ``¬ (¬ P)`` is ``(P → false) → false``, and so on.

Delete the ``sorry,`` below and replace them with a legitimate proof.

-------------------------------------------------------------------------/

theorem self_imp_not_not_self (P : Prop) : P → ¬ (¬ P) :=
begin
  sorry,
end

theorem contrapositive (P Q : Prop) : (P → Q) → (¬Q → ¬P) :=
begin
  sorry,
end
```

(continues on next page)

(continued from previous page)

```
example (P : Prop) : ¬ (¬ (¬ P)) → ¬ P :=
begin
  sorry,
end
```

2.3 Proof by contradiction

You can prove exactly one of the converses of the above three using just `exact`, `intro`, `have`, and `apply`. Can you find which one?

```
/-------------------------------------------------------------------------

You can prove exactly one of the following three using just
``exact``, ``intro``, ``have``, and ``apply``.

Can you find which one?

-------------------------------------------------------------------------/

theorem not_not_self_imp_self (P : Prop) : ¬ ¬ P → P:=
begin
  sorry,
end

theorem contrapositive_converse (P Q : Prop) : (¬Q → ¬P) → (P → Q) :=
begin
  sorry,
end

example (P : Prop) : ¬ P → ¬ ¬ ¬ P :=
begin
  sorry,
end
```

This is because it is not true that `¬ ¬ P = P` *by definition*, after all, `¬ ¬ P` is `(P → false) → false` which is drastically different from `P`. There is an extra axiom called **the law of excluded middle** which says that either `P` is inhabited or `¬ P` is inhabited (and there is no *middle* option) and so `P ↔ ¬ ¬ P`. This is the axiom that allows for proofs by contradiction. Lean provides us the following tactics to use it.

ex-falso	Changes the target of the current goal to `false`. The name derives from "ex falso, quodlibet" which translates to "from contradiction, anything". You should use this tactic when there are contradictory hypotheses present.
by_cases	If `P : Prop`, then `by_cases P,` creates two goals, the first with a hypothesis `hp: P` and second with a hypothesis `hp: ¬ P`. Mathematically, this is saying either `P` is true or `P` is false. `by_cases` is the most direct application of the law of excluded middle.
by_contradiction	If the target of the current goal is `Q`, then `by_contradiction,` changes the target to `false` and adds `hnq : ¬ Q` as a hypothesis. Mathematically, this is proof by contradiction.
push_neg	`push_neg,` simplifies negations in the target. For example, if the target of the current goal is `¬ ¬ P`, then `push_neg,` simplifies it to `P`. You can also push negations across a hypothesis `hp : P` using `push_neg at hp,`.
contrapose!	If the target of the current goal is `P → Q`, then `contrapose!,` changes the target to `¬ Q → ¬ P`. If the target of the current goal is `Q` and one of the hypotheses is `hp : P`, then `contrapose! hp,` changes the target to `¬ P` and changes the hypothesis to `hp : ¬ Q`. Mathematically, this is replacing the target by its contrapositive.

Even though the list is long, these tactics are almost all *obvious*. The only two slightly unusual tactics are `exfalso` and `by_cases`. You'll see `by_cases` in action later. For the following exercises, you only require `exfalso`, `push_neg`, and `contrapose!`.

```
/-------------------------------------------------------------------------

``exfalso``

  Changes the target of the current goal to ``false``.

``push_neg``

  ``push_neg,`` simplifies negations in the target.
  You can push negations across a hypothesis ``hp : P`` using
  ``push_neg at hp,``.


``contrapose!``

  If the target of the current goal is  ``P → Q``,
  then ``contrapose!,`` changes the target to  ``¬ Q → ¬ P``.

  If the target of the current goal is ``Q`` and
  one of the hypotheses is ``hp : P``, then
  ``contrapose! hp,`` changes the target to  ``¬ P`` and
  changes the hypothesis to ``hp : ¬ Q``.


Delete the ``sorry,`` below and replace them with a legitimate proof.

-------------------------------------------------------------------------/

theorem not_not_self_imp_self (P : Prop) : ¬ ¬ P → P:=
begin
  sorry,
end
```

(continues on next page)

(continued from previous page)

```
theorem contrapositive_converse (P Q : Prop) : (¬Q → ¬P) → (P → Q) :=
begin
  sorry,
end

example (P : Prop) : ¬ P → ¬ ¬ ¬ P :=
begin
  sorry,
end

theorem principle_of_explosion (P Q : Prop) : P → (¬ P → Q) :=
begin
  sorry,
end
```

2.4 Geometry

Finally, let's do some geometry! We will introduce the incidence axioms, and start proving some lemmas from them.

```
constants Point Line : Type*
constant belongs : Point → Line → Prop
local notation A `∈` L := belongs A L
local notation A `∉` L := ¬ belongs A L
```

Here is how we can introduce axioms.

```
-- I1: there is a unique line passing through two distinct points.
axiom I1 (A B : Point) (h : A ≠ B) : ∃! (ℓ : Line) , A ∈ ℓ ∧ B ∈ ℓ

-- I2: any line contains at least two points.
axiom I2 (ℓ : Line) : ∃ A B : Point, A ≠ B ∧ A ∈ ℓ ∧ B ∈ ℓ

-- I3: there exists 3 non-collinear points.
axiom I3 : ∃ A B C : Point, (A ≠ B ∧ A ≠ C ∧ B ≠ C ∧ (∀ ℓ : Line, (A ∈ ℓ ∧ B ∈ ℓ) →␣
↪(¬ (C ∈ ℓ) )))
```

Axiom I3 really says that there are 3 non-collinear points. We can make actually define what it means to be collinear and prove a statement which is easier to remember.

```
-- We can make our own definitions
def collinear (A B C : Point) : Prop := ∃ (ℓ : Line), (A ∈ ℓ ∧ B ∈ ℓ ∧ C ∈ ℓ)

-- So let's prove that axiom I3 really says that there are 3 non-collinear points
example : ∃ A B C : Point, ¬ collinear A B C :=
begin
  sorry
end
```

In the morning we proved quite in detail the following theorem (we called Theorem 1). Before trying to prove it, make sure that the *Lean* statement is really what the English sentence says.

```
-- Two distinct lines meet at most at one point
example (r s : Line) (h : r ≠ s) (A B : Point) : A ∈ r ∧ B ∈ r ∧ A ∈ s ∧ B ∈ s → A␣
↪= B :=
```

(continues on next page)

(continued from previous page)

```
begin
  sorry
end
```

Let's prove another useful lemma: given a line, there is a point outside it.

```
-- Use I3 to prove the following lemma
lemma exists_point_not_on_line (ℓ : Line): ∃ A : Point, A ∉ ℓ :=
begin
  sorry
end

-- Challenge: is it true for two lines? If so, prove it
lemma exists_point_not_on_two_line (r s : Line): ∃ A : Point, A ∉ r ∧ A ∉ s :=
begin
  sorry
end
```

LOGIC IN LEAN - PART 2

Your mission today is to wrap up the remaining bits of logic and move on to doing some "actual math". Remember to **always save your work**. You might find the *Glossary of tactics* page and the *Pretty symbols* page useful.

Before we move on to new stuff, let's understand what we did yesterday.

3.1 Behind the scenes

A note on brackets: It is not uncommon to compose half a dozen functions in Lean. The brackets get really messy and unwieldy. As such, Lean will often drop the brackets by following the following conventions.

- The function `P → Q → R → S` stands for `P → (Q → (R → S))`.
- The expression `a + b + c + d` stands for `((a + b) + c) + d`.

An easy way to remember this is that, arrows are bracketed on the right and binary operators on the left.

3.1.1 Proof irrelevance

It might feel a bit weird to say that a proposition has proofs as its inhabitants. Proofs can get huge and it seems unnecessary to have to remember not just the statement but also its proof. This is something we don't normally do in math. To hide this complication, in type theory there is an axiom, called *proof irrelevance*, which says that if `P : Prop` and `hp1 hp2 : P` then `hp1 = hp2`. Taking our *analogy* with sets further, you can think of a proposition as a set which is either empty or contains a single element (false or true). In fact, in some forms of type theory (e.g. homotopy type theory) this is taken as the definition of propositions. This is of course not true for general types. For example, `0 : ℕ ≠ 1 : ℕ`.

3.1.2 Proofs as functions

Every time you successfully construct a proof of a theorem say

```
theorem tautology (P : Prop) : P → P :=
begin
  intro hp,
  exact hp,
end
```

Lean constructs a *proof term* `tautology : ∀ P : Prop, P → P` (you can see this by typing `#check tautology`).

In type theory, the *for all* quantifier, ∀, is a generalized function, called a dependent function. For all practical purposes, we can think of `tautology` as having the type `(P : Prop) → (P → P)`. Note that this is not a function in the classical sense of the word because the codomain `(P → P)` *depends* on the input variable `P`. If `Q : Prop`, then `tautology(Q)` is a term of type `Q → Q`.

Consider a theorem with multiple hypothesis, say

```
theorem hello_world (hp : P) (hq : Q) (hr : R) : S
```

Once we provide a proof of it, Lean will create a proof term `hello_world : (hp:P) → (hq:Q) → (hr:R) → S`. So that if we have terms `hp' : P`, `hq' : Q`, `hr' : R` then `hello_world hp' hq' hr'` (note the convenient lack of brackets) will be a term of type `S`.

Once constructed, any term can be used in a later proof. For example,

```
example (P Q : Prop) : (P → Q) → (P → Q) :=
begin
  exact tautology (P → Q),
end
```

This is how Lean simulates mathematics. Every time you prove a theorem using tactics a *proof term* gets created. Because of proof irrelevance, Lean forgets the exact content of the proof and only remembers its type. All the proof terms can then be used in later proofs. All of this falls under the giant umbrella of the Curry–Howard correspondence.

We'll now continue our study of the remaining logical operators: *and* (∧), *or* (∨), *if and only if* (↔), *for all* (∀), *there exists* (∃).

3.2 And / Or

The operators *and* (∧) and *or* (∨) are very easy to use in Lean. Given a term `hpq : P ∧ Q`, there are tactics that let you create terms `hp : P` and `hq : Q`, and vice versa. Similarly for `P ∨ Q`, with a subtle change (see below).

Note that when multiple goals are open, you are trying to solve the topmost goal.

`cases`	`cases` is a general tactic that breaks a complicated term into simpler ones. If `hpq` is a term of type `P ∧ Q`, then `cases hpq with hp hq,` breaks it into `hp : P` and `hp : Q`. If `fg` is a term of type `P ↔ Q`, then `cases fg with f g,` breaks it into `f : P → Q` and `g : Q → P`. If `hpq` is a term of type `P ∨ Q`, then `cases hpq with hp hq,` creates two goals and adds the hypotheses `hp : P` and `hq : Q` to one each.
`split`	`split` is a general tactic that breaks a complicated goal into simpler ones. If the target of the current goal is `P ∧ Q`, then `split,` breaks up the goal into two goals with targets `P` and `Q`. If the target of the current goal is `P × Q`, then `split,` breaks up the goal into two goals with targets `P` and `Q`. If the target of the current goal is `P ↔ Q`, then `split,` breaks up the goal into two goals with targets `P → Q` and `Q → P`.
`left`	If the target of the current goal is `P ∨ Q`, then `left,` changes the target to `P`.
`right`	If the target of the current goal is `P ∨ Q`, then `right,` changes the target to `Q`.

```
/-----------------------------------------------------------------------------

``cases``

  ``cases`` is a general tactic that breaks up complicated terms.
```

(continues on next page)

(continued from previous page)

```
  If ``hpq`` is a term of type ``P ∧ Q`` or ``P ∨ Q`` or ``P ↔ Q``, then use
  ``cases hpq with hp hq,``.

``split``

  If the target of the current goal is ``P ∧ Q`` or ``P ↔ Q``, then use
  ``split,``.

``left``/``right``

  If the target of the current goal is ``P ∨ Q``, then use
  either ``left,`` or ``right,`` (choose wisely).

``exfalso``

  Changes the target of the current goal to ``false``.

Delete the ``sorry,`` below and replace them with a legitimate proof.

--------------------------------------------------------------------------/

example (P Q : Prop) : P ∧ Q → Q ∧ P :=
begin
  sorry,
end

example (P Q : Prop) : P ∨ Q → Q ∨ P :=
begin
  sorry,
end

example (P Q R : Prop) : P ∧ false ↔ false :=
begin
  sorry,
end

theorem principle_of_explosion (P Q : Prop) : P ∧ ¬ P → Q :=
begin
  sorry,
end
```

3.3 Quantifiers

As mentioned it the introduction the *for all* quantifier, ∀, is a generalization of a function. As such the tactics for dealing with ∀ are the same as those for →.

`have`	If `hp` is a term of type `∀ x : X, P x` and `y` is a term of type `X` then `have hpy := hp(y)` creates a hypothesis `hpy : P y`.
`intro`	If the target of the current goal is `∀ x : X, P x`, then `intro x,` creates a hypothesis `x : X` and changes the target to `P x`.

The *there exists* quantifier, ∃, in type theory is very intuitive. If you want to prove a statement `∃ x : X, P x` then you need to provide a witness. If you have a term `hp : ∃ x : X, P x` then from this you can extract a witness.

`cases`	If `hp` is a term of type `∃ x : X, P x`, then `cases hp with x key,` breaks it into `x : X` and `key : P x`.
`use`	If the target of the current goal is `∃ x : X, P x` and `y` is a term of type `X`, then `use y,` changes the target to `P y` and tries to close the goal.

3.4 Geometry

Now it's your turn! Introduce Hilbert's axioms for between-ness. We'll give you the ones for incidence from yesterday.

```
import tactic
constants Point Line : Type*
constant belongs : Point → Line → Prop
local notation A `∈` L := belongs A L
local notation A `∉` L := ¬ belongs A L

-- I1: there is a unique line passing through two distinct points.
axiom I1 (A B : Point) (h : A ≠ B) : ∃! (ℓ : Line) , A ∈ ℓ ∧ B ∈ ℓ

-- I2: any line contains at least two points.
axiom I2 (ℓ : Line) : ∃ A B : Point, A ≠ B ∧ A ∈ ℓ ∧ B ∈ ℓ

-- I3: there exists 3 non-collinear points.
axiom I3 : ∃ A B C : Point, (A ≠ B ∧ A ≠ C ∧ B ≠ C ∧ (∀ ℓ : Line, (A ∈ ℓ ∧ B ∈ ℓ) →
↪(¬ (C ∈ ℓ) )))

-- We can make our own definitions
def collinear (A B C : Point) : Prop := ∃ (ℓ : Line), (A ∈ ℓ ∧ B ∈ ℓ ∧ C ∈ ℓ)

-- We define the between-ness relation
constant between : Point → Point → Point → Prop
local notation A `*` B `*` C := between A B C
```

CHAPTER

FOUR

PRETTY SYMBOLS IN LEAN

To produce a pretty symbol in Lean, type the *editor shortcut* followed by space or tab.

Unicode	Editor Shortcut	Definition
→	`\to`	function or implies
↔	`\iff`	if and only if
←	`\l`	used by the `rw` tactic
¬	`\not`	negation operator
∧	`\and`	and operator
∨	`\or`	or operator
∃	`\exists`	there exists quantifier
∀	`\forall`	for all quantifier
ℕ	`\nat`	type of natural numbers
ℤ	`\int`	type of integers
∘	`\circ`	composition of functions
≠	`\ne`	not equal to
∈	`\in`	belongs to
∉	`\notin`	does not belong to
∟	`\angle`	angle
▵	`\triangle`	triangle
≅	`\cong`	congruence of segments
≃	`\simeq`	congruence of angles

CHAPTER

FIVE

GLOSSARY OF TACTICS

5.1 Implications in Lean

`exact`	If `P` is the target of the current goal and `hp` is a term of type `P`, then `exact hp,` will close the goal. Mathematically, this saying "this is *exactly* what we were required to prove".
`intro`	If the target of the current goal is a function `P → Q`, then `intro hp,` will produce a hypothesis `hp : P` and change the target to `Q`. Mathematically, this is saying that in order to define a function from `P` to `Q`, we first need to choose an arbitrary element of `P`.

`have`	`have` is used to create intermediate variables. If `f` is a term of type `P → Q` and `hp` is a term of type `P`, then `have hq := f hp,` creates the hypothesis `hq : Q`.
`apply`	`apply` is used for backward reasoning. If the target of the current goal is `Q` and `f` is a term of type `P → Q`, then `apply f,` changes target to `P`. Mathematically, this is equivalent to saying "because `P` implies `Q`, to prove `Q` it suffices to prove `P`".

5.2 Proof by contradiction

`ex-falso`	Changes the target of the current goal to `false`. The name derives from "ex falso, quodlibet" which translates to "from contradiction, anything". You should use this tactic when there are contradictory hypotheses present.
`by_cases`	If `P : Prop`, then `by_cases P,` creates two goals, the first with a hypothesis `hp: P` and second with a hypothesis `hp: ¬ P`. Mathematically, this is saying either `P` is true or `P` is false. `by_cases` is the most direct application of the law of excluded middle.
`by_contradiction`	If the target of the current goal is `Q`, then `by_contradiction,` changes the target to `false` and adds `hnq : ¬ Q` as a hypothesis. Mathematically, this is proof by contradiction.
`push_neg`	`push_neg,` simplifies negations in the target. For example, if the target of the current goal is `¬ ¬ P`, then `push_neg,` simplifies it to `P`. You can also push negations across a hypothesis `hp : P` using `push_neg at hp,`.
`contrapose!`	If the target of the current goal is `P → Q`, then `contrapose!,` changes the target to `¬ Q → ¬ P`. If the target of the current goal is `Q` and one of the hypotheses is `hp : P`, then `contrapose! hp,` changes the target to `¬ P` and changes the hypothesis to `hp : ¬ Q`. Mathematically, this is replacing the target by its contrapositive.

5.3 And / Or

`cases`	`cases` is a general tactic that breaks a complicated term into simpler ones. If `hpq` is a term of type `P ∧ Q`, then `cases hpq with hp hq,` breaks it into `hp : P` and `hp : Q`. If `hpq` is a term of type `P × Q`, then `cases hpq with hp hq,` breaks it into `hp : P` and `hp : Q`. If `fg` is a term of type `P ↔ Q`, then `cases fg with f g,` breaks it into `f : P → Q` and `g : Q → P`. If `hpq` is a term of type `P ∨ Q`, then `cases hpq with hp hq,` creates two goals and adds the hypotheses `hp : P` and `hq : Q` to one each.
`split`	`split` is a general tactic that breaks a complicated goal into simpler ones. If the target of the current goal is `P ∧ Q`, then `split,` breaks up the goal into two goals with targets `P` and `Q`. If the target of the current goal is `P × Q`, then `split,` breaks up the goal into two goals with targets `P` and `Q`. If the target of the current goal is `P ↔ Q`, then `split,` breaks up the goal into two goals with targets `P → Q` and `Q → P`.
`left`	If the target of the current goal is `P ∨ Q`, then `left,` changes the target to `P`.
`right`	If the target of the current goal is `P ∨ Q`, then `right,` changes the target to `Q`.

5.4 Quantifiers

`have`	If `hp` is a term of type `∀ x : X, P x` and `y` is a term of type `y` then `have hpy := hp(y)` creates a hypothesis `hpy : P y`.
`intro`	If the target of the current goal is `∀ x : X, P x`, then `intro x,` creates a hypothesis `x : X` and changes the target to `P x`.

`cases`	If `hp` is a term of type `∃ x : X, P x`, then `cases hp with x key,` breaks it into `x : X` and `key : P x`.
`use`	If the target of the current goal is `∃ x : X, P x` and `y` is a term of type `X`, then `use y,` changes the target to `P y` and tries to close the goal.

5.5 Proving "trivial" statements

`norm_num`	`norm_num` is Lean's calculator. If the target has a proof that involves *only* numbers and arithmetic operations, then `norm_num` will close this goal. If `hp : P` is an assumption then `norm_num at hp,` tries to use simplify `hp` using basic arithmetic operations.
`ring`	`ring,` is Lean's symbolic manipulator. If the target has a proof that involves *only* algebraic operations, then `ring,` will close the goal. If `hp : P` is an assumption then `ring at hp,` tries to use simplify `hp` using basic algebraic operations.
`linarith`	`linarith,` is Lean's inequality solver.
`simp`	`simp,` is a very complex tactic that tries to use theorems from the mathlib library to close the goal. You should only ever use `simp,` to *close a goal* because its behavior changes as more theorems get added to the library.

5.6 Equality

`refl`	If the current goal is of the form `X = X` or `P ↔ P`, then `refl` will finish the proof. As long as both sides are defined to be equal, this will work. For example, it will work with the goal `3 = 2 + 1` because *by definition* the number 3 is defined to be 2 plus one. Mathematically, this says "check that both sides are qual *by definition*".

`rw`	If `f` is a term of type `P = Q` (or `P ↔ Q`), then
	`rw f,` searches for `P` in the target and replaces it with `Q`.
	`rw ←f,` searches for `Q` in the target and replaces it with `P`.
	If additionally, `hr : R` is a hypothesis, then
	`rw f at hr,` searches for `P` in the expression `R` and replaces it with `Q`.
	`rw ←f at hr,` searches for `Q` in the expression `R` and replaces it with `P`.
	Mathematically, this is saying because `P = Q`, we can replace `P` with `Q` (or the other way around).