Lean at MC2020

Release 0.1

Apurva Nakade
Jalex Stark

Jul 14, 2020

CONTENTS

1 Introduction 1

2 Logic in Lean - Part 1 3

3 Logic in Lean - Part 2 11

4 Infinitely Many Primes 19

5 Sqrt 2 is irrational 25

6 Bits & Pieces 29

7 Pretty Symbols in Lean 33

8 Glossary of tactics 35

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, 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 only two things Lean can do is *create* terms and *check* their types. By iterating these two operations one 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
  exact rfl,
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 Goal` window. There are several exercises in these notes to be done this way.

These notes are designed for a 5-day Lean crash course 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 covers 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. You're highly encouraged to join one or both of these channels.

1.3 Acknowledgments.

These notes are developed by Apurva Nakade and Jalex Stark with a lot of help from Mathcamp campers and Mathcamp staff Joanna and Maya (thanks!). 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.
    5. Jalex Stark'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 mission, should you choose to accept it, is to understand the philosophy of type theory (in Lean). Don't try to memorize anything, that will happen automatically. Instead, try to ~~realize that there is no spoon~~ 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 is by bookmarking the Lean window in your web browser.

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 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. (Lean ignores multiple whitespaces, tabs, and new lines. You could theoretically write the entire code in a single line but then we can never be friends.)

- `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.

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

``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. Here are some hints if you get stuck.

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

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

(continues on next page)

(continued from previous page)

```
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

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.

`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.
`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 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.

footnotes

[1] Except under staff supervision.

CHAPTER

THREE

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.

Finally, we know enough Lean tactics 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.

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

``by_cases``

  If ``P`` is a proposition, then ``by_cases P,`` creates two goals,
    the first with a hypothesis ``hp: P`` and
    second with a hypothesis ``hp: ¬ P``.

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).

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

``by_cases``

  If ``P`` is a proposition, then ``by_cases P,`` creates two goals,
    the first with a hypothesis ``hp: P`` and
    second with a hypothesis ``hp: ¬ P``.

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

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

(continues on next page)

(continued from previous page)

```
-- 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 Proving "trivial" statements

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.

Similarly, the mathlib library also contains the following definition of `prime`.

```
def nat.prime (p : ℕ) : Prop
:=
  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 might come at the cost of your sanity. Fortunately, there are tactics in Lean for proving trivial proofs such as these.

`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.

```
import tactic data.nat.prime

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

``norm_num``

  Useful for arithmetic.

``ring``

  Useful for basic algebra.

``linarith``

  Useful for inequalities.

``simp``

  Complex simplifier. Use only to close goals.

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

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

example : 1 > 0 :=
begin
  sorry,
end

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

(continues on next page)

(continued from previous page)

```
example : 101 ∣ 2020 :=
begin
  sorry,
end


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

-- you will need the definition
-- a ∣ b := (∃ k : ℕ, a = b * k)
example (m a b : ℕ) :  m + a ∣ m^2 + (a + b) * m + a * b :=
begin
  sorry,
end

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

-- if none of the simplifiers work, try doing ``contrapose!``
-- sometimes the simplifiers need a little help
example (n : ℕ) : 0 < n ↔ n ≠ 0 :=
begin
  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**.

4.1 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. The standard trick is to make the LHS (almost) equal or to the RHS and then use one of the simplifiers (`norm_num`, `ring`, `linarith`, or `simp`) to close the goal. *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` followed by tab.

```
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 :=
```

(continues on next page)

(continued from previous page)

```
begin
  ring,
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
  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
```

4.1.1 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

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

``unfold``

  If it gets hard to keep track of the definition of ``surjective``,
  you can use ``unfold surjective,`` or ``unfold surjective at h,``
  to get rid of it.

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

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

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

/-
```

(continues on next page)

(continued from previous page)

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

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

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

4.2 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, exact 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.) **Warning:** If you need to type the divisibility symbol, type `\mid`. This is **not** the vertical line on your keyboard.

```
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
```

4.3 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)`
- `prime.not_dvd_one : (prime p) → ¬ p ∣ 1`

Factorials

- `n.fact := n! --n factorial`
- `fact_pos : ∀ (n : ℕ), 0 < n.fact`
- `dvd_fact : 0 < m → m ≤ n → m ∣ n.fact`

Smallest factor

- `n.min_fac` := 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)
prime.not_dvd_one : (prime p) → ¬ p ∣ 1

n.fact := n! (n factorial)
fact_pos : ∀ (n : ℕ), 0 < n.fact
dvd_fact : 0 < m → m ≤ n → m ∣ n.fact

n.min_fac := 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.fact + 1).min_fac,
  sorry,
end
```

4.4 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.

The hope is that one day we won't have to do this and a theorem proving AI will 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 non-zero 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.

```
/-
prime.dvd_of_dvd_pow : ∀ {p m n : ℕ}, p.prime → p ∣ m ^ n → p ∣ m
-/
lemma two_dvd_of_two_dvd_sq {k : ℕ}
  (hk : 2 ∣ k ^ 2)
: 2 ∣ k :=
begin
  apply prime.dvd_of_dvd_pow,
  sorry,
end

-- to switch the target from ``P = Q`` to ``Q = P``,
-- use the tactic ``symmetry,``
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
```

(continues on next page)

(continued from previous page)

```
  linarith,
end

lemma division_lemma_m {m n : ℕ}
  (hmn : 2 * m ^ 2 = n ^ 2)
: 2 ∣ m :=
begin
  apply two_dvd_of_two_dvd_sq,
  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
  by_contradiction,
  rcases a with ⟨m, n, hmn, h_cop⟩,
  -- rcases is a way of doing cases iteratively
  -- you get the brackets by typing ``\langle`` and ``\rangle``
  sorry,
end
```

5.3.3 Lemmas for proving (*) assuming m ≠ 0

```
lemma ne_zero_ge_zero {n : ℕ}
  (hne : n ≠ 0)
: (0 < n)
:=
begin
  contrapose! hne,
  sorry,
end

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

lemma cancellation_lemma {k m n : ℕ}
(hk_pos : 0 < k^2)
```

(continues on next page)

(continued from previous page)

```
(hmn : 2 * (m * k) ^ 2 = (n * k) ^ 2)
: 2 * m ^ 2 = n ^ 2
:=
begin
  apply (nat.mul_left_inj hk_pos).mp,
  ring at *,
  exact hmn,
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)
-/
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
  intro key,
  rcases key with ⟨m, n, hmn, hme0⟩,
  set k := m.gcd n with hk,
  -- might be useful to declutter
  -- 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 vmcsp
  def tau := "TAU on M-Th from 1-3"
  #eval tau
end vmcsp

def tau := "no TAU on F"
#eval tau
#eval vmcsp.tau

open vmcsp

#eval tau -- error
#eval vmcsp.tau
```

When we declare that we are working in the namespace `vmscp`, every identifier we declare has a full name with prefix "`vmscp`". 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 * m) = (n * n) ∧
  m ≠ 0
:=
begin
  sorry,
end

-- Assume the above theorem

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

lemma div_2 : (2 : ℚ).denom = 1 :=
begin
  ring,
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)

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
int.nat_abs_mul_self' : ∀ (a : ℤ), ↑(a.nat_abs) * ↑(a.nat_abs) = a * a
rat.denom_ne_zero : ∀ (q : ℚ), q.denom ≠ 0
-/

/-
Use ``squeeze_simp at hp,`` to commute products with coercions.
See the goal window!
-/

theorem sqrt2_irrational :
¬ (∃ q : ℚ, 2 = q * q)
:=
begin
  by_contradiction,
  cases a with q key,
  have clear_denom := rat.eq_iff_mul_eq_mul.mp key,
  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)

```
/-
gpow_add : ∀ {G : Type u_1} (a : G) (m n : ℤ), a ^ (m + n) = a ^ m * a ^ n
add_comm : ∀ {G : Type u_1} (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
```

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

[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

8.1 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`".

8.2 Proof by 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.
`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.

8.3 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`.

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`.
`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.

8.5 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.

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).