Lean at MC2022

Release 0.1

Aaron Anderson, Apurva Nakade
Jalex Stark

Jun 27, 2022

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.

```
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
  rw easy_theorem_statement,                      -- a tactic
end

theorem
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 designed for a 5-day Lean crash course at Mathcamp 2022, based on a similar class at Mathcamp 2020. On Days 1 and 2 you'll learn the basics of type theory and some basic `tactics` in Lean. On Days 3, 4, 5 you'll use these to prove increasingly complex theorems, namely the infinitude of primes and irrationality of $\sqrt{2}$.

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 at least one of the following two options.

1. Play the Natural Number Game.
2. Read Theorem Proving in Lean.

The Natural Number Game is a fun (and highly addictive!) game that proves same basic properties of natural numbers in Lean. Theorem Proving in Lean is a comprehensive online book that aims to cover all the theorem proving aspects of Lean in great detail.

The Lean community is very welcoming to newcomers, and people are available on the Lean Zulip chat group round the clock to answer questions. You can also join Kevin Buzzard's Discord server which has a relatively younger crowd.

1.3 Acknowledgments.

These notes are developed by Aaron Anderson, updating notes for Mathcamp 2020 by Apurva Nakade and Jalex Stark with a lot of help from Mathcamp campers and Mathcamp staff Joanna and Maya. Large chunks of these notes are taken from various learning resources available on the leanprover-community website.

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. Kevin Buzzard's Twitch channel. In particular, checkout this video about summer projects.
4. Discord server
5. Lean Zulip chat group

CHAPTER

TWO

LOGIC IN LEAN - PART 1

Today's goal is to understand the philosophy of type theory (in Lean). Don't try to memorize anything, that will happen automatically. Instead, try to do as many exercises as you can. Practice is the only way to learn a new programming language. And **always save your work**. The easiest way to do this in the browser is by bookmarking the Lean page, which contains your code in its URL.

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 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. (Lean ignores multiple whitespaces, tabs, and new lines. You could theoretically write the entire code in a single line. Please don't.)

- `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. We'll learn all these symbols soon.
- `:= 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.

2.1.2 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 denote implication. (Type it in Lean editors with `\to`.)

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 `refine` and `rintro`.

`refine`	If `P` is the target of the current goal and `hp` is a term of type `P`, then `refine hp,` will close the goal. Mathematically, this saying "this is what we were required to prove".
`rintro`	If the target of the current goal is a function `P → Q`, then `rintro 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 (introduce) an arbitrary element of `P`. If you want to use this repeatedly, you can type `rintro h1 h2` instead of `rintro h1,` and then `rintro h2,`.

```
import tactic
/-------------------------------------------------------------------------

``refine``

  If ``P`` is the target of the current goal
```

(continues on next page)

(continued from previous page)

```
  and ``hp`` is a term of type ``P``,
  then ``refine hp,`` will close the goal.


``rintro``

  If the target of the current goal is a function ``P → Q``, then
  ``rintro 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
```

We know how to start a proof, and how to finish a proof, but what about partial progress? Here's two approaches. One uses a new tactic, `have`, for forward reasoning, and the other uses `refine` again for backward reasoning.

In both of these cases, if `f` is a term of type `P → Q`, then we can think of `f` as a function, sending proofs of `P` to proofs of `Q`. If `hp` is a term of type `P`, we can literally write `f (hp)`, although often we can skip the parentheses and just write `f hp`.

`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`.
`refine`	`refine` can be used for backward reasoning. If the target of the current goal is `Q` and `f` is a term of type `P → Q`, then `refine f _,` changes target to `P`. Mathematically, this is equivalent to saying "because `P` implies `Q`, to prove `Q` it suffices to prove `P`". The _ stands in for a proof of `P` that we will provide later.

Often these two tactics can be used interchangeably. When writing a big proof, you often want a healthy combination of the two that makes the proof readable.

```
import tactic
/-------------------------------------------------------------------------

``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`` .


``refine``

  If the target of the current goal is ``Q`` and
     ``f`` is a term of type ``P → Q``, then
     ``refine 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
```

We will be learning a lot of tactics this week. If ever you lose track of them, check out the *Glossary of tactics*, which lists all of the tactics that are mentioned in these notes, as well as some others which are not needed for this class, but may come up if you read other code in Lean.

2.3 And / Or

The operators *and* (∧) and *or* (∨) are easy to use in Lean. (You can type them in Lean editors with `\and` and `\or`.) 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. The easiest way to keep track of multiple goals is with brackets. After you use a tactic with multiple goals, you should use `{ },` to bracket off your attempt to solve the first goal, and `{ },` to bracket off your second goal. Then if you put your cursor in between the brackets, the goal monitor on the right should only display one goal at a time!

`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`. (This is because `P ↔ Q` is actually shorthand for `(P → Q) ∧ (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 → 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.
  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).

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

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

theorem bracket_example (P Q : Prop) (hp : P) (hq : Q) : P ∧ Q :=
begin
  split,
  {
    sorry,
  },
  {
    sorry,
  }
end

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

example (P Q : Prop) : P ∨ Q → Q ∨ P :=
begin
```

(continues on next page)

(continued from previous page)

```
  sorry,
end
```

2.3.1 Optional Sidenote on Brackets

We've discussed that building a term of type `P` is pretty much the same thing as providing a proof of `P`. We've also seen that if you want to provide a term of type `P ∧ Q`, all you need is a term `hp : P`, a term `hq : Q`, and the `split` tactic. However, you don't *need* the `split` tactic for this, you can also build the term directly, using the angle brackets ⟨⟩, typed with `\langle` and `\rangle`. For example:

```
example (P Q : Prop) (hp : P) (hq : Q) : P ∧ Q :=
begin
  refine ⟨hp, hq⟩,
end
```

This works because `⟨hp, hq⟩` is a term of type `P ∧ Q`, because Lean defines `P ∧ Q` to be the type of ordered pairs, consisting of a term of type `P` and then a term of type `Q`. If you want to explore this, try using this to rewrite your above proofs that use ∧. (If you do, what does `refine ⟨_, _⟩,` do?)

2.4 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`. (Type ¬ it as `\not`.)

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`.

For the following exercises, recall that `¬ P` is defined as `P → false`, `¬ (¬ P)` is `(P → false) → false`, and so on. Here are some hints if you get stuck.

```
import tactic
/-------------------------------------------------------------------------

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
```

(continues on next page)

(continued from previous page)

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

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

Now that we're working with negations, we can start to talk about everybody's favorite or least favorite proof technique, contradiction. Or at least, a version of it called the "Principle of Explosion". This says that you can derive any fact from a contradiction. In Lean, this is written as `false → P`, and whenever you need it, there is a hypothesis `false.elim : false → P`, which works *for any* `P : Prop`.

```
import tactic
/-------------------------------------------------------------------------

Recall that for any ``P : Prop``, you can use ``false.elim : false → P``
  to prove ``P`` from a contradiction.

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

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

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

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

2.5 Final Remarks

You might be wondering, if type theory is so cool why have I not heard of it before?

Many programming languages highly depend on type theory (that's where the term `datatype` comes from). Once you define a term `x : ℕ`, a computer can immediately check that all the manipulations you do with `x` are valid manipulations of natural numbers (so you don't accidentally divide by 0[1] , for example).

Unfortunately, this also means that the term `1 : ℕ` is different from the term `1 : ℤ`. In Lean, if you do `(1 : ℕ - 2 : ℕ)` you get `0 : ℕ` but if you do `(1 : ℤ - 2 : ℤ)` you get `-1 : ℤ`, that's because natural numbers and subtraction are not buddies. Another issue is that `1 : ℕ = 1 : ℤ` is not a valid statement in type theory. This is not the end of the world though. Lean allows you to *coerce* `1 : ℕ` to `1 : ℤ` if you want subtraction to work properly, or `1 : ℕ` to `1 : ℚ` if you want division to work properly.

This, and a few other such things, is what drives most mathematicians away from type theory. But these things are only difficult when you're first learning them. With practice, type theory becomes second nature, the same as set theory. In fact, the exact type theoretic system Lean uses is *equiconsistent* with a slightly stronger version of ZFC, the generally-accepted axiom system for set theory. (See Mario Carneiro's MS thesis)

[1] Except under staff supervision.

footnotes

CHAPTER

THREE

LOGIC IN LEAN - PART 2

The goal 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
  rintro hp,
  refine 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
  refine 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.

3.1.3 Optional Sidenote on Lambda

Speaking of generalized functions, and terms, we can define the term `tautology` directly, without using `rintro`:

```
theorem tautology (P : Prop) : P → P :=
begin
  refine λ hp, hp,
end
```

The λ, typed `\lambda`, plays basically the role of `rintro`. In general, the term `λ x, y` will define a (generalized) function that on input `x`, gives output `y`. For instance, once we can talk about addition, `λ x, x + 2` will be the function that adds 2 to a given natural number. If you want to, you can play around with using λ and `rintro` interchangeably.

3.2 The Law of the Excluded Middle

You can prove exactly one of the following using just `refine`, `rintro`, and `have`. Can you find which one?

```
import tactic

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

You can prove exactly one of the following three using just
``refine``, ``rintro``, and ``have``.

Can you find which one?

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

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
```

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`. Lean gives it to us in the form of `em P : P ∨ ¬ P`, although it's not always included. Because some mathematicians would prefer to avoid using this in their proofs, you have to type the lines `noncomputable theory` and `open_locale classical` near the top of the file, to show that you're ok with using all of classical logic!

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

``em``

  If ``P : Prop``, then ``em P : P ∨ ¬ P`` lets you use the law of the excluded␣
↪middle on ``P``.

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

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

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

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

There are more specialized tactics that combine `false.elim` and `em` with other tactics to streamline the process of dealing with negations. You can read about them at *Glossary of tactics*, and if you want, you can try to shorten some of your above proofs with them.

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 →. (Type it as `\forall`.)

`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`.
`rintro`	If the target of the current goal is `∀ x : X, P x`, then `rintro x,` creates a hypothesis `x : X` and changes the target to `P x`.

The *there exists* quantifier, ∃, in type theory, uses similar tools to 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. (Type it as `\exists`.)

`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.

Finally, we know enough Lean to start doing some fun stuff.

3.3.1 Barber paradox

Let's disprove the "barber paradox" due to Bertrand Russell. The claim is that in a certain town there is a (male) barber that shaves all the men who do not shave themselves. (Why is this a paradox?) Prove that this is a contradiction. Here are some hints if you get stuck.

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

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

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

-- men is type.
-- x : men means x is a man in the town
-- shaves x y is inhabited if x shaves y

variables (men : Type) (barber : men)
variable  (shaves : men → men → Prop)

example : ¬ (∀ x : men, shaves barber x ↔ ¬ shaves x x) :=
  begin
    sorry,
  end
```

3.3.2 Mathcampers singing paradox

Assume that the main lounge is non-empty. At a fixed moment in time, there is someone in the lounge such that, if they are singing, then everyone in the lounge is singing. (See hints).

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

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

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

-- camper is a type.
-- If x : camper then x is a camper in the main lounge.
-- singing(x) is inhabited if x is singing

theorem math_campers_singing_paradox
  (camper : Type)
  (singing : camper → Prop)
  (alice : camper) -- making sure that there is at least one camper in the lounge
  : ∃ x : camper, (singing x → (∀ y : camper, singing y)) :=
begin
  sorry,
end
```

3.3.3 Relationship conundrum

A relation `r` on a type `X` is a map `r : X → X → Prop`. We say that `x` is *related* to `y` if `r x y` is inhabited.

- `r` is reflexive if `∀ x : X`, `x` is related to itself.
- `r` is symmetric if `∀ x y : X`, `x` is related to `y` implies `y` is related to `x`.
- `r` is transitive if `∀ x y z : X`, `x` is related to `y` and `y` is related to `x` implies `z` is related to `z`.
- `r` is connected if for all `x : X` there is a `y : Y` such that `x` is related to `y`.

Show that if a relation is symmetric, transitive, and connected, then it is also reflexive.

```
import tactic

variable X : Type

theorem reflexive_of_symmetric_transitive_and_connected
  (r : X → X → Prop)
  (h_symm : ∀ x y : X, r x y → r y x)
  (h_trans : ∀ x y z : X, r x y → r y z → r x z)
  (h_connected : ∀ x, ∃ y, r x y)
: (∀ x : X, r x x) :=
begin
  sorry,
end
```

3.4 Equality

So far we have not seen how to deal with propositions of the form `P = Q`, for example, `1 + 2 + ... + n = n(n + 1)/2`. Proving these propositions by hand requires messing around with the axioms of type theory. *Using* equalities on the other hand is very easy. The rewrite tactic (usually shortened to `rw`) let's you replace the left hand side of an equality with the right hand side.

`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`. Additionally, if `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)".

To get the left arrow, type `\l`. If you want to rewrite a bunch of things in a row, you can type `rw [h1, h2, h3],`.

```
import tactic data.nat.basic
open nat

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

  ``rw``

    If ``f`` is a term of type ``P = Q`` (or ``P ↔ Q``), then
    ``rw f`` replaces ``P`` with ``Q`` in the target.
    Other variants:
      ``rw f at hp``, ``rw ←f``, ``rw ←f at hr``.

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

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

theorem add_self_self_eq_double
  (x : ℕ)
: x + x = 2 * x :=
begin
  rw two_mul,
end

/-
For the following problem, use
  mul_comm a b : a * b = b * a
-/

example (a b c d : ℕ)
  (hyp : c = d * a + b)
  (hyp' : b = a * d)
: c = 2 * (a * d) :=
begin
  sorry,
end

/-
For the following problem, use
```

(continues on next page)

(continued from previous page)

```
  nat.sub_self (x : ℕ) : x - x = 0
-/

example (a b c d : ℕ)
  (hyp : c = b * a - d)
  (hyp' : d = a * b)
: c = 0 :=
begin
  sorry,
end
```

3.4.1 Surjective functions

Recall that a function `f : X → Y` is surjective if for every `y : Y` there exists a term `x : X` such that `f(x) = y`. In type theory, for every function `f` we can define a corresponding proposition `surjective (f) := ∀ y, ∃ x, f x = y` and a function being surjective is equivalent to saying that the proposition `surjective(f)` is inhabited.

```
import tactic
open function

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

``rw``

  If it gets hard to keep track of the definition of ``surjective``,
  you can use ``rw surjective,`` or ``rw surjective at h,``
  to get rid of it. (This rewrites using the definition of surjective).
  Typing ``rw surjective at *,`` will unfold it
  everywhere at once.

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

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

variables X Y Z : Type
variables (f : X → Y) (g : Y → Z)

/-
surjective (f : X → Y) := ∀ y, ∃ x, f x = y

You may also want to try ``function.comp_app``
-/

example
  (hf : surjective f)
  (hg : surjective g)
  : surjective (g ∘ f) :=
begin
  sorry,
end

example
  (hgf : surjective (g ∘ f))
  : surjective g :=
begin
```

(continues on next page)

(continued from previous page)

```
  sorry,
end
```

CHAPTER

FOUR

INFINITELY MANY PRIMES

Today we will prove that there are infinitely many primes using mathlib library. Our focus will be on how to *use* the library to prove more complicated theorems. Remember to always **save your work**. First, we're going to need to learn how Lean deals with divisibility of natural numbers.

4.1 Divisibility and Primes

In mathlib, divisibility for natural numbers is defined as the following *proposition*.

```
a ∣ b := (∃ k : ℕ, a = b * k)
```

For example, `2 ∣ 4` will be a proposition `∃ k : ℕ, 4 = 2 * k`. **Very important.** The statement `2 ∣ 4` is not saying that "2 divides 4 *is true*". It is simply a proposition that requires a proof. **Warning:** If you need to type the divisibility symbol, type `\mid`. This is **not** the vertical line on your keyboard.

Similarly, the mathlib library also contains a definition of `prime`. It's a little complicated, but the library has this theorem connecting it back to the definition you know:

```
theorem nat.prime_def_lt'' {p : ℕ} :
  nat.prime p ↔
    2 ≤ p                                    -- p is at least 2
    ∧                                        -- and
    ∀ {m : ℕ}, m ∣ p, m = 1 ∨ m = p          -- if m divides p, then m = 1 or m = p.
```

Same as with divisibility, for every natural number `n`, `nat.prime n` is a *proposition*. So that `nat.prime 101` requires a proof. It is possible to go down the rabbit hole and prove it using just the axioms of natural numbers. However, this is exhausting work, and exactly the kind of thing we'd rather the computer do!

4.2 Trivial calculations

Here are two of Lean's many tactics that automate basic calculations for us.

`norm_num`	`norm_num` is Lean's calculator. If the target has a proof that involves *only* natural numbers and arithmetic operations, then `norm_num` will close this goal. This is usually the most powerful tactic for dealing with natural numbers. If `hp : P` is an assumption then `norm_num at hp,` tries to simplify `hp` using basic arithmetic operations.
`ring_nf`	`ring_nf,` is Lean's algebraic manipulator. If the target has a proof that involves *only* addition and multiplication, then `ring_nf,` will close the goal. If `hp : P` is an assumption then `ring_nf at hp,` tries to simplify `hp` using basic algebraic operations.
`linarith`	`linarith,` is Lean's inequality solver. It can read and use your hypotheses, and can sometimes also solve facts that aren't explicitly about inequalities.

```
import tactic data.nat.prime

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

``norm_num``

  Useful for arithmetic of natural numbers.

``ring_nf``

  Useful for basic algebra with + and *.

``linarith``

  Useful for inequalities.


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

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

example : 1 > 0 :=
begin
  sorry,
end

example : 101 ∣ 2020 :=
begin
  sorry,
end

example : nat.prime 101 :=
begin
  sorry,
end

example (m a b : ℕ) :  m^2 + (a + b) * m + a * b = (m + a) * (m + b) :=
begin
  sorry,
end

example (a b c : ℕ) : a < b → b ≤ c → a < c :=
begin
```

(continues on next page)

(continued from previous page)

```
  sorry,
end

example (m a b : ℕ) :  m + a ∣ m^2 + (a + b) * m + a * b :=
begin
  sorry,
end

-- try ``rw nat.prime_def_lt'' at hp,`` to get started
example (p : ℕ) (hp : nat.prime p) : ¬ (p = 1) :=
begin
  sorry,
end

example (a b : ℕ) : ¬ a ≤ b → b < a :=
begin
  sorry,
end
```

4.3 Creating subgoals

Often when we write a long proof in math, we break it up into simpler problems. This is done in Lean using the `have` tactic.

`have`	`have hp : P,` creates a new goal with target `P` and adds `hp : P` as a hypothesis to the original goal.

The use of `have` that we have already seen is related to this one. When you use the tactic `have hq := f(hp),` Lean is internally replacing it with `have hq : Q, refine f(hp),`.

`have` is crucial for being able to use theorems from the library. To use these theorems you have to create terms that match the hypothesis *exactly*. Consider the following example. The type `n > 0` is not the same as `0 < n`. If you need a term of type `n > 0` and you only have `hn : 0 < n`, then you can use `have hn2 : n > 0, linarith,` and you will have constructed a term `hn2` of type `n > 0`.

We will need the following lemma later. Remember to save your proof. (Here's a hint if you need one.)

```
import tactic data.nat.prime
open nat

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

``have``

  ``have hp : P,`` creates a new goal with target ``P`` and
  adds ``hp : P`` as a hypothesis to the original goal.

You'll need the following theorem from the library:

nat.dvd_sub : n ≤ m → k ∣ m → k ∣ n → k ∣ m - n

   (Note that you don't need to provide n m k as inputs to dvd_sub
   Lean can infer these from the rest of the expression.
   More on this tomorrow.)
```

(continues on next page)

(continued from previous page)

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

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

theorem dvd_sub_one {p a : ℕ} : (p ∣ a) → (p ∣ a + 1) → (p ∣ 1) :=
begin
  sorry,
end
```

4.4 Infinitely many primes

We'll now prove that there are infinitely many primes. The strategy is to show that there is a prime greater than `n`, for every natural number `n`. We will choose this prime to be smallest non-trivial factor of `n! + 1`. We'll need the following definitions and theorems from the library.

Primes

- `m ∣ n := ∃ k : ℕ, m = n * k`
- `m.prime := 2 ≤ p ∧ (∀ (m : ℕ), m ∣ p → m = 1 ∨ m = p)`
- `nat.prime.not_dvd_one : (prime p) → ¬ p ∣ 1`

Factorials

- `factorial n` is defined to be n!
- `factorial_pos : ∀ (n : ℕ), 0 < factorial n`
- `dvd_factorial : 0 < m → m ≤ n → m ∣ factorial n`

Smallest factor

- `n.min_fac` is defined to be the smallest non-trivial factor of `n`
- `min_fac_prime : n ≠ 1 → n.min_fac.prime`
- `min_fac_pos : ∀ (n : ℕ), 0 < n.min_fac`
- `min_fac_dvd : ∀ (n : ℕ), n.min_fac ∣ n`

Check out data.nat.prime for more theorems about primes. The exercise below is very open-ended. You should take your time, check the goal window at every step, and sketch out the proof on paper whenever you get lost.

```
import tactic data.nat.prime
noncomputable theory
open_locale classical

open nat

theorem dvd_sub_one {p a : ℕ} : (p ∣ a) → (p ∣ a + 1) → (p ∣ 1) :=
begin
  sorry,
end

/-
dvd_sub_one : (p ∣ a) → (p ∣ a + 1) → (p ∣ 1)
```

(continues on next page)

(continued from previous page)

```
m ∣ n := ∃ k : ℕ, m = n * k
m.prime :=  2 ≤ p ∧ (∀ (m : ℕ), m ∣ p → m = 1 ∨ m = p)
nat.prime.not_dvd_one : (prime p) → ¬ p ∣ 1

factorial n is defined to be n!
factorial_pos : ∀ (n : ℕ), 0 < factorial n
dvd_factorial : 0 < m → m ≤ n → m ∣ factorial n

n.min_fac is defined to be the smallest non-trivial factor of n
min_fac_prime : n ≠ 1 → n.min_fac.prime
min_fac_pos : ∀ (n : ℕ), 0 < n.min_fac
min_fac_dvd : ∀ (n : ℕ), n.min_fac ∣ n
-/

theorem exists_infinite_primes (n : ℕ) : ∃ p, nat.prime p ∧ p ≥ n :=
begin
  set p := (n.factorial + 1).min_fac, -- Use `set` instead of `have` when you're just␣
↪making an abbreviation for a number.
  sorry,
end
```

4.5 Final remarks

It would be great if there was a one-to-one correspondence between "hand-written proofs" and proofs in Lean. But that is far from the case. When we write proofs we leave out a lot of details without even realizing it and expect the reader to be intelligent enough to fill them in. This is both a bug and feature. On the one hand this makes proofs readable. On the other hand too many "obviously true" arguments make proofs undecipherable and often wrong.

Unlike human readers, computers are pretty dumb (as of writing these notes). They can only do what you tell them to do and you cannot expect them to "fill in the details". But it is humanly impossible to teach a computer every single trivial fact about, say the natural numbers. The Lean math library contains a lot of trivial theorems but this collection is far from comprehensive. So theorem proving is Lean often involves the following steps:

- Scan the library to see which definitions and theorems might be useful.
- Choose the right hypotheses and wording for your theorem to match the theorems in the library. (Sadly, changing the wording slightly might end up making the proof infinitely harder to prove.)
- Break the theorem into small lemmas so that you can use the simplifiers more frequently.

As time goes on, we hope that theorem proving AIs can do more and more of this work and eventually eliminate the difference between human proofs and machine proofs.

CHAPTER

FIVE

SQRT 2 IS IRRATIONAL

Today we will teach Lean that $\sqrt{2}$ is irrational. Let us start by reviewing some concepts we encountered yesterday.

5.1 Implicit arguments

Consider the following theorem which says that the smallest non-trivial factor of a natural number greater than 1 is a prime number.

```
min_fac_prime : n ≠ 1 → n.min_fac.prime
```

It needs only one argument, namely a term of type `n ≠ 1`. But we have not told Lean what `n` is! That's because if we pass a term, say `hp : 2 ≠ 1` to `min_fac_prime` then from `hp` Lean can infer that `n = 2`. `n` is called an *implicit* argument. An argument is made implicit by using curly brackets `{` and `}` instead of the usual `(` and `)` while defining the theorem.

```
theorem min_fac_prime {n : ℕ} (hne1 : n ≠ 1) : n.min_fac.prime := ...
```

Sometimes the notation is ambiguous and Lean is unable to infer the implicit arguments. In such a case, you can force all the arguments to become explicit by putting an `@` symbol in from on the theorem. For example,

```
@min_fac_prime : (n : ℕ) → n ≠ 1 → n.min_fac.prime
```

Use this sparingly as this makes the proof very hard to read and debug.

5.2 The two haves

We have seen two slightly different variants of the `have` tactic.

```
have hq := ...
have hq : ...
```

In the first case, we are defining `hq` to be the term on the right hand side. In the second case, we are saying that we do not know what the term `hq` is but we know it's type.

Let's consider the example of `min_fac_prime` again. Suppose we want to conclude that the smallest factor of 10 is a prime. We will need a term of type `10.min_fac.prime`. If this is the target, we can use `apply min_fac_prime,`. If not, we need a proof of `10 ≠ 1` to provide as input to `min_fac_prime`. For this we'll use

```
have ten_ne_zero : 10 ≠ 1,
```

which will open up a goal with target `10 ≠ 1`. If on the other hand, you have another hypothesis, say `f : P → (10 ≠ 1)` and a term `hp : P`, then

```
have ten_ne_zero := f(hp)
```

will immediately create a term of type `10 ≠ 1`. More generally, remember that

1. "`:=`" stands for definition. `x := ...` means that `x` is defined to be the right hand side.
2. "`:`" is a way of specifying type. `x : ...` means that the type of `x` is the right hand side.
3. "`=`" is only ever used in propositions and has nothing to do with terms or types.

5.3 Sqrt(2) is irrational

We will show that there do not exist positive natural numbers `m` and `n` such that

```
2 * m ^ 2 = n ^ 2  -- (*)
```

The crux of the proof is very easy. You simply have to start with the assumption that `m` and `n` are coprime *without any loss of generality* and derive a contradiction. But proving that *without a loss of generality* is a valid argument requires quite a bit of effort. This proof is broken down into several parts. The first two parts prove `(*)` assuming that `m` and `n` are coprime. The rest of the parts prove the *without loss of generality* part.

For this problem you'll need the following definitions.

- `m.gcd n : ℕ` is the gcd of `m` and `n`.
- `m.coprime n` is defined to be the proposition `m.gcd n = 1`.

The descriptions of the library theorems that you'll be needing are included as comments. Have fun!

5.3.1 Lemmas for proving (*) assuming m and n are coprime.

```
/-
nat.prime.dvd_of_dvd_pow : ∀ {p m n : ℕ}, p.prime → p ∣ m ^ n → p ∣ m

Challenge mode: start with nat.even_or_odd instead
-/
lemma two_dvd_of_two_dvd_sq {k : ℕ} (hk : 2 ∣ k^2) :
  2 ∣ k :=
begin
  sorry,
end

lemma division_lemma_n {m n : ℕ}
  (hmn : 2 * m ^ 2 = n ^ 2)
: 2 ∣ n :=
begin
  sorry,
end

lemma div_2 {m n : ℕ} (hnm : 2 * m = 2 * n) : (m = n) :=
begin
  linarith,
end
```

(continues on next page)

(continued from previous page)

```
lemma division_lemma_m {m n : ℕ}
  (hmn : 2 * m ^ 2 = n ^ 2)
: 2 ∣ m :=
begin
  sorry,
end
```

5.3.2 Prove (*) assuming m and n are coprime.

```
/-
theorem nat.not_coprime_of_dvd_of_dvd  : 1 < d → d ∣ m → d ∣ n → ¬m.coprime n
-/

theorem sqrt2_irrational' :
  ¬ ∃ (m n : ℕ),
  2 * m^2 = n^2 ∧
  m.coprime n
:=
begin
  rintro ⟨m, n, hmn, h_cop⟩,
  -- these brackets let you combine ``rintro`` with (several iterations of) ``cases``
  -- try using ``rintro h`` and then ``rcases h with ⟨m, n, hmn, h_cop⟩,`` instead
  -- you get the brackets by typing ``\langle`` and ``\rangle``
  sorry,
end
```

5.3.3 Lemmas for proving (*) assuming m ≠ 0

```
/-
pow_pos : ∀ {a : ℕ}, 0 < a → ∀ (n : ℕ), 0 < a ^ n
-/
lemma ge_zero_sq_ge_zero {n : ℕ} (hne : 0 < n) : (0 < n^2)
:=
begin
  sorry,
end

/-
nat.mul_left_inj : ∀ {a b c : ℕ}, 0 < a → (b * a = c * a ↔ b = c)
-/
lemma cancellation_lemma {k m n : ℕ}
(hk_pos : 0 < k^2)
(hmn : 2 * (m * k) ^ 2 = (n * k) ^ 2)
: 2 * m ^ 2 = n ^ 2
:=
begin
  sorry,
end
```

5.3.4 Prove (*) assuming m ≠ 0

```
/-
gcd_pos_of_pos_left : ∀ {m : ℕ} (n : ℕ), 0 < m → 0 < m.gcd n
gcd_pos_of_pos_right : ∀ (m : ℕ) {n : ℕ}, 0 < n → 0 < m.gcd n
exists_coprime : ∀ {m n : ℕ}, 0 < m.gcd n → (∃ (m' n' : ℕ), m'.coprime n' ∧ m = m' *␣
↪m.gcd n ∧ n = n' * m.gcd n)
nat.pos_of_ne_zero : ∀ {n : ℕ}, n ≠ 0 → 0 < n

-/
theorem wlog_coprime :
  (∃ (m n : ℕ),
  2 * m^2 = n^2 ∧
  m ≠ 0 )
  → (∃ (m' n' : ℕ),
    2 * m'^2 = n'^2 ∧
    m'.coprime n' )
:=
begin
  rintro ⟨m, n, hmn, hm0⟩,
  set k := m.gcd n with hk,
  -- this abbreviation reduces clutter
  -- ``set`` is similar to ``have``
  -- you can replace all the ``m.gcd n`` with ``k`` using ``rw ←hk,`` if needed
  sorry,
end

theorem sqrt2_irrational'' :
  ¬ ∃ (m n : ℕ),
  2 * m^2 = n^2 ∧
  m ≠ 0
:=
begin
  sorry,
end
```

CHAPTER

SIX

BITS & PIECES

6.1 Namespaces

Lean provides us with the ability to group definitions into nested, hierarchical *namespaces*:

```
namespace mcsp
  def tau := "TAU on T-F from 2-4"
  #eval tau
end mcsp

def tau := "no TAU on S"
#eval tau
#eval mcsp.tau

open mcsp

#eval tau -- error
#eval mcsp.tau
```

When we declare that we are working in the namespace `mcsp`, every identifier we declare has a full name with prefix "`mcsp`". Within the namespace, we can refer to identifiers by their shorter names, but once we end the namespace, we have to use the longer names.

The `open` command brings the shorter names into the current context. Often, when we import a theory file, we will want to open one or more of the namespaces it contains, to have access to the short identifiers. Further if `x` is a term of type `nat` and `f` is a term defined in namespace `nat` then `nat.f x` can be shortened to `x.f`. Note that $\mathbb{N}$ is just another notation for `nat`.

6.2 Coercions

In type theory every term has a type and two terms of different types cannot be equal to each other. This makes it impossible to write statements like `|m|^2 = m^2` where `m : ℤ` and `|m| : ℕ` is the absolute value of `m`. But in math, we do want this statement to be true! The round about way to deal with this is through *coercions*. Lean will coerce the above equality to live entirely in integers as, `↑|m|^2 = m^2`. This is done using an injective function $\mathbb{N} \to \mathbb{Z}$.

Sometimes it is possible (and necessary) to get rid of the coercions. For example, say we start out with `↑|m|^2 = m^2` and eventually reduce it to `↑|m|^2 = ↑1`. The tactic for getting rid of coercions is `norm_cast` which will reduce the above expression to `|m|^2 = 1`.

`norm_cast`	`norm_cast`, tries to clear out coercions. `norm_cast at hp`, tries to clear out coercions at the hypothesis `hp`.

```
import tactic data.nat.basic data.int.basic
noncomputable theory
open_locale classical

theorem sqrt2_irrational_nat :
  ¬ ∃ (m n : ℕ),
  2 * m^2 = n^2 ∧
  m ≠ 0
:=
begin
  sorry,
end

-- Assume the above theorem

lemma num_2 : (2 : ℚ).num = 2 :=
begin
  sorry,
end

lemma denom_2 : (2 : ℚ).denom = 1 :=
begin
  sorry,
end

/-
q.denom = denominator of q (valued in ℕ)
q.num = numerator of q (valued in ℤ)

for integer m,
m.nat_abs = absolute value of m (valued in ℕ)

int.nat_abs_mul_self' : ∀ (a : ℤ), ↑(a.nat_abs) * ↑(a.nat_abs) = a * a
int.coe_nat_inj : ∀ {m n : ℕ}, ↑m = ↑n → m = n

rat.mul_self_denom : ∀ (q : ℚ), (q * q).denom = q.denom * q.denom
rat.mul_self_num : ∀ (q : ℚ), (q * q).num = q.num * q.num
rat.denom_ne_zero : ∀ (q : ℚ), q.denom ≠ 0

-/

theorem sqrt2_irrational :
¬ (∃ q : ℚ, 2 = q * q)
:=
begin
  rintro ⟨q, h⟩,
  have clear_denom := rat.eq_iff_mul_eq_mul.mp h,
  sorry,
end
```

6.3 Type classes

Type classes are used to construct complex mathematical structures. Any family of types can be marked as a type class. We can then declare particular elements of a type class to be instances. You can think of a type class as "template" for constructing particular instances.

Consider the example of groups. A group is defined a type class with the following attributes.

```
structure group : Type u → Type u
fields:
group.mul : Π {α : Type u} [c : group α], α → α → α
group.mul_assoc : ∀ {α : Type u} [c : group α] (a b c_1 : α), a * b * c_1 = a * (b *␣
↪c_1)
group.one : Π {α : Type u} [c : group α], α
group.one_mul : ∀ {α : Type u} [c : group α] (a : α), 1 * a = a
group.mul_one : ∀ {α : Type u} [c : group α] (a : α), a * 1 = a
group.inv : Π {α : Type u} [c : group α], α → α
group.mul_left_inv : ∀ {α : Type u} [c : group α] (a : α), a⁻¹ * a = 1
```

If you look at the source code you'll see that the `class group` is built gradually by extending multiple classes.

```
class has_one      (α : Type u) := (one : α)
-- a group has an identity element

class has_mul      (α : Type u) := (mul : α → α → α)
-- a group has multiplication

class has_inv      (α : Type u) := (inv : α → α)
-- a group has an inverse function

class semigroup (G : Type u) extends has_mul G :=
(mul_assoc : ∀ a b c : G, a * b * c = a * (b * c))
-- the multiplication is associative

class monoid (M : Type u) extends semigroup M, has_one M :=
(one_mul : ∀ a : M, 1 * a = a) (mul_one : ∀ a : M, a * 1 = a)
-- multiplication by one is trivial

class group (α : Type u) extends monoid α, has_inv α :=
(mul_left_inv : ∀ a : α, a⁻¹ * a = 1)
-- multiplication is associative
```

To define an arbitrary group `G` we first create it as a type `G : Type` and then make it an instance of `group` using `[group G]`. You can also prove that existing types are instances of `group` using the `instance` keyword. Type classes allow us to prove theorems in vast generalities. For example, any theorem about groups can immediately be applied to integers once we show that integers are an instance of `group`. If you look at data.int.basic you'll see that first fifty lines of code prove that $\mathbb{Z}$ is an instance of several type classes.

```
import group_theory.order_of_element
import tactic

#print classes
#print instances inhabited

class cyclic_group (G : Type*) extends group G :=
(has_generator:  ∃ g : G, ∀ x : G, ∃ n : ℤ, x = g^n)
```

(continues on next page)

(continued from previous page)

```
/-
zpow_add : ∀ {G : Type u_1} [group G] (a : G) (m n : ℤ), a ^ (m + n) = a ^ m * a ^ n
add_comm : ∀ {G : Type u_1} [add_comm_semigroup G] (a b : G), a + b = b + a
-/

lemma mul_comm_of_cyclic
  {G : Type*}
  [hc: cyclic_group G]
  (g : G)
: ∀ a b : G, a * b = b * a :=
begin
  have has_generator := hc.has_generator,
  sorry,
end
```

6.4 Recursion and Induction

Lots of things in Lean are defined using recursion and proved using induction. While this extends beyond just the natural numbers, let's try some familiar examples using the natural numbers and the familiar principle of induction.

First let's see how to make a recursive definition. I'll define afunction called `sum_first : ℕ → ℕ` so that `sum_first n` is the sum of the first `n` natural numbers.

```
import data.nat.basic
import tactic

def sum_first :
  ℕ → ℕ -- the type of the function you want to define recursively
| 0 := 0 -- the definition at 0
| (n + 1) := sum_first n + (n + 1) -- the definition at (n + 1), which can use the␣
↪definition at n
```

Now let's prove the famous closed formula for `sum_first n`, using induction. To do this, we'll want the following two tactics:

Now let's try the proof. Remember that `rw` can be useful for unfolding definitions.

```
import data.nat.basic
import tactic

def sum_first :
  ℕ → ℕ -- the type of the function you want to define recursively
| 0 := 0 -- the definition at 0
| (n + 1) := sum_first n + (n + 1) -- the definition at (n + 1), which can use the␣
↪definition at n

/-- nat.succ_eq_add_one : ∀ (n : ℕ), n.succ = n + 1 -/
theorem sum_first_formula : ∀ (n : ℕ), 2 * sum_first n = (n + 1) * n :=
begin
  sorry,
end
```

If you want more practice proving things about natural numbers, including plenty of induction, try the Natural Number Game.

CHAPTER

SEVEN

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
∣	`\mid`	divisibility[1]
ℕ	`\nat`	type of natural numbers
ℤ	`\int`	type of integers
∘	`\circ`	composition of functions
≠	`\ne`	not equal to
⟨⟩	`\langle` and `\rangle`	used to build complicated types out of simple types

[1] Be very careful! The symbol for divisibility is not the | symbol on your keyboard. Lean will through a cryptic error if you use it.

CHAPTER

EIGHT

GLOSSARY OF TACTICS AND LEMMAS

Here's a summary of all the tactics and some of the lemmas we will introduce in this class, as well as some other common ones you may encounter.

8.1 Implications in Lean

`refine`	If `P` is the target of the current goal and `hp` is a term of type `P`, then `refine hp,` will close the goal. Mathematically, this saying "this is what we were required to prove". If you can't fully close a goal, but want to work somewhat from the end, you can use _ to fill in the missing pieces. For instance, if the target of the current goal is `Q` and `f` is a term of type `P → Q`, then `refine f _,` changes the target to `P`. If you can fully close a goal, you can also type `exact hp,`, which does pretty much the same thing.
`rintro`	If the target of the current goal is a function `P → Q`, then `rintro 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 (introduce) an arbitrary element of `P`. If you want to use this repeatedly, you can type `rintro h1 h2` instead of `rintro h1,` and then `rintro h2,`. If you want to use this to introduce a variable of a more complicated type that you would then apply `cases` to, you can try something like `rintro ⟨x1, x2, x3⟩,` where ⟨⟩ are typed with `\langle` and `\rangle`.
`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`. You can also create subgoals with `have hp : P,` which will create a separate goal to prove `P`. Once you have closed this goal, you'll have the hypothesis `hp : P` at your disposal.
`set`	`set` is used to create intermediate variables or abbreviations. It's pretty similar to `have`, with one important difference. If you type `have x : X := y,`, Lean remembers that `x : X`, but does not remember that `x = y`. Meanwhile, `set` remembers, so if you type `set x : X := y with hx,` you also get `hx : x = y`, which you can use to rewrite.
`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`". This is similar to using `refine _,`.

8.2 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`. You can also accomplish this with `refine ⟨_, _⟩`.
`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`.
`rcases`	`rcases` is a more general form of `cases`. Needs the symbols ⟨⟩, which are typed with `\langle` and `\rangle`. For an example, say you have `h : ∃ (m n : ℕ), 2 * m ^ 2 = n ^ 2 ∧ 0 < m`. Then you can type `rcases h with ⟨m, n, hmn, hme0⟩,` to break `h` into its 4 component parts.

8.3 Negations and Proof by Contradiction

`false.elim`	Not a tactic, but a lemma. If `P : Prop`, then `false.elim : false → P` lets you prove `P` from a contradiction.
`exfalso`	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.
`em`	Not a tactic, but a lemma. If `P : Prop`, then `em P : P ∨ ¬ P` lets you use the law of the excluded middle on `P`.
`by_cases`	If `P : Prop`, then `by_cases hp : P,` creates two goals, the first with a hypothesis `hp: P` and second with a hypothesis `hp: ¬ P`. This lets you use the law of the excluded middle, combining `em` with `cases`.
`by_contra`	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. This is essentially a combination of `rintro` with `false.elim`.
`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.

8.4 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`.
`rintro`	If the target of the current goal is `∀ x : X, P x`, then `rintro 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`. See also `rcases` to avoid using `cases` repeatedly.
`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. You can also use `refine ⟨_, _⟩`, and then you get two goals, one with target `X`, and the other is the fact `P y`, where `y` is the witness you entered for `X`. If you already have the witness `y`, you may type `refine ⟨y, _⟩,`.

8.5 Proving "trivial" statements

`refl`	`refl,` proves things that are literally true by definition.
`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_nf`	`ring_nf,` is Lean's symbolic manipulator. If the target has a proof that involves *only* algebraic operations, then `ring_nf,` will close the goal. If `hp : P` is an assumption then `ring_nf 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. If you really want to use `simp,` but it doesn't close the goal, try `squeeze_simp,`, and click the instructions given in the goal window.

8.6 Equality

`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).
	You can also use this to unfold definitions, for instance if `f : X → Y`, then `rw surjective,` will change the goal `surjective f` to `∀ (b : Y), ∃ (a : X), f a = b`, so you can see what you're trying to prove. For this purpose, you could also use the tactic `unfold`, as in `unfold surjective,`.

8.7 Induction

`induction`	If `n : ℕ` is a natural number variable, `P : ℕ → Prop` is a property of natural numbers, and you want to prove `P n` using induction, then `induction n using k ih,` will create two goals. One has target `P 0`, this is the base case. The other has target `P (k.succ)`, where `k.succ = k + 1`. (You can rewrite away the `.succ` with `nat.succ_eq_add_one`.) You're also provided an induction hypothesis, `ih : P k`.
`refl`	`refl,` proves things that are literally true by definition. Often this will handle your base case.