MTH1001 in Lean

Release 1.00

Gihan Marasingha

Nov 02, 2020

CONTENTS

CHAPTER

ONE

THE THEORY OF PROPOSITIONAL LOGIC

1.1 Propositions and propositional variables

From an informal perspective, propositions are statements that can be assigned a truth value.

The formal perspective, which is more useful when you want to prove a theorem, propositions are simply expressions that can be constructed recursively by applying connectives to other propositions.

As with any recursive definition, we must begin with a base set of propositions. These are called *propositional variables*. Technically, we have an infinite list of propositional variables, labelled $P_1, P_2, \ldots, P_n, \ldots$. For practical purposes, we often refer to propositional variables using different letters of the alphabet whether uppercase $P, Q, R, \ldots$ or lower case $p, q, r, \ldots$.

The five connectives of propositional logic are shown in Table 1.1, listed in decreasing order of precedence. Negation is a unary connective, which means that it applies only to one proposition. The remaining connectives are binary, applying to two propositions.

Table 1.1: Connectives of propositional logic and their order of precedence

Connective	Name
$\neg$	Negation
$\wedge$	Conjunction, And
$\vee$	Disjunction, Or
$\rightarrow$	Implication
$\leftrightarrow$	If and only if

Definition 1 (Proposition)**.** *A proposition is a propositional variable or one of* $(\neg\alpha)$, $(\alpha\wedge\beta)$, $(\alpha\vee\beta)$, $(\alpha \rightarrow \beta)$, $(\alpha \leftrightarrow \beta)$, *where* α *and* β *are propositions.*

For example, if our propositional variables are P, Q, and R, then all the following are propositions: P, $(\neg P)$, $(\neg(P\wedge Q))$, $((P \vee Q) \leftrightarrow (\neg R))$.

To simplify writing propositions, we may always remove the outermost parentheses and remove other parentheses according to the order of precedence. That is, if $\oplus$ and $\otimes$ are (generic) connectives where $\oplus$ has a higher order of precedence than $\otimes$, if α, β, and γ are propositions, then $((\alpha \oplus \beta) \otimes \gamma)$ can be replaced with $\alpha \oplus \beta \otimes \gamma$. Likewise, $(\alpha \otimes (\beta \oplus \gamma))$ can be replaced with $\alpha \otimes \beta \oplus \gamma$. This rule can be applied recursively.

Thus $((P \wedge Q) \rightarrow (P \vee (Q \wedge (\neg R))))$ can be more simply written as $P \wedge Q \rightarrow P \vee Q \wedge \neg R$.

The proposition, $((P \wedge Q) \vee R)$ can be written as $(P \wedge Q) \vee R$, but we cannot remove the inner parentheses as the connective $\vee$ has a lower order of precendence than $\wedge$.

Two propositions α and β are considered to be equal if and only if they are written *identically*, with the exception of parenthesis dropping as described in the previous paragraph.

Thus $P \wedge Q$ is *not* equal to $Q \wedge P$. However, $((P \wedge Q) \vee R)$ is equal to $(P \wedge Q) \vee R$ and to $P \wedge Q \vee R$.

In Section 2.1, we discuss how to represent propositions in Lean.

1.2 Derivations and rules of inference

Propositional logic is concerned with making derivations, based on *premises*, using *rules of inference*.

Each premise takes the form $h : \alpha$, where h is any symbol (usually a lowercase letter, with or without a subscript) and α is a proposition.

Intuitively, the judgment $h : \alpha$ is to be read 'h is a proof of α'. We will give a more rigorous interpretation of this judgment in Section 3.1.

Here is an example of a rule of inference. This rule is called left conjunction elimination.

> Given $h : P \wedge Q$, we have a proof of P.

In essence, a rule of inference is a black box that takes certain inputs (the premises) and produces an output. In this case, the output is a proof of P. Here, and in every statement of a rule of interfence, the names of propositions that appear are to be treated as generic. Thus P and Q can be replaced with any propositions.

As an example application of this rule of inference, suppose we have a premise $h : (P \to Q) \wedge R$. Left conjunction elimination, applied to h, produces a proof of $P \to Q$.

Rules of inference can be applied in sequence. Suppose we have a premise $k_1 : (Q \wedge P) \wedge R$. Applling left conjunction elimination to k_1 gives a proof of $Q \wedge P$. Let us denote by k_2 the proof of $Q \wedge P$ that results from the application of left conjunction elimination to k_1. By applying left conjunction elimination to the hypothesis k_2, we have a proof of Q.

We use the word *hypothesis* to denote either a premise or a result derived by a rule of inference during the course of a proof. At any stage in a proof, the entire set of hypotheses developed or introduced up to that point, together with any variables, is called the *context*. In the above proof, the context initially consists of thepremise k_1 and the varialbles P, Q, and R. After the first application of left conjunction elimination, the context will also include the hypotheses k_2.

1.3 Conjunction

1.3.1 Conjunction elimination

There are two conjunction elimination rules, left and right.
Rule 2 (Conjunction elimination)**.**

- *(*Left and elimination*) given $h : P \wedge Q$, we have a proof of P.*
- *(*Right and elimination*) given $h : P \wedge Q$, we have a proof of Q.*

As an example, given that P and Q are propositions, we will deduce Q from the premise $h : (P \wedge Q) \wedge R$.

> We have $h_2 : P \wedge Q$ by left conjunction elimination on h. The result follows by right conjunction elimination on h_2.

1.3.2 Conjunction elimination in Lean

This is expressed in Lean as follows.

```
variables p q r : Prop

example (h : (p ∧ q) ∧ r ) : q :=
begin
  have h₂ : p ∧ q, from h.left,
  exact h₂.right
end
```

The `exact` tactic is a 'finishing command' that closes the goal with the supplied proof term. Here, `h.left` is the proof term that results from applying left conjunction elimination to `h`. The `have` tactic introduces a new goal, in this case `h₂ : p ∧ q`. It should be followed by a tactic that closes the goal. Here, `from h.left` is a synonym for `exact h.left`.

In the above proof, `h₂.right` is the proof term that results from applying conjunction elimination to `h₂`. As `h₂` is a proof of `p ∧ q`, we have that `h₂.right` is a proof of `q`.

Tactic-style Lean proofs are designed to be worked with *interactively*, not to be read. If you are reading this online, click *try it* above to open the code snippet in a browser window. Note that the first time you press *try it!*, a copy of Lean will be downloaded to your browser. This may take a few minutes.

At each point in the proof, Lean displays the *goal* (that which you are trying to prove) and the *context* in a separate pane of your window.

At the start of the proof above, Lean will display the following, indicating that the context consists of three propositional variables, `p`, ``q`, and `r`, together with the premise `h : (p ∧ q) ∧ r`. The goal (indicated with the *turnstile* symbol ⊢) is that of proving `q`.

```
p q r : Prop,
h : (p ∧ q) ∧ r
⊢ q
```

If you place your cursor after the line with the `have` statement, the context changes to the following, in which `h₂ : p ∧ q` has been added.

```
p q r : Prop,
h : (p ∧ q) ∧ r,
h₂ : p ∧ q
⊢ q
```

To make the proof more readable, you can use the `show` tactic. This tactic announces what remains to be proved. Below, we follow `show q,` with `from h₂.right`, which is a synonym for `exact h₂.right`.

```
example (h : (p ∧ q) ∧ r ) : q :=
begin
  have h₂ : p ∧ q, from h.left,
  show q, from h₂.right,
end
```

The above is a Lean representation of the following mathematical proof.

> We have $h_2 : P \wedge Q$ by left conjunction elimination on h. We show q by right conjunction elimination on h_2.

Rather than introducing an intermediate hypothesis `h₂`, the proof can be carried out in one line.

```
example (h : (p ∧ q) ∧ r) : q :=
begin
  show q, from (h.left).right
end
```

If a Lean proof can be accomplished with one tactic, one need not use a `begin` … `end` block but can instead write the tactic after `by`, as below.

```
example (h : (p ∧ q) ∧ r) : q :=
by exact (h.left).right
```

This can be written mathematically as follows.

> The result follows by right conjunction elimination applied to the result of left conjunction elimination applied to h.

Alternatively, we can use the `cases` tactic, which performs left and right and elimination simultaneously.

```
example (h : (p ∧ q) ∧ r) : q :=
begin
  cases h with h₂ h₃, -- We have `h₂ : p` and `h₃ : q` by left and right conjunction␣
↪elimination on `h`.
  exact h₂.right      -- The result follows by right conjunction elimination on `h₂`.
end
```

A mathematical statement of this proof would be:

This can be written mathematically as follows.

> We have $h_2 : p \wedge q$ and $h_3 : r$ by left and right conjunction elimination, respectively, on h. The result follows by right conjunction elimination on h_2.

1.3.3 Conjunction introduction

The rule of conjunction introduction can be expressed in two forms, forward and backward.
Rule 3 (Conjunction introduction, forward)**.** *Given $h_1 : P$ and $h_2 : Q$, we have a proof of $P \wedge Q$.*
Rule 4 (Conjunction introduction, backward)**.** *To prove $P \wedge Q$, it suffices to prove P and Q.*
Example 5 (Commutativity of conjunction (I))**.** *Let P and Q be propositions. Given $h : P \wedge Q$, we have a proof of $Q \wedge P$.*

We'll give both a forward and a backward proof.

Forward Proof. We have $h_2 : P$ and $h_3 : Q$ by left and right conjunction elimination on h. The result follows by conjunction introduction on h_3 and h_2. □

Lean uses `and.intro` to represent forward conjunction introduction.

```
example (h : p ∧ q) : q ∧ p :=
begin
  have h₂ : p, from h.left,
  have h₃ : q, from h.right,
  exact and.intro h₃ h₂,
end
```

Backward Proof. By conjunction introduction, it suffices to prove 1. Q and 2. P.

1. We show Q from right conjunction introduction on h.

2. We show P from left conjunction elimination on h

□

Lean uses `split` to represent backward conjunction introduction. As used below, the `split` tactic replaces the goal of proving `q ∧ p` with two goals 1. to prove `q` and 2. to prove `p`.

```
example (h : p ∧ q) : q ∧ p :=
begin
  split,                -- By and introduction, it suffices to prove both `q` and `p`.
  show q, from h.right, -- We show `q` by right and elimination on `h`.
  show p, from h.left,  -- We show `p` by left and elimination on `h`.
end
```

If a rule of inference introduces multiple goals, it is good practice (though not required) to enclose the proof of each new goal in braces. For good measure, I throw in a `show` at the start of the proof to demonstrate that `show` need not be followed by a tactic that immediately closes the goal (such as `from` or `exact`). Here, the scope of `show` is the entire proof.

```
example (h : p ∧ q) : q ∧ p :=
begin
  show q ∧ p, split,
  { show q, from h.right, },
  { show p, from h.left, },
end
```

Associativity of conjunction, in parts

Example 6. *Let P, Q, and R be propositions. Given $h : (P \wedge Q) \wedge R$, we have a proof of $P \wedge (Q \wedge R)$.*

Here's a forward proof.

Proof.
- We have $h_2 : P \wedge Q$ and $h_3 : R$ by left and right conjunction elimination on h.
- We have $h_4 : P$ and $h_5 : Q$ by left and right conjunction elimination on h_2.
- We have $h_6 : Q \wedge R$ by conjunction introduction on h_5 and h_3.
- The result follows by conjunction introduction on h_4 and h_6.

□

The same proof can be represented in Lean. In the last line below (just to show that we can), we we the anonymous constructor notation to express conjunction introduction.

```
example (h : (p ∧ q) ∧ r) : p ∧ (q ∧ r) :=
begin
  cases h with h₂ h₃,                    -- We have `h₂ : (p ∧ q)`, `h₃ : r` by left &
↪ right and elim. on `h`.
  cases h₂ with h₄ h₅,                   -- We have `h₄ : p` and `h₅ : q` by left &␣
↪right and elim. on `h₂`.
  have h₆ : q ∧ r, from and.intro h₅ h₃,  -- We have `h₆ : q ∧ r` by and introduction␣
↪on `h₅` and `h₃`.
  show p ∧ (q ∧ r), from ⟨h₄, h₆⟩,        -- We show `p ∧ (q ∧ r)` by and␣
↪introduction on `h₄` and `h₆`.
end
```

Here's a proof of Example 6 that combines forward and backward reasoning. reasoning. The reiteration tactic is discussed more fully in Section 1.4.

Proof.

- We have $h_2 : P \wedge Q$ and $h_3 : R$ by left and right conjunction elimination on h.
- By conjunction introduction, it suffices to prove 1. P and 2. $Q \wedge R$.
  1. We show P from left conjunction elimination on h_2.
  2. We show $Q \wedge R$. By conjunction introduction, it suffices to show 1. Q and 2. R.
     a. We show Q from right conjunction elimination on h_2.
     b. We show R by reiteration on h_3.

□

This proof can also be represented in Lean.

```
example (h : (p ∧ q) ∧ r) : p ∧ (q ∧ r) :=
begin
  cases h with h₂ h₃,           -- We have `h₂ : p ∧ q` and `h₃ : r` by left and right␣
↪conjunction elimination on `h`.
  split,                         -- By conjunction introduction, it suffices to prove␣
↪`p` and `q ∧ r`.
  { show p, from h₂.left, },     -- We show `p` by left and elimination on `h₂`.
  { show q ∧ r, split,           -- We show `q ∧ r`. By conjunction introduction, it␣
↪suffices to prove `q` and `r`.
    { show q, from h₂.right, }, -- We show `q` by right and elimination on `h₂`.
    { show r, from h₃ }, } ,     -- We show `r` by reiteration on `h₃`.
end
```

Of course, associativity also works in the other direction.

Example 7. *Let P, Q, and R be propositions. Given $h : P \wedge (Q \wedge R)$, we have a proof of $(P \wedge Q) \wedge R$.*

Here is an (incomplete) forward proof. Fill in each 'sorry' to complete the proof.

Proof.

- We have $h_2 : P$ and h_3 : sorry by sorry on h.
- We have h_4 : sorry and $h_5 : R$ by left and right conjunction elimination on h_3.
- We have $h_6 : P \wedge Q$ by sorry.
- The result follows by sorry.

□

Likewise, fill in each `sorry` to complete the forward Lean proof below.

```
example (h : p ∧ (q ∧ r)) : (p ∧ q) ∧ r :=
begin
  cases h with h₂ h₃,
  cases h₃ with h₄ h₅,
  have h₆ : p ∧ q, from sorry,
  sorry,
end
```

As an exercise, replace each sorry below to give a mixed forward and backward proof of Example 7.

Proof.

- We have $h_2 : P$ and h_3 : sorry by sorry on h.
- By conjunction introduction, it suffices to prove 1. sorry and 2. sorry.

1. sorry
2. sorry

□

Likewise, fill in each `sorry` to complete the forward and backward Lean proof below.

```
example (h : p ∧ (q ∧ r)) : (p ∧ q) ∧ r :=
begin
  cases h with h₂ h₃,
  split,
  { split,
    { sorry, },
    { sorry, }, },
  { sorry, },
end
```

1.4 Reiteration

Rule 8 (Reiteration)**.** *Given P, we have a proof of P.*

Reiteration is a slighly unusual rule. Though we can often avoid using reiteration, it is required in proving statements such as $P \to P$.

For the moment, we present a silly example in which we use reiteration, albeit needlessly.
Example 9. *Let P and Q be propositions. Given $h : P \wedge Q$, we have a proof of Q.*

Proof. We have $h_2 : Q$ by right conjunction elimination on h. The result follows by reiteration on h_2. □

Reiteration is represented in Lean via the `exact` (or `from`) tactic applied to an already-deduced proof term. The code below shows a Lean representation of the proof above.

```
example (h : p ∧ q) : q :=
begin
  have h₂ : q, from h.right, -- We have `h₂ : q` by right and elimination on `h`.
  exact h₂,                  -- The result follows by reiteration on `h₂`.
end
```

A more verbose mathematical proof concludes by reminding the reader of the goal. Below, for example, we write, 'We show Q by …' in place of 'The result follows by …'.

Proof. We have $h_2 : Q$ by right conjunction elimination on h. We show Q by reiteration on h_2. □

The Lean equivalent is the combination of the `show` and `from` tactics.

```
example (h : p ∧ q) : q :=
begin
  have h₂ : q, from h.right, -- We have `h₂ : q` by right and elimination on `h`.
  show q, from h₂,           -- We show `q` by reiteration on `h₂`.
end
```

1.5 Implication

The *conditional* connective $\to$ is read 'implies'. The proposition $P \to Q$ can be read either as 'P implies Q' or as 'if P, then Q'.

In a proposition of the form $P \to Q$, the proposition P is called the *antecedent* and Q is called the *consequent*. The proposition $P \to Q$ is called an *implication* or a *conditional*.

1.5.1 Implication elimination

Rule 10 (Implication elimination, forward)**.** *Given $h_1 : P \to Q$ and $h_2 : P$, we have a proof of Q.*
Rule 11 (Implication elimination, backward)**.** *Given $h_1 : P \to Q$, to prove Q, it suffices to prove P.*

Here's an example application of implication elimination.
Example 12. *Let P, Q, and R be propositions. Given $h_1 : P \to Q$, $h_2 : Q \to R$ and $h_3 : P$, we have a proof of R.*

We'll give two proofs of this. One using forward reasoning and one with backward reasoning.

Forward proof. By implication elimination on h_1 and h_3, we have $h_4 : Q$. We show R by implication elimination on h_2 and h_4. □

In Lean, the proof of `q` from `h₁ : p → q` and `h₂ : p` is simply denoted `h₁ h₂`. The Lean translation of the foward proof of Example 12 is given below.

```
example (h₁ : p → q) (h₂ : q → r) (h₃ : p) : r :=
begin
  have h₄ : q, from h₁ h₃, -- We have `h₄ : q`, by implication elimination on `h₁` and
↪`h₃`.
  show r, from h₂ h₄       -- We show `r` by implication elimination on `h₂` and `h₄`.
end
```

Another approach it to dispose of h_4 entirely. This is harder to read, but quicker to write.

Short forward proof. R follows by implication elimination on h_2 and the result of implication elimination on h_1 and h_3. □

```
example (h₁ : p → q) (h₂ : q → r) (h₃ : p) : r :=
begin
  show r, from h₂ (h₁ h₃)
end
```

The next proof uses one backward application and one forward application of implication elimination.

Backward proof. To prove R, it suffices, by implication elimination on h_2 to prove Q. We show Q by implication elimination on h_1 and h_3. □

In Lean, given `h₁ : p → q` and a goal to prove `q`, we transform the goal into one of proving `p` using `apply h₁`. We use this to translate the above backward proof of Example 12.

```
example (h₁ : p → q) (h₂ : q → r) (h₃ : p) : r :=
begin
  apply h₂,          -- By implication elimination on `h₂`, it suffices to prove `q`.
  show q, from h₁ h₃ -- We show `q` by implication elimination on `h₁` and `h₃`.
end
```

If desired, we could give an entirely backward proof, finishing with reiteration.

```
example (h₁ : p → q) (h₂ : q → r) (h₃ : p) : r :=
begin
  apply h₂, -- By implication elimination on `h₂`, it suffices to prove `q`.
  apply h₁, -- By implication elimination on `h₁`, it suffices to prove `p`.
  exact h₃, -- This follows by reiteration on `h₃`.
end
```

Here's an exercise in which the first line of the proof uses backward implication elimination. You'll also have to use conjunction introduction.

```
example (h₁ : p ∧ q → r) (h₂ : p) (h₃ : q) : r :=
begin
  apply h₁, -- By implication elimination on `h₁`, it suffices to prove `p ∧ q`.
  sorry
end
```

To really test your understanding of implication elimination, see if you can do the following exercise.

```
example (h₁ : d → a) (h₂ : f → b) (h₃ : e → c) (h₄ : e → a)
        (h₅ : d → e) (h₆ : b → e) (h₇ : c) (h₈ : f) : a :=
begin
  sorry
end
```

1.5.2 Implication introduction

Implication introduction is one of the most important rules of inference. It is the only rule, in propositional logic, that permits us to derive a goal on *no premises*. Due to this, impliciation introduction only has a backward form.
Rule 13 (Implication introduction)**.** *To prove $P \to Q$ is to assume $h : P$ and derive Q.*
Example 14. *Let P and Q be propositions. Then $Q \to (P \to Q)$.*

Proof.
- By implication introduction, it suffices to assume $h_1 : Q$ and deduce $P \to Q$.
- To show $P \to Q$, it suffices, by implication introduction, to assume $h_2 : P$ and derive Q.
- We show Q by reiteration on h_1.

□

In Lean, to prove `p → q`, we begin with the `intro` tactic to admit the assumption of the antecedent `p` into the context and to change the goal to that of proving `q`. For example, if the initial goal is to prove `p → q`, then `intro h` adds `h : p` into the context and changes the goal to that of proving `p`.

Here's a Lean proof of the theorem above.

```
example : q → (p → q) :=
begin
  intro h₁,         -- Assume `h₁ : q`. It suffices to prove `p → q`.
  intro h₂,         -- Assume `h₂ : p`. It suffices to prove `q`.
  show q, from h₁, -- We show `q` by reiteration on `h₁`.
end
```

To make explicit what is being assumed, you may instead use the `assume` tactic. Below, `assume h₁ : q` has an identical effect to `intro h₁` above. The only difference is that `assume` explicitly asserts that `h₁` is an assumption of `q`. This aids the human reader.

```
example : q → (p → q) :=
begin
  assume h₁ : q,    -- Assume `h₁ : q`. It suffices to prove `p → q`.
  assume h₂ : p,    -- Assume `h₂ : p`. It suffices to prove `q`.
  show q, from h₁,  -- We show `q` by reiteration on `h₁`.
end
```

The next result requires reiteration.

Theorem 15 (Reflexivity of implication)**.** *Let P be a proposition. Then $P \to P$.*

Proof. Assume $h : P$. By implication introduction, it suffices to prove P. The result follows by reiteration on h. □

```
theorem id : p → p :=
begin
  assume h : p,    -- Assume `h : p`. It suffices to prove `p`.
  show p, from h,  -- We show `p` by reiteration on `h`.
end
```

Theorem 16 (Commutativity of conjunction (II))**.** *Let P and Q be propositions. Then $P \wedge Q \to Q \wedge P$.*

Proof.

- By implication introduction, it suffices to assume $h : P \wedge Q$ and deduce $Q \wedge P$.
- To show $P \wedge Q$, it suffices, by conjunction introduction, to prove both 1. Q and 2. P.
  1. We show Q from right conjunction elimination on h.
  2. We show P from left conjunction elimination on h.

□

Here is the same proof in Lean. Note that we use `theorem` below instead of `example`. This produces a named result. Here, we call the result `and_of_and`. We'll discuss theorems further in Section 1.6.

```
theorem and_of_and : p ∧ q → q ∧ p :=
begin
  intro h,                    -- Assume `h : p ∧ q`. It suffices to prove `q ∧ p`.
  split,                      -- By `∧` intro., it suffices to prove both `q` and `p`.
  { show q, from h.right, },  -- We show `q` from right `∧` elimination on `h`.
  { show p, from h.left, },   -- We show `p` from left `∧` elimination on `h`.
end
```

The proof above uses backward conjunction introduction to prove `q ∧ p`. We can alternatively use forward conjunction introduction. Additionally, I use `assume` below instead of `intro` to improve readability.

```
theorem and_of_and : p ∧ q → q ∧ p :=
begin
  assume h : p ∧ q,                  -- Assume `h : p ∧ q`. It suffices to prove `q ∧␣
↪p`.
  have h₂ : q, from h.right,         -- We have `h₂ : q` by right conjunction␣
↪elimination on `h`.
  have h₃ : p, from h.left,          -- We have `h₃ : p` by left conjunction␣
↪elimination on `h`.
  show q ∧ p, from and.intro h₂ h₃, -- We show `q ∧ p` from conjunction introduction␣
↪on `h₂` and `h₃`.
end
```

If you've been paying close attention, you'll note that the proofs above virtually the same as our proofs of Example 5, the result that given $h : P \wedge Q$, we have a proof of $Q \wedge P$. The only difference is the addition of `intro h` as the first line of the Lean proof or 'Assume $h : P \wedge Q$, it suffices to prove $Q \wedge P$' as the first line of the mathematical proof.

In general, by enough applications of implication introduction, one can transform a result that involves premises into a result with no premises.

1.6 Theorems

1.6.1 Reusing results

One great thing about mathematics is that we don't constantly have to reinvent the wheel. Once a result is proved, we can use it to prove other results.

A *theorem* is a named result. In the previous section, we have a mathematical theorem we can refer to by number as Theorem 16 or by name as the *Commutativity of conjunction (II)* theorem. We called the corresponding Lean theorem `and_of_and`.

Think about how you might prove the following.
Example 17. *Let A and B be propositions. Then $(A \to B) \wedge (B \wedge A) \to (B \wedge A) \wedge (A \to B)$.*

We can get our hands dirty and leap straight into a proof as follows.

From the rules of inference. Assume $h : (A \to B) \wedge (B \wedge A)$. It suffices to prove $(B \wedge A) \wedge (A \to B)$. By conjunction introduction, it suffices to prove both 1. $B \wedge A$ and 2. $A \to B$.

1. This follows from right conjunction elimination on h.
2. This follows from left conjunction elimination on h.

□

But this proof is virtually identical to our proof of Theorem 16. Indeed, the *statement* of Example 17 is essentially that of Theorem 16, only with $A \to B$ in place of P and $B \wedge A$ in place of Q.

Indeed, one should think of Theorem 16 as stating that $P \wedge Q \to Q \wedge P$ *for all* propositions P and Q. We will develop the notion of 'for all' further in Section 3.

For the moment, we should think of the variables P and Q that appear in the statement of Theorem 16 as being *placeholders*, *inputs* or *parameters* that we can replace with any given terms, called *arguments*.

For example, taking $A \to B$ and $B \wedge A$ as arguments to the theorem gives a one-line proof of Example 17.

Using a previously proved theorem with explicit arguments. The result follows by Theorem 16 applied to $A \to B$ and $B \wedge A$. □

A Lean proof of the result above uses the theorem `and_of_and`, our Lean version of Theorem 16. We repeat (a more concise version of) this theorem below along with our proof of the new result.

```
theorem and_of_and (p q : Prop) : p ∧ q → q ∧ p :=
begin
  intro h,
  exact and.intro (h.right) (h.left)
end

example : (a → b) ∧ (b ∧ a) → (b ∧ a) ∧ (a → b) :=
by exact and_of_and (a → b) (b ∧ a)
```

1.6.2 Placeholders

Often, it isn't necessary to present the arguments explicitly. There are two alternatives. One is the use of the Lean placeholder, denoted by an underscore character, _. Whenever Lean encounters an _, it tries to *infer* an appropriate term. In the example below, Lean will infer that the first and second underscores should be replaced with `a → b` and `b ∧ a` respectively.

```
example : (a → b) ∧ (b ∧ a) → (b ∧ a) ∧ (a → b) :=
by exact and_of_and _ _
```

1.6.3 Implicit arguments

In situations like the above, it is evident that the arguments *must be* $A \to B$ and $B \wedge A$ because those are the arguments that match the form of the theorem with the form of the goal. It is typical in such situations not to state the arguments explicity in a mathematical proof but to leave them implicit.

Here's our shortened proof of Example 17.

Using a previously proved theorem with implicit arguments. The result follows by Theorem 16. □

To enable the use of implicit arguments in Lean, we need to use a special syntax when stating our theorem. In the statement of theorem `and_of_and_v2` below, we enclose the variable declarations in braces `{p q : Prop}` in contrast to the parentheses `(p q : Prop)` in the earlier version.

In application of the theorem, we write merely `exact and_of_and_v2` in place of our previous `exact and_of_and (a → b) (b ∧ a)`. In the new proof, the arguments `a → b` and `b ∧ a` to the theorem `and_of_and_v2` are implicit.

```
theorem and_of_and_v2 {p q : Prop} : p ∧ q → q ∧ p :=
begin
  assume h,
  exact and.intro (h.right) (h.left)
end

example : (a → b) ∧ (b ∧ a) → (b ∧ a) ∧ (a → b) :=
by exact and_of_and_v2
```

1.6.4 Using theorems with the `apply` tactic

Another way to use a theorem is via the `apply` tactic. In Section 1.5.1, we used `apply` with terms of type `p → q` when the goal is of type `q`. In that case, the apply tactic replaces the goal with one of proving `p`.

More generally, the `apply` tactic can be used on a term `h` whenever the type of the goal matches the 'conclusion' of the type of `h`. The `apply` tactic replaces the goal with as many subgoals as there are 'premises' of `h` and tries to close the goal by inference.

Let's see how `apply` works when used with the theorem `and_or_and` which states, for all propositions `p` and `q` that `p ∧ q → q ∧ p`. The goal is to prove `(b ∧ a) ∧ (a → b)`. The `apply` tactic matches the goal with the conclusion `p ∧ q → q ∧ p` and introduces new goals for `p` and `q`. Lean automatically infers that `p` should be replaced with `b ∧ a` and that `q` should be replaced with `a → b`, closing these new goals.

```
theorem and_of_and (p q : Prop) : p ∧ q → q ∧ p :=
begin
  intro h,
  exact and.intro (h.right) (h.left)
end

example (a b : Prop) : (a → b) ∧ (b ∧ a) → (b ∧ a) ∧ (a → b) :=
by apply and_of_and
```

In more interesting examples, Lean cannot automatically close the new goals introduced by `apply`.

We begin with a juicy theorem whose proof is a good exercise in the rules of inference for implication.
Theorem 18 (Transitivity of implication)**.** *Let P, Q, and R be propositions. Then* $(P \to Q) \to ((Q \to R) \to (P \to R))$.

Proof. Assume $h_1 : P \to Q$. By implication introduction, it suffices to prove $(Q \to R) \to (P \to R)$.

Assume $h_2 : Q \to R$. By implication introduction, it suffices to prove $P \to R$.

Asssume $h_3 : P$. By implication introduction, it suffices to prove R.

By implication eliminiation on h_2, it suffices to prove Q.

We show Q by implication elimination on h_1 and h_3. □

The proof has a direct translation into Lean.

```
theorem imp_trans1 : (p → q) → (q → r) → (p → r) :=
begin
  assume h₁ : p → q,  -- Assume `h₁ : p → q`. By implication introduction, it␣
↪suffices to prove `(q → r) → (p → r)`.
  assume h₂ : q → r,  -- Assume `h₂ : q → r`. By implication introduction, it␣
↪suffices to prove `p → r`.
  assume h₃ : p,      -- Assume `h₃ : p`. It suffices to prove `r`.
  apply h₂,           -- By implication elimination on `h₂`, it suffices to prove `q`.
  show q, from h₁ h₃, -- We show `q` by implication elimination on `h₁` and `h₃`.
end
```

There are several ways to think about Theorem 18.

First, it can be seen as a statement with propositional parameters P, Q, and R that can be replaced with arguments, say S, T, and U to give a proof of

$$(S \to T) \to ((T \to U) \to (S \to U)).$$

Second, we can develop this idea via the rules of inference for implication to 'peel off' the antecedent of the theorem and intepret it as stating that for given propositions S, T, and U *and* given $h_1 : S \to T$, we have a proof of $(T \to U) \to (S \to U)$.

Third, we can peel off the next implication. The theorem then states that given propositions S, T, and U, given $h_1 : S \to T$ and $h_2 : T \to U$, we have a proof of $S \to U$. There's even a fourth option that I leave for the reader to determine.

We use the third interpretation of the theorem in proving the result below.
Example 19. *Let S, T, and U be propositions. Given $k_1 : S \to T \wedge S$ and $k_2 : T \to U$, we have a proof of $S \to U$.*

Proof. Applying Theorem 18 (to propositions S, $T \wedge S$, and U), it suffices to prove $S \to T \wedge S$ and $T \wedge S \to U$.

1. We show $S \to T \wedge S$ by reiteration on k_1.
2. We show $T \wedge S \to U$ as follows. Assume $k_3 : T \wedge S$. By implication introduction, it suffices to prove U. We have $k_4 : T$ by left conjunction elimination on k_3. The result follows by implication elimination on k_2 and k_4.

□

This translates neatly into Lean via the `apply` tactic.

```
example (k₁ : s → t ∧ s) (k₂ : t → u) : s → u :=
begin
  apply imp_trans1,
  { show s → t ∧ s, from k₁, },
  { show t ∧ s → u,
    assume k₃ : t ∧ s,
    have k₄ : t, from k₃.left,
    show u, from k₂ k₄, },
end
```

1.6.5 Exercises

Prove the following result, a variant (with no premises) of our previous result Example 6.
Theorem 20 (Associativity of conjunction)**.** *Let P, Q, and R be propositions. Then* $(P \wedge Q) \wedge R \to P \wedge (Q \wedge R)$.

Here's a Lean template for the proof.

```
theorem and_assoc1 : (p ∧ q) ∧ r → p ∧ (q ∧ r) :=
begin
  sorry
end
```

As an exercise in applying theorems, prove the following, subject to the following restrictions. Your proof must begin with implication introduction. It must end with reiteration. All other steps must be applications of either the above theorem or our result on the commutativity of conjunction, Theorem 5
Theorem 21. *Let S, T, and U be propositions. Then* $S \wedge (T \wedge U) \to (S \wedge T) \wedge U$.

Here is a Lean template for the proof.

```
theorem and_assoc2 : s ∧ (t ∧ u) → (s ∧ t) ∧ u :=
begin
  intro h,
  sorry,
end
```

1.7 If and only if

The biconditional connective $\leftrightarrow$ is also called 'if and only if' or 'iff'. The proposition $P \leftrightarrow Q$ is read 'P if and only if Q' or 'P is equivalent to Q'.

There are strong parallels between the rules of inference for iff and those for conjunction.
Rule 22 (If and only if elimination)**.**

- *(*Left iff elimination*) given $h : P \leftrightarrow Q$, we have a proof of $P \to Q$.*
- *(*Right iff elimination*) given $h : P \leftrightarrow Q$, we have a proof of $Q \to P$.*

In Lean, if `h : p ↔ q`, then `h.1` (alternatively `iff.elim_left h`) is a proof of `p → q`. Likewise, `h.2` (alternatively `iff.elim_right h`) is a proof of `q → p`.
Rule 23 (Iff introduction, forward)**.** *Given $h_1 : P \to Q$ and $h_2 : Q \to P$, we have a proof of $P \leftrightarrow Q$.*

In Lean, given `h₁ : p → q` and `h₂ : q → p`, the term `iff.intro h₁ h₂` is a proof of `p ↔ q`. The same proof term can be denoted using the anonymous constructor notation as `⟨h₁, h₂⟩`. Recall that ⟨ and ⟩ are written as `\<` and `\>` respectively.

Rule 24 (Iff introduction, backward)**.** *To prove $P \leftrightarrow Q$, it suffices to prove $P \to Q$ and $Q \to P$.*

We'll use these rules of inference to prove our (almost) final form of commutativity of conjunction. The proof below uses Theorem 16, that if P and Q are propositions, then $P \wedge Q \to Q \wedge P$.

Theorem 25 (Commutativity of conjunction (III))**.** *Let R and S be propositions. Then $R \wedge S \leftrightarrow S \wedge R$.*

Proof. By iff introduction, it suffices to prove 1. $R \wedge S \to S \wedge R$ and 2. $S \wedge R \to R \wedge S$. We close both goals by Theorem 16. □

In Lean, using one proof to close more than one goal is denoted by the `;` tactic combinator, as used in the proof below.

```
theorem and_comm : r ∧ s ↔ s ∧ r :=
begin
  split;
  apply and_of_and,
end
```

1.7.1 Converse

Given a conditional $P \to Q$, its *converse* is the conditional $Q \to P$. The rules of inference for iff effectively assert that to prove a biconditional $P \leftrightarrow Q$ is to prove a conditional $P \to Q$ and its converse $Q \to P$.

1.7.2 Reflexivity, symmetry, transitivity of iff

Iff has some particularly nice properties.

- Reflexivity. For every proposition P, we have $P \leftrightarrow P$.
- Symmetry. For all propositions P and Q, given $h : P \leftrightarrow Q$, we have $Q \leftrightarrow P$.
- Transitivity. For all propositions P, Q, and R, given $h_1 : P \leftrightarrow Q$ and $h_2 : Q \leftrightarrow R$, we have $P \leftrightarrow R$.

Reflexivity. By iff introduction, it suffices to prove $P \to P$ and $P \to P$. Both these goals are closed by Theorem 15, the reflexivity of implication. □

```
example : p ↔ p :=
begin
  split,                    -- By iff introduction, it suffices to prove `p → p` and␣
↪`p → p`
  { show p → p, from id }, -- We show `p → p` from reflexivity of implication.
  { show p → p, from id }, -- We show `p → p` from reflexivity of implication.
end
```

As in our Lean proof of Theorem 25, we may employ the `;` tactic combinator to combine the proofs of the two subgoals that arise from the use of the `split` tactic.

```
example : p ↔ p :=
begin
  split;         -- By iff introduction, it suffices to prove `p → p` and `p → p`.
  { exact id }, -- We close both subgoals by reflexivity of implication.
end
```

The proof of symmetry of iff is almost identical to the proof of Exampe 5, the commutativity of conjunction.

Symmetry of iff. By iff introduction, it suffices to prove $Q \to P$ and $P \to Q$. We show $Q \to P$ by right iff elimination on h. We show $P \to Q$ by left iff elimination on h. □

The Lean proof is virtually identical that that of Exampe 5

```
example (h : p ↔ q) : q ↔ p :=
begin
  split,                      -- By iff introduction, it suffices to prove `q → p` and
↪`p → q`.
  { show q → p, from h.2 }, -- We show `q → p` by right iff elimination on `h`.
  { show p → q, from h.1 }, -- We show `p → q` by left iff elimination on `h`.
end
```

We now proof transitivity. That is, given $h_1 : P \leftrightarrow Q$ and $h_2 : Q \leftrightarrow R$, we have a proof of $P \leftrightarrow R$.

Transitivity of iff. By iff introduction, it suffices to prove 1. $P \to R$ and 2. $R \to P$.

1. We show $P \to R$. Applying the transtivity of implication (Theorem 18), it suffices to prove a. $P \to A$ and b. $A \to R$ (for some proposition A).
   a. We show $P \to Q$ by left iff eliminiation on h_1.
   b. We show $Q \to R$ by left iff elimination on h_2.
2. The proof of $R \to P$ is similar and is left to the reader.

□

```
example (h₁ : p ↔ q) (h₂ : q ↔ r) : p ↔ r :=
begin
  split,                              -- By iff intro., it suffices to prove `p → r`
↪and `r → p`.
  { show p → r, apply imp_trans1,    -- We show `p → r`. By transitivity of `→`, it
↪suffices to prove `p → ?` and `? → r`.
    { show p → q, from h₁.1, },      -- We show `p → q` by left iff elimination on
↪`h₁`.
    { show q → r, from h₂.1, }, },   -- We show `q → r` by left iff elimination on
↪`h₂`.
 { show r → p, sorry  },             -- The proof of `r → p` is left to the reader.
end
```

1.8 Rewriting

Whenever two propositions P and Q are judged to be equal, the proposition P can be replaced with P, wherever P appears. This process is called *rewriting*. Technically, equality is a notion of predicate logic rather than propositional logic.

1.8.1 Rewriting a goal

We'll use rewriting in proving the following result.
Example 26. *Let x, y, z be natural numbers. Then $x * (y + z) = x * z + y * x$.*

Our proof will call on the following intermediate results (which will be proved in due course).
Theorem 27 ((Left) distributivity of multiplication over addition)**.** *Let a, b, c be natural numbers. Then $a * (b + c) = a * b + a * c$.*
Theorem 28 (Commutativity of addition)**.** *Let a and b be natural numbers. Then $a + b = b + a$.*
Theorem 29 (Commutativity of multiplication)**.** *Let a and b be natural numbers. Then $a * b = b * a$.*

Returning to the example, we have to prove $x * (y + z) = x * z + y * x$. As a first step, we can rewrite this using the left distributivity of multiplication over addition (or, more simply, distributivity) applied to x, y and z. By application, I mean that the variables x, y and z take the roles of a, b and c, respectively in the distributive law.

The distributive law with these varaibles subsituted reads $x * (y + z) = x * y + x * z$. We rewrite the goal using the proposition. The left side of the goal is replaced with the right side of the preceding equation, changing the goal to one of proving $x * y + x * z = x * z + y * x$.

Proof. Rewriting using distributivity applied to x, y, and z, the goal is to prove $x * y + x * z = x * z + y * x$.

Rewriting using commutativity of addition applied to $x * y$ and $x * z$, the goal is to prove $x * z + x * y = x * z + y * x$.

Rewriting using commutativity of multiplication applied to x and y, the goal is to prove $x * z + y * x = x * z + y * x$.

This is trivially true (formally, it's true by reflexivity of $=$). □

With two of the above rewrites, it isn't stricly necessary to identify the variables being used. For example, in the initial goal, the distributivity law could only possibly apply to the variables x, y, and z as there is no expression of the form $a * (b + c)$ in the goal except for $x * (y + z)$.

In Lean, we use the `rw` tactic to denote rewriting. Given `h : P = Q`, the tactic `rw h` will replace the first occurrence (reading left-to-right) of `P` with `Q`. If the expression `h` depends on variables or other hypotheses, then Lean will look for the first expression in the goal that matches the shape of `h` and instantiate variables as necessary.

In the code below, `mul_add` is the theorem that for all natural numbers `a`, `b`, and `c`, we have `a * (b + c) = a * b + a * c`. Lean matches the left side of this equation with `x * (y + z)` after instantiating `a` as `x`, `b` as `y` and `c` as `z`.

Having performed that rewrite, the goal becomes `x * y + x * z = x * z + y * x`.

The `add_comm` states that for all natural numbers `a` and `b`, we have `a + b = b + a`. There are *two* subexpressions in the goal `x * y + x * z = x * z + y * x` that match with `add_comm`, namely `x * y + x * z` and `x * z + y * x`. However, Lean performs the rewrite on the first subexpression that matches. In this case, it's `x * y + x * z`.

Having performed `rw add_comm`, the goal becomes `x * z + x * y = x * z + y * x`.

We need to be precise is our last rewrite. This rewrite involves the theorem `mul_comm` which states that for all natural numbers `a` and `b`, we have `a * b = b * a`. The first subexpression of the goal to which this theorem applies is `x * z`. That is, the result of performing `rw mul_comm` would be identical to the result of performing `rw x z`, namely to change the goal to `z * x + x * y = x * z + y * x`.

However, rewriting `x * z` as `z * x` doesn't resolve the goal! Instead, we need to rewrite applying `mul_comm` to `x` and `y`.

This leaves, as a goal, `x * z + y * x = x * z + y * x`. Lean automatically closes this goal by the reflexivity of = (viz. the fact that `a = a`, for every `a`).

```
example : x * (y + z) = x * z + y * x :=
begin
  rw mul_add,
  rw add_comm,
  rw mul_comm x y,
end
```

Several rewrites can be combined by enclosing them in brackets, as below.

```
example : x * (y + z) = x * z + y * x :=
by rw [mul_add, add_comm, mul_comm x y]
```

1.8.2 Rewriting a hypothesis

Given a hypothesis $h : P = Q$, we can rewrite any other hypothesis k by replacing occurrences of P in k with Q.
Example 30. *Let x, y, z be natural numbers. Given $k : y * x = z$, we have $x * (y + z) = z + x * z$.*

Proof. Rewrite using commutativity of multiplication at k to give $k : x * y = z$.

Rewrite using distributivity. The goal is $x * y + x * z = z + x * z$.

Rewrite using k. The goal is $z + x * z = z + x * z$, which is trivially true. □

The Lean version of 'rewrite using h at k' is `rw h at k` as shown below.

```
example (k : y * x = z) : x * (y + z) = z + x * z :=
begin
  rw mul_comm at k,
  rw mul_add,
  rw k,
end
```

1.8.3 Rewriting in reverse

Given a hypothesis $h : P = Q$, we've seen that we can rewrite the goal (or another hypothesis) by replacing occurrences of P with Q. However, by symmetry of $=$, we can express h as $h : Q = P$ and then go on to replace occurrences of Q in the goal (or another hypothesis) by P.

We may refer to this process as 'rewriting using h in reverse'.

Recall that the distributive law states $a * (b + c) = a * b + a * c$. We can use this (in reverse) to rewrite the expression $x * y + x * z = (z + y) * x$ as $x * (y + z) = (z + y) * x$. This is the first step in the proof of the following result.
Example 31. *Let x, y and z be natural numbers. Then $x * y + x * z = (z + y) * x$.*

Proof. Rewriting using the distributive law (in reverse), the goal is $x * (y + z) = (z + y) * x$.

Rewriting using the commutativity of multiplication, the goal is $(y + z) * x = (z + y) * x$.

Rewriting using the commutativity of addition, the goal is $(z + y) * x = (z + y) * x$. This is trivially true. □

In Lean, we denote rewriting using `h` in reverse as `rw ←mul_add`, as in the example below.

```
example : x * y + x * z = (z + y) * x :=
begin
  rw ←mul_add,
  rw add_comm,
  rw mul_comm,
end
```

1.9 Propositional extensionality and rewriting

In advanced courses on mathematical logic (for the avoidance of doubt, this is not an advanced course), one typically proves theorems *about* systems of logic. One such theorem is that given propositions P and Q and given $h : P \leftrightarrow Q$, if the proposition Q is substituted for P wherever P appears in a theorem then result will still be a theorem.

We will not prove this theorem of meta propositional logic but we pause to note that the reflexivity, symmetry, and transitivity properties of $\leftrightarrow$, when taken together, suggest very strongly that propositions P and Q should be treated as equal if $P \leftrightarrow Q$.

To simplify matters, we will take treat this theorem as a rule, the principle of propositional extensionality.
Rule 32 (Propositional extensionality)**.** *Let P and Q be propositions. Given $h : P \leftrightarrow Q$, we have $P = Q$.*

1.9.1 Rewriting a goal

Given $h : P \leftrightarrow Q$, we can denote the use of propositional extensionality on the by writing, 'Rewriting using h, the goal is …', as in Section 1.8. We use this form of expression in the examples below.

In our first example, we use propositional extensionality to give a quick proof of transitivity of implication. Given $h_1 : P \leftrightarrow Q$ and $h_2 : Q \leftrightarrow R$ we have a proof of $P \leftrightarrow R$.

Proof. Rewriting using h_1, the goal is to prove prove $Q \leftrightarrow R$. This holds by reiteration on h_2. □

Compare this to the rather more involved proof given in Section 1.7.2.

In Lean, we use the `rw` tactic to rewrite the goal.

```
example (h₁ : p ↔ q) (h₂ : q ↔ r) : p ↔ r :=
begin
  rw h₁,     -- Rewriting using `h₁`, the goal is to prove `q ↔ r`.
  exact h₂, -- This holds by reiteration on `h₂`.
end
```

In the next example, we rewrite using De Morgan's law[1] $\neg(P \vee Q) \leftrightarrow \neg P \wedge \neg Q$ and our commtativity of conjunction result, Theorem 25.
Example 33. *Let P and Q be propositions. Then $\neg(P \vee Q) \leftrightarrow \neg Q \wedge \neg P$.*

Proof. Rewriting using De Morgan's law, the goal is to prove $\neg P \wedge \neg Q \leftrightarrow \neg Q \wedge \neg P$. This holds by applying commutativity of conjunction. □

In the Lean proof below, `not_or_distrib` is the name of the relvant De Morgan's law.

[1] We'll prove this later as Theorem 56. Another of De Morgan's laws is Theorem 62, which asserts $\neg(P \wedge Q) \leftrightarrow \neg P \vee \neg Q$.

```
example : ¬(p ∨ q) ↔ ¬q ∧ ¬p :=
begin
  rw not_or_distrib, -- Rewrite using De Morgan's law. The goal is `¬p ∧ ¬q ↔ ¬q ∧
↪¬p`.
  apply and_comm,    -- This holds by applying commutativity of conjunction.
end
```

1.9.2 Rewriting a hypothesis

We can use rewriting (i.e. propositional extensionality) on hypotheses just as we can on goals.
Example 34. *Let P and Q be propositions. Given $k : \neg(P \vee Q)$, we have a proof of $\neg Q \wedge \neg P$.*

Proof. Rewriting using De Morgan's law at k, we have $k : \neg P \wedge \neg Q$. Rewriting using commutativity of conjunction, the goal is $\neg P \wedge \neg Q$. The result follows by reiteration on k. □

```
example (k : ¬(p ∨ q)) : ¬q ∧ ¬p :=
begin
  rw not_or_distrib at k, -- Rewriting using De Morgan's law at `k`, we have `k : ¬p
↪∧ ¬q`.
  rw and_comm,            -- Rewriting using commutativity of conjunction, the goal
↪is `¬p ∧ ¬q`.
  exact k,                -- This holds by reiteration on `k`.
end
```

1.9.3 Rewriting in reverse

Given $h : P \leftrightarrow Q$ to rewrite using h in reverse is to replace occurrences of Q in the goal (or in another hypothesis) with P.

By rewriting in reverse, we give an alternative proof of Example 34. For this, we need the commutative law of disjunction (to be proved later). Namely, given propositions S and T, we have $S \vee T \leftrightarrow T \vee S$.

We are now in a position to prove Example 34.

Proof. Rewriting using De Morgan's law in reverse, the goal is $\neg(Q \vee P)$.

Rewriting using commutativity of disjunction, the goal is $\neg(P \vee Q)$.

We close the goal by reiteration on k. □

As in Section 1.8, we denote rewriting using `h` in reverse as `rw ←mul_add`, as in the example below.

```
example (h : ¬(p ∨ q)) : ¬q ∧ ¬p :=
begin
  rw ←not_or_distrib,
  rw or_comm,
  exact h,
end
```

1.10 Disjunction

1.10.1 Disjunction introduction

There are two disjunction introduction rules, left and right. Each comes in both a forward and a backward flavour.
Rule 35 (Disjunction introduction, forward)**.**

- *(*Left or introduction*) given $h : P$, we have a proof of $P \vee Q$.*
- *(*Right or introduction*) given $h : Q$, we have a proof of $P \vee Q$.*

Example 36. *Let S, T, and U be propositions. Given $h : T$, we have $S \vee (T \vee U)$.*

Forward. We have $h_2 : T \vee U$ by left or introduction on h. The result follows by right or introduction on h_2. □

In Lean, given `h : p`, the expression `or.inl h` is a proof term for `p ∨ q`. Given, `h : q`, the expression `or.inr h` is a proof term for `p ∨ q`.

```
example (h : t) : s ∨ (t ∨ u) :=
begin
  have h₂ : t ∨ u, from or.inl h,
  exact or.inr h₂,
end
```

Rule 37 (Disjunction introduction, backward)**.**

- *(*Left or introduction*) to prove $P \vee Q$, it suffices to prove P.*
- *(*Right or introduction*) to prove $P \vee Q$, it suffices to prove Q.*

We give a backward proof of Example 36.

Backward. We show $S \vee (T \vee U)$. By backward right or introduction, it suffices to prove $T \vee U$. By backward left introduction, it suffices to prove T. We show T by reiteration on h. □

The Lean tactic for backward left or introduction is `left`. That for backward right or introduction is `right`.

```
example (h : t) : s ∨ (t ∨ u) :=
begin
  right,           -- By right or introduction, it suffices to prove `t ∨ u`.
  left,            -- By left or introduction, it suffice to prove `t`.
  show t, from h,  -- We show `t` by reiteration on `h`.
end
```

1.10.2 Disjunction elimination

This is the most challenging rule so far. We start with an example.

Suppose we know of Peggy that she keeps rabbits or she grows strawberries. Further, we have 1. if Peggy keeps rabbits, then she needs straw as bedding material for the rabbits. We also have 2. if Peggy grows strawberries, then she needs straw to keep the strawberries off the ground and help keep them clean. We deduce that Peggy needs straw.

The deduction here is an application of disjunction elimination.
Rule 38 (Disjunction elimination, backward)**.** *Let P, Q, and R be propositions. Given $h : P \vee Q$, to prove R, it suffices 1. to show R on the assumption $h_1 : P$ and 2. to show R on the assumption $h_2 : Q$.*

Here is an archetypal example of backward disjunction elimination.
Example 39. *Let P, Q, and R be propositions. Given $h : P \vee Q$, $k_1 : P \to R$, and $k_2 : Q \to R$, we have a proof of R.*

Proof. By or elimination on h, it suffices 1. to show R on the assumption $h_1 : P$ and 2. to show R on the assumption $h_2 : Q$.

1. We show R by implication elimination on k_1 and h_1.
2. We show R by implication elimination on k_2 and h_2.

□

The Lean tactic used for decomposing a conjunction is `cases`. Suppose the current goal is `r`. Given `h : p ∨ q`, the expression `cases h with h₁ h₂` causes Lean to create two new goals. 1. to prove `r` with `h₁ : p` added to the context and 2. to prove `r` with `h₂ : q` added to the context.

```
example (h : p ∨ q) (k₁ : p → r) (k₂ : q → r) : r :=
begin
-- By or elim. on `h`, it suffices
  -- 1. to show `r` on the assumption `h₁ : p` and
  -- 2. to show `r` on the assumption `h₂ : q`.
  cases h with h₁ h₂,
  { show r, from k₁ h₁, }, -- We show `r` by implication elimination on `k₁` and `h₁`.
  { show r, from k₂ h₂, }, -- We show `r` by implication elimination on `k₂` and `h₂`.
end
```

The forward disjunction elimination rule is a restatement of the result just proved.
Rule 40 (Disjunction elimination, forward)**.** *Let P, Q, and R be propositions. Given $h : P \vee Q$, $k_1 : P \to R$ and $k_2 : Q \to R$, we have a proof of R.*

The parallel between the forward and backward versions of disjunction elimination are evident when one realises that to prove $k_1 : P \to R$, for example, is to assume $h_1 : P$ and to deduce R.

In Lean, given `h : p ∨ q`, `k₁ : p → r`, and `k₂ : q → r`, the expression `or.elim h k₁ k₂` is a proof term for `r`. This considerably shortens the proof of the previous result.

```
example (h : p ∨ q) (k₁ : p → r) (k₂ : q → r) : r :=
by exact or.elim h k₁ k₂
```

1.10.3 Commutativity and associativity of disjunction

As a more interesting example, we have a preliminary result on the commutativity of disjunction.
Theorem 41 (Commutativity of disjunction (I))**.** *Let S and T be propositions. Given $h : S \vee T$, we have $T \vee S$.*

We give a proof via backward or elimination and forward or introduction.

Proof. By or elimination on h, it suffices 1. to assume $h_1 : S$ and deduce $T \vee S$ and 2. to assume $h_2 : T$ and deduce $T \vee S$.

1. Assume $h_1 : S$. We show $T \vee S$ by right or introduction on h_1.
2. Assume $h_2 : T$. We show $T \vee S$ by left or introduction on h_2.

□

```
theorem or_of_or (h : s ∨ t) : t ∨ s :=
begin
  cases h with h₁ h₂,
  { exact or.inr h₁,},
  { exact or.inl h₂, },
end
```

Using the above result, we can prove a more symmetrical version of the commutativity result.
Theorem 42 (Commutativity of disjunction (II))**.** *Let P and Q be propositions. Then $P \vee Q \leftrightarrow Q \vee P$.*

Proof. By iff introduction, it suffices to prove 1. $P \vee Q \to Q \vee P$ and 2. $Q \vee P \to P \vee Q$. Both these goals can be closed by applying Theorem 41. □

```
theorem or_comm : p ∨ q ↔ q ∨ p :=
begin
  split;
  apply or_of_or
end
```

The associativity result is more involved. We give a partial Lean proof of the result. Fill in the `sorry` s below to complete the proof.

```
theorem or_assoc : (p ∨ q) ∨ r ↔ p ∨ (q ∨ r) :=
begin
  split,
  { intro h,
    cases h with h₁ h₂,
    { cases h₁ with m₁ m₂,
      { left, exact m₁, },
      { right, left, exact m₂, }, },
    { sorry, }, },
  { sorry, },
end
```

1.11 False and negation

1.11.1 Rules for false and negation

The symbol $\bot$, read 'arbitrary contradiction' or 'false' is a constant proposition. Its use is governed most fundamentally by the principle *ex falso sequitur quodlibet*, 'out of false, whatever you like follows', also called false elimination.

This principle is illustrated by such English-language phrases as, 'If Torquay United wins the FA Cup, then I'm a monkey's uncle'. The premise 'Torquay United wins the FA Cup' being considered a contradiction, I can derive anything from its assumption.
Rule 43 (False elimination (*ex falso*), forward)**.** *Given $h : \bot$, we have a proof of P.*
Rule 44 (False elimination (*ex falso*), backward)**.** *To prove P, it suffices to prove $\bot$.*
Example 45. *Let P and Q be propositions. We have $(P \to \bot) \to (P \to Q)$.*

Proof.
- Assume $h_1 : P \to bot$. It suffices to prove $P \to Q$ (by implication intro.).
- Assume $h_2 : P$. It suffice to prove Q (by implication intro.).
- By false elimination, it suffices to prove $\bot$.
- We show $\bot$ by implication elimination on h_1 and h_2.

□

`false` is the Lean equivalent of $\bot$. If `h` is a proof term for `false`, then `false.elim h` is a proof term for `p` (where `p` is the current goal).

```
example : (p → false) → (p → q) :=
begin
  assume h₁ : p → false,
  assume h₂ : p,
  show q, from false.elim (h₁ h₂)
end
```

Alternativly, the tactic `exfalso` can be used to change the current goal to one of proving `false`.

```
example : (p → false) → (p → q) :=
begin
  assume h₁ : p → false,
  assume h₂ : p,
  exfalso,
  show false, from h₁ h₂
end
```

We write $\neg P$ as a shorthand for $P \to \bot$. The expression $\neg P$ is read 'the negation of P' or 'not P'. With this notation, we can re-express the above example.

Theorem 46. *Let P and Q be propositions. Given $h_1 : \neg P$ and $h_2 : P$, we have a proof of Q.*

This follows on applying Example 45 with h_1 and h_2.

```
example (h₁ : ¬p) (h₂ : p) : q :=
begin
  exfalso,
  show false, from h₁ h₂
end
```

Alternatively, the powerful `contradiction` tactic searches the context for a contradition (viz. the appearance of a proposition and its neagation in the context) and uses it to close the goal.

```
example (h₁ : ¬p) (h₂ : p) : q :=
begin
  contradiction
end
```

In the special case where Q is $\bot$, this result is called *false introduction* and is often treated as a rule of inference, though it is really just implication elimination in disguise.

Theorem 47 (False introduction, forward)**.** *Given $h_1 : \neg P$ and $h_2 : P$, we have a proof of $\bot$.*

Theorem 48 (False introduction, backward)**.** *Given $h_1 : \neg P$, to prove $\bot$, it suffices to prove P.*

Another derived result is negation introduction, which is implication introduction in disguise.

Theorem 49 (Negation introduction)**.** *Let P be a proposition. To prove $\neg P$ is to assume $\bot$ and derive P.*

1.11.2 Applications of false and negation

Example 50. *Let P and Q be propositions. Given $h : \neg(P \vee Q)$, we have $\neg P$.*

Here's a proof using negation introduction and backward false introduction.

Proof. Assume $h_1 : P$. By negation introduction, it suffices to derive $\bot$. By false introduction on h, it suffices to prove $p \vee q$. This follows by left or introduction on h_1. □

In the Lean proof below, we use `apply` to prove invoke (backward) false introduction, just as we would when invoking implication elimination. This is because false introduction *is* implication elimination.

Likewise, we use `assume` for negation introduction, just as we do for implication introduction.

```
example (h : ¬(p ∨ q)) : ¬p :=
begin
  assume h₁ : p,    -- It suffices to prove `false`.
  apply h,          -- By false introduction on `h`, it suffices to prove `p ∨ q`.
  exact or.inl h₁, -- This follows by left or introduction on `h₁`.
end
```

The next example shows one can derive the implication $P \to Q$ given the hypothesis $\neg P \vee Q$. We'll later show the converse of this assertion.

Example 51 (One direction of material conditional)**.** *Let P and Q be propositions. Then $\neg P \vee Q \to (P \to Q)$.*

Proof. Assume $h_1 : \neg P \vee Q$. Assume $h_2 : P$. It suffices to prove Q. By or elimination on h_1, it suffices to 1. assume $h_3 : \neg P$ and derive Q and 2. assume $h_4 : Q$ and derive Q.

1. Assume $h_3 : \neg P$. By false elimination, it suffices to prove $\bot$. We show this by false introduction on h_3 and h_2.
2. Assume $h_4 : Q$. We show Q by reiteration on h_4.

□

In the Lean proof below we write `h₃ h₂` for a forward false introduction using `h₃` and `h₂`. This is because we are really doing implication elimination on `h₃` and `h₂`.

```
example : (¬p ∨ q) → (p → q) :=
begin
  intros h₁ h₂,                  -- Assume `h₁ : ¬p ∨ q`. Assume `h₂ : p`. It suffices␣
↪to prove `q`.
  -- By or elim. on `h₁`, it suffices to 1. assume `h₃ : ¬p` and derive `q` and 2.␣
↪assume `h₄ : q` and derive `q`
  cases h₁ with h₃ h₄,
  { exfalso,                     -- By false elimination, it suffices to prove `false`.
    show false, from h₃ h₂, },  -- We show false by false introduction on `h₃` and␣
↪`h₂`.
  { show q, from h₄, },          -- We show `q` by reiteration on `h₄`.
end
```

The next result is one half of a result that shows P is equivalent to $\neg\neg P$. We prove the other half of this result in Section 1.12.

Theorem 52 (One direction of double negation)**.** *Let P be a proposition. We have $P \to \neg\neg P$.*

Proof.

- Assume $h_1 : P$. It suffices to prove $\neg\neg P$.
- Assume $h_2 : \neg P$. By negation introduction, it suffices to prove $\bot$.
- We show $\bot$ by false introduction on h_2 and h_1.

□

```
theorem not_not_of_self : p → ¬¬p :=
begin
  intros h₁ h₂,           -- Assume `h₁ : p`. Assume `h₂ : ¬p`. It suffices to prove␣
↪`false`.
  show false, from h₂ h₁, -- We show `false` by false introduction on `h₂` and `h₁`.
end
```

The contrapositive of a conditional $P \to Q$ is the proposition $\neg Q \to \neg P$. In Section 1.12, we'll prove that a proposition is equivalent to its contrapositive. We'll prove one half of that assertion now.

Theorem 53 (One direction of contrapositive theorem, *modus tollens*)**.** *Let P and Q be propositions. Then* $(P \to Q) \to (\neg Q \to \neg P)$.

We give the Lean proof below with the corresponding mathematical proof in the comments. Note I could have replaced the first two lines with a single line, `intros h₁ h₂ h₃`.

```
theorem mt : (p → q) → (¬q → ¬p) :=
begin
  intros h₁ h₂, -- Assume `h₁ : p → q`, `h₂ : ¬q`. It suffices to prove `¬p`.
  intro h₃,     -- Assume `h₃ : p`. By negation introiduction, it suffices to prove␣
↪`false`.
  apply h₂,     -- By fale introduction on `h₂`, it suffices to prove `q`.
  exact h₁ h₃,  -- This follows by implication elimination on `h₁` and `h₃`.
end
```

De Morgan's laws state 1. $\neg(P \wedge Q) \leftrightarrow \neg P \vee \neg Q$ and 2. $\neg(P \vee Q) \leftrightarrow \neg P \wedge \neg Q$.

In total, there are four directions to prove. We give proofs of three directions in this section and the remaining direction in Section 1.12.

Theorem 54 (De Morgan, 'not or of not and not')**.** *Let P and Q be propositions. Then* $\neg P \wedge \neg Q \to \neg(P \vee Q)$.

Our first proof uses each rule of inference explicitly.

```
theorem not_or_of_not_and_not : ¬p ∧ ¬q →  ¬(p ∨ q) :=
begin
  assume h₁ : ¬p ∧ ¬q, -- Assume `h₁ : ¬p ∧ ¬q`. By implication introduction, it␣
↪suffices to prove `¬(p ∨ q)`.
  cases h₁ with h₃ h₄, -- We have `h₃ : ¬p` and `h₄ : ¬q` by left and right `∧` elim.␣
↪on `h₁`.
  assume h₅ : p ∨ q,   -- Assume `h₅ : p ∨ q`. By negation introduction, it suffices␣
↪to prove `false`.
  -- By or elimination on `h₅`, it suffices to 1. assume `h₆ : p` and derive `false`␣
↪and 2. assume `h₇ : q` and derive `false`.
  cases h₅ with h₆ h₇,
  { exact h₃ h₆, },    -- The goal in the first case is closed by false introduction␣
↪on `h₃` and `h₆`.
  { exact h₄ h₇, },    -- The goal in the second case is closed by false introduction␣
↪on `h₄` and `h₇`.

end
```

The second proof greatly simplifies this by using the `rintro` tactic, which introduces an assumption and decomposes it recursively. That is, it applies `cases` recursively to each introduced hypothesis.

Note the differences in how we use `rintro`. In our first application, we use `rintro ⟨h₃, h₄⟩` to introduce and decompose a hypothesis representing the conjunction `¬p ∧ ¬q`. The anonymous-constructor-like notation `⟨h₃, h₄⟩` appears here because the conjunction connective has one constructor, `and.intro`. The decomposition introduces `h₃ : ¬p` and `h₄ : ¬q` into the context.

We next use `rintro (h₆ | h₇)` to introduce and decompose a hypothesis representing the disjuction `p ∨ q`. As disjunction has two constructors, `or.inl` and `or.inr`, the decomposition introduces *two* new goals. This corresponds to or elimination.

```
theorem not_or_of_not_and_not : ¬p ∧ ¬q →  ¬(p ∨ q) :=
begin
  rintro ⟨h₃, h₄⟩,   -- By `→` intro and left and right `∧` elim, we have `h₃ : ¬p`␣
↪and `h₄ : ¬q`.
  -- By `→` intro and or elim., it suffices to 1. assume `h₆ : p` and derive `false`␣
↪and 2. assume `h₇ : q` and derive `false`.
```

(continues on next page)

(continued from previous page)

```
  rintro (h₆ | h₇),
  { exact h₃ h₆, }, -- The goal in the first case is closed by false introduction on␣
↪`h₃` and `h₆`.
  { exact h₄ h₇, }, -- The goal in the second case is closed by false introduction on␣
↪`h₄` and `h₇`.
end
```

Several successive `rintro` lines can be combined, albeit with some loss of readability.

```
theorem not_or_of_not_and_not : ¬p ∧ ¬q →  ¬(p ∨ q) :=
begin
  rintro ⟨h₃, h₄⟩ (h₆ | h₇),
  { exact h₃ h₆, }, -- The goal in the first case is closed by false introduction on␣
↪`h₃` and `h₆`.
  { exact h₄ h₇, }, -- The goal in the second case is closed by false introduction on␣
↪`h₄` and `h₇`.
end
```

Theorem 55 (De Morgan, 'not and not of not or'). *Let P and Q be propositions. Then $\neg(P \vee Q) \to \neg P \wedge \neg Q$.*

Here is a partial Lean proof. Fill in the `sorry` s. Hint: you've already seen the proof of the first subgoal earlier in this section.

```
theorem not_and_not_of_not_or : ¬(p ∨ q) → ¬p ∧ ¬q :=
begin
  intro h₁, -- Assume `h₁ : ¬(p ∨ q)`. It suffices to prove `¬p ∧ ¬q`.
  split,    -- By and introduction, it suffices to prove 1. `¬p` and 2. `¬q`.
  { sorry },
  { sorry },
end
```

By iff introduction on Theorem 54 and Theorem 55, we have our first complete De Morgan's law.
Theorem 56 (De Morgan, 'not or distrib'). *Let P and Q be propositions. Then $\neg(P \vee Q) \leftrightarrow \neg P \wedge \neg Q$.*

```
theorem not_or_distrib : ¬(p ∨ q) ↔ ¬p ∧ ¬ q :=
begin
  split,
  { exact not_and_not_of_not_or, },
  { exact not_or_of_not_and_not, },
end
```

Theorem 57 (De Morgan, 'not and of not or not'). *Let P and Q be propositions. Then $\neg P \vee \neg Q \to \neg(P \wedge Q)$.*

Here's part of the Lean proof. Fill in the `sorry`.

```
theorem not_and_of_not_or_not : ¬p ∨ ¬q → ¬(p ∧ q) :=
begin
  assume h₁ : ¬p ∨ ¬q, -- Assume `h₁ : ¬p ∨ ¬q`. It suffices to prove `¬(p ∧ q)`.
  rintro ⟨h₂, h₃⟩,      -- Introduce `p ∧ q`. By and elim., `h₂ : p` and `h₃ : q`. By␣
↪neg. intro, it suffices to prove `false`.
  -- By or elim. on `h₁`, it suffices to 1. assume `h₄ : ¬p` and prove `false` and 2.␣
↪assume `h₅ : ¬q` and prove `false`.
  cases h₁ with h₄ h₅,
  { exact h₄ h₂, }, -- This follows by negation introduction on `h₁` and `h₃`.
  { sorry },
end
```

1.12 Classical reasoning

1.12.1 Intuitionistic and constructive reasoning

You've now seen all the rules of inference for 'intuitionistic' propositional logic. Congratulations! Intuitionistic logic (more generally 'constructive reasoning') is a form of reasoning first investigated by L. E. J. Brouwer in the early 20th century.

From a constructive proof of a theorem, one can extract a method for constructing the described mathematical object. Almost all school mathematics is constructive.

For example, in school you prove that every quadratic equation $ax^2 + bx + c = 0$, where a, b, c are real (or complex) numbers with $a \neq 0$, has a (real or complex) solution

$$\frac{-b + \sqrt{a^2 - 4ac}}{2a}.$$

This is an explicit construction of a root of the quadratic. Can you conceive of a proof of existence that *doesn't* involve the construction of a root?

For a more elementary example, let P be a proposition. Can you prove $P \vee \neg P$? Using constructive reasoning you can't. In general, a constructive proof of $A \vee B$ requires that you prove A or that you prove B.

A mathematician using constructive reasoning will not be able to say, 'The Queen likes tea or the Queen doesn't like tea' unless they can prove either that the Queen likes tea or that the Queen doesn't like tea.

1.12.2 The law of the excluded middle

To resolve this issue, 'classical reasoning' takes $P \vee \neg P$ as an axiom[2], called the law of the excluded middle.
Axiom 58 (Law of the excluded middle)**.** *Let P be a proposition. Then we have a proof of $P \vee \neg P$.*

A mathematician using this law can deduce, 'The Queen likes tea or the Queen doesn't like tea', without any knowledge of the Queen's beverage preferences.

In Lean, the law of the excluded middle is a theorem called `classical.em`. Here's a very short example in which we prove `q ∨ ¬q`.

```
example : q ∨ ¬q := classical.em q
```

We'll give a proof of the result below, using the law of the excluded middle. The proof needs elements of predicate logic together with results on even and odd numbers. We will cover this in-depth in Section 3.
Example 59. *Let x be an integer. Then $x(x + 3)$ is even.*

Proof. By definition of 'even', our goal is to prove that there exists an integer m such that $x(x + 3) = 2m$.

Let P denote 'x is even'. By the law of the excluded middle, applied to P, we have $h_1 : P \vee \neg P$.

By or elimination on h_1, it suffices to 1. assume $h_2 : P$ and derive the goal and 2. assume $h_3 : \neg P$ and derive the goal.

1. Assume $h_2 : P$. That is, x is even. By definition, there exists an integer k such that $x = 2k$. Assume k is an integer for which $x = 2k$. Then

$$x(x + 3) = 2k(x + 3) = 2(k(x + 3)).$$

Take $m := k(x + 3)$. Then there is an integer m such that $x(x + 3) = 2m$.

[2] Technically,many systems that use classical reasoning, including Lean, assume the *axiom of choice*, a rather more sophisticated statement, and use it to deduce the law of the excluded middle.

2. Assume $h_3 : \neg P$. That is, x is not even. Thus (why?) x is odd. So there is an integer k such that $x = 2k+1$. Assume k is an integer for which $x = 2k + 1$. Then $x + 3 = (2k + 1) + 3 = 2(k + 2)$. We deduce

$$x(x + 3) = x \times (2(k + 2)) = 2(x(k + 2)).$$

Take $m := x(k + 2)$. Then there is an integer m such that $x(x + 3) = 2m$.

This completes the proof. □

1.12.3 Proof by cases

Example 59 is an instance of a method of proof called *proof by cases*. Below, give a proof of the general proof method.
Theorem 60 (Proof by cases)**.** *Let A and B be propositions. Given $h_1 : A \to B$ and $h_2 : \neg A \to B$, we have a proof of B.*

Proof. By the law of the excluded middle, applied to A, we have $h : A \vee \neg A$. By or elimination applied to h, to prove B, it suffices to 1. assume $k_1 : A$ and derive B and 2. assume $k_2 : \neg A$ and derive B.

1. Assume $k_1 : A$. We show B by implication elimination on h_1 and k_1.

2. Assume $k_2 : \neg A$. We show B by implication elimination on h_2 and k_2. □

As an application of proof by cases, we show the remaining direction of De Morgan's law.
Theorem 61 (De Morgan's law, 'not or not of not and')**.** *Let P and Q be propositions. Then $\neg(P \wedge Q) \to \neg P \vee \neg Q$.*

Proof. Assume $h_1 : \neg(P \wedge Q)$. It suffices to prove $\neg P \vee \neg Q$.

Via proof by cases, it suffices to 1. assume $h_2 : P$ and derive $\neg P \vee \neg Q$ and 2. assume $h_2 : \neg P$ and derive $\neg P \vee \neg Q$.

1. Assume $h_2 : P$. By right or introduction, it suffices to prove $\neg Q$. Assume $h_3 : Q$. By negation introduction, it suffices to prove $\bot$. By false introduction on h_1, it suffices to prove $P \wedge Q$. This follows by and introduction on h_1 and h_2.

2. We leave this part of the proof as an exercise for the reader.

□

This method of reasoning is represented in Lean with the `by_cases` tactic. Depending on your version of Lean, you may need to declare that propositions are 'decidable' using the command `local attribute [instance] classical.prop_decidable`. Fill in the `sorry` below.

```
local attribute [instance] classical.prop_decidable

theorem not_or_not_of_not_and : ¬(p ∧ q) → ¬p ∨ ¬q :=
begin
  assume h₁ : ¬(p ∧ q),     -- Assume `h₁ : ¬(p ∧ q)`. It suffices to prove `¬p ∨ ¬q`.
  -- Via proof by cases, it suffices to prove we can derive `¬p ∨ ¬q` 1. assuming `h₂
↪: p` and 2. assuming `h₂ : ¬p`.
  by_cases h₂ : p,
  { right,                  -- Assume `h₂ : p`. By right or introduction, it suffices
↪to prove `¬q`.
    assume h₃ : q,          -- Assume `h₃ : q`. By negation introduction, it suffices
↪to prove `false`.
    apply h₁,               -- By false introduction on `h₁`, it suffices to prove `p ∧
↪q`.
    exact ⟨h₂, h₃⟩, },      -- This follows by and introduction on `h₂` and `h₃`.
  { sorry },
end
```

By iff introduction on our previous Theorem 57 and Theorem 61, we deduce the final iff De Morgan's law.
Theorem 62 (De Morgan's law, 'not and distrib'). *Let P and Q be propositions. Then* $\neg(P \wedge Q) \leftrightarrow \neg P \vee \neg Q$.

```
theorem not_and_distrib : ¬(p ∧ q) ↔ ¬p ∨ ¬ q :=
begin
  split,
  { exact not_or_not_of_not_and, },
  { exact not_and_of_not_or_not, },
end
```

1.12.4 Double negation

Theorem 63 (Double negation). *Let P be a proposition. Then* $\neg\neg P \leftrightarrow P$.

Proof. By iff introduction, it suffices to prove 1. $\neg\neg P \to P$ and 2. $P \to \neg\neg P$.

1. Assume $h_1 : \neg\neg P$. It suffices to prove P. Via proof by cases, it suffices to prove P separately on the assumptions a. $h_2 : P$ and b. $h_2 : \neg P$.
   a. Assume $h_2 : P$. We show P by reiteration on h_2.
   b. Assume $h_2 : \neg P$. By false elimination, it suffices to prove $\bot$. We show $\bot$ by false introduction on h_1 and h_2.
2. This follows by Theorem 52.

□

```
theorem not_not : ¬¬p ↔ p :=
begin
  split,                        -- By iff intro., it suffices to prove 1. `p → ¬¬p`␣
↪and 2. `¬¬p → p`.
  { assume h₁ : ¬¬p,            -- Assume `h₁ : ¬¬p`. It suffices to prove `p`.
    by_cases h₂ : p,            -- We'll show `p` via proof by cases on `p`.
    { exact h₂,  },             -- Assume `h₂ : p`. The goal, `p` follows by reiteration␣
↪on `h₂`.
    { exfalso,                  -- By false elimination, it suffices to prove `false`.
      exact h₁ h₂, }, },        -- This follows by false introduction on `h₁` and `h₂`.
  { exact not_not_of_self, }, -- Goal 2. follows by a previously-proved theorem.
end
```

As an example application, we give another proof of :Theorem 61, this time using double negation instead of proof by cases. Recall that `not_or_distrib` is our constructive result `¬(a ∨ b) ↔ ¬a ∧ ¬b`.

```
example : ¬(p ∧ q) → ¬p ∨ ¬ q :=
begin
  assume h₁ : ¬(p ∧ q), -- Assume `h₁ : ¬(p ∧ q)`. It sfufice to prove `¬p ∨ ¬q`.
  have h₂ : ¬¬(¬p ∨ ¬q) ↔ ¬p ∨ ¬q, from not_not, -- By double negation, we have␣
↪`¬¬(¬p ∨ ¬q) ↔ ¬p ∨ ¬q`.
  rw ←h₂,                  -- Rewriting with `h₂`, the goal is to show `¬¬(¬p ∨ ¬q)`.
  rw not_or_distrib,       -- Rewriting with `not_or_distrib`, the goal is to show␣
↪`¬(¬¬p ∧ ¬¬q)`
  repeat { rw not_not, }, -- Repeatedly rewriting with double negation, the goal is␣
↪to show `¬(p ∧ q)`.
  show ¬(p ∧ q), from h₁, -- We show `¬(p ∧ q)` from `h₁`.
end
```

1.12.5 Proof by contradiction

Negation introduction is the (derived) rule that $\neg P$ is proved by assuming P and deriving $\bot$. Proof by contradiction (also called *reductio ad absurdum*) is a similar result, formed by subsituting Q for $\neg P$ and using double negation.
Theorem 64 (Proof by contradiction)**.** *Let Q be a proposition. To prove Q, it suffices to assume $\neg Q$ and derive $\bot$.*

This theorem is really just a restatement of one direction of the double negation result.

Proof. By left iff elimination on the double negation result, Theorem 63, we have $\neg\neg Q \to Q$.

By implication introduction, this means Q follows on the assumption $\neg\neg Q$ By definition of $\neg$, to prove $\neg\neg Q$ is to assume $\neg Q$ and derive $\bot$. □

In Lean, we invoke proof by contradiction using the `by_contradiction` tactic. We see this in use below in yet another proof of Theorem 61

```
example : ¬(p ∧ q) → ¬p ∨ ¬ q :=
begin
  assume h₁ : ¬(p ∧ q),       -- Assume `h₁ : ¬(p ∧ q)`. It sfufice to prove `¬p ∨ ¬q`.
  by_contradiction h₂,        -- For a contradiciton, assume `h₂ : ¬(¬p ∨ ¬q)`. It␣
↪suffices to prove `false`.
  rw not_or_distrib at h₂,    -- Rewriting using De Morgan's law `not_or_distrib`, `h₂`␣
↪is `¬¬p ∧ ¬¬q`.
  repeat {rw not_not at h₂}, -- Repeatedly using double negation, `h₂` is `p ∧ q`.
  show false, from h₁ h₂,     -- We show false by false introduction on `h₁` and `h₂`.
end
```

1.12.6 Proof by contrapositive

Recall that the contrapositive of a conditional $P \to Q$ is the proposition $\neg Q \to \neg P$. Using constructive reasoning, we previously proved Theorem 53, that $(P \to Q) \to (\neg Q \to \neg P)$.

By proving the converse of this result we will have, by iff introduction, $(\neg Q \to \neg P) \leftrightarrow (P \to Q)$.
Theorem 65 (Proof by contrapositive, 'not imp not')**.** *Let P and Q be propositions. Then $\neg Q \to \neg P \leftrightarrow P \to Q$.*

Proof. By iff introduction, it suffices to prove 1. $(\neg Q \to \neg P) \to (P \to Q)$. and 2. $(P \to Q) \to (\neg Q \to \neg P)$.

1. Assume $h_1 : \neg Q \to \neg P$. It suffices to prove $P \to Q$. Assume $h_2 : P$. It suffices to prove Q. For a contradiction, assume $h_3 : \neg Q$. It suffices to prove $\bot$. We have $h_4 : \neg P$ from implication elimination on h_1 and h_3. We show $\bot$ by false introduction on h_4 and h_2.
2. We show $(P \to Q) \to (\neg Q \to \neg P)$ by Theorem 53, modus tollens.

□

```
theorem not_imp_not : (¬q → ¬p) ↔ (p → q) :=
begin
  split, -- By iff intro., it suffices to prove 1. `¬q → ¬p → p → q` and 2. `p → q␣
↪→ ¬q → ¬p`.
  { intros h₁ h₂, -- Assume `h₁ : ¬q → ¬p`, `h₂ : p`. It suffices to prove `q`.
    by_contradiction h₃, -- For a contradiction, assume `h₃ : ¬q`. It suffices to␣
↪prove `false`.
    have h₄ : ¬p, from h₁ h₃, -- We have `h₄ : ¬p` by implication elimination on `h₁`␣
↪and `h₃`.
    show false, from h₄ h₂, -- We show `false` by false introduction on `h₄` and `h₂`.
  },
```

(continues on next page)

(continued from previous page)

```
  { exact mt }, -- We show `(p → q) → (¬q → ¬p)` by modus tollens.
end
```

As an example, we'll use proof by contrapositive to give another proof of Theorem 61.

```
example : ¬(p ∧ q) → ¬p ∨ ¬ q :=
begin
  rw ←not_imp_not, -- It suffices to prove the contrapositive, `¬(¬p ∨ ¬q) → ¬¬(p ∧␣
↪q)`.
  rw not_or_distrib, -- By De Morgan's Law `not_or_distrib`, it suffices to prove␣
↪`¬¬p ∧ ¬¬q → ¬¬(p ∧ q)`.
  repeat {rw not_not}, -- By repeated double negation, it suffices to prove `p ∧ q →␣
↪p ∧ q`.
  exact id, -- This follows by `id`, the reflexivity of implication.
end
```

CHAPTER

TWO

PROPOSITIONAL LOGIC LEAN SUMMARY

2.1 Propositional variables and special symbols

Before working with propositions in Lean, you must introduce their names as variables. For example, the following statement introduces propositional variables `p`, `q`, and `r`.

If you're reading this online, click *try it* below and a Lean window will open in which you can examine and work with the Lean code.

Note: the first time you open the window an entire copy of Lean will be downloaded to your browser. This may take a minute or more, depending or your internet speed. Alternatively, you can copy and paste the text into a CoCalc window.

```
variables p q r : Prop
-- This is a one-line comment. In has no effect in Lean.
```

In Table 2.1, we indicate how to type common symbols in Lean. After typing each sequence of characters, press space to ask Lean to convert the sequence into a symbol.

Table 2.1: Lean Symbols

Symbol	Name	Lean
$\wedge$	Conjunction (and)	`\and`
$\vee$	Disjunction (or)	`\or`
$\to$	Implies	`\r` or `\to`
$\leftrightarrow$	If and only if	`\iff`
$\bot$	False, contradiction	`false`
$\forall$	For all	`\all`
$\exists$	There exists	`\ex`
h_1	Subscript	`h\1`
$\langle$ and $\rangle$	French quotes	`\<` and `\>`

2.2 Reiteration

The reiteration rule states that p follows from $h : p$.

This is represented very simply in a term-style Lean proof. The first line of the proof below can be read as 'Given `h : p`, here follows a proof of `p`'.

```
example (h : p) : p :=
h -- Application of reiteration.
```

Reiteration is represented in a tactic-style proof by the `exact` tactic.

```
example (h : p) : p :=
begin
  exact h,
end
```

2.3 Conjunction (and)

2.3.1 Conjunction elimination

There are two conjunction elimination rules, left and right.

- (**Left and elimination**) given $h : p \wedge q$, we have a proof of p.
- (**Right and elimination**) given $h : p \wedge q$, we have a proof of q.

In a tactic-style proof, we can perform left and right and elimination simultaneously with the `cases` tactic. Below, `cases h with hp hq` introduces hypotheses `hp : p` and `hq : q` into the context.

```
example (h : p ∧ q) : q :=
begin
  cases h with hp hq, -- Equivalent to both left and right and elimination.
  exact hq, -- Closes the goal via reiteration using the proof term `hq`
end
```

Given `h : p ∧ q`, a proof term for `p ∧ q`, the terms `h.left` and `h.right` are proof terms for `p` and `q`, respectively.

```
example (h : p ∧ q) : q :=
h.right -- Term-style right and elimination.
```

We can use the `have, ..., from` notation to insert a term-style proof into a tactic-style proof. Below, `h.right` is a proof term for `q`.

```
example (h : p ∧ q) : q :=
begin
  have hq : q, from
    h.right, -- Term-style right and elimination.
  exact hq,
end
```

2.3.2 Conjunction introduction

Forward: given $h_1 : p$ and $h_2 : q$, we have a proof of $p \wedge q$.

Backward: to prove $p \wedge q$, it suffices to prove p and q.

The `split` tactic applies conjunction introduction backward.

```
example (h₁ : p) (h₂ : q) : p ∧ q :=
begin
  split, -- This replaces the goal p ∧ q with two new goals: 1. p and 2. q.
  { exact h₁, }, -- This closes the goal for p.
```

(continues on next page)

(continued from previous page)

```
  { exact h₂, }, -- This closes the goal for q.
end
```

The `and.intro` function, applied to `h₁ : p` and `h₂ : q`, gives a proof term for `p ∧ q`. This is a forward application of conjunction introduction.

```
example (h₁ : p) (h₂ : q) : p ∧ q :=
and.intro h₁ h₂
```

This can also be written using French quotes (a general Lean notation for the so-called constructor of an inductive data type).

```
variables p q : Prop

-- BEGIN
example (h₁ : p) (h₂ : q) : p ∧ q :=
⟨h₁, h₂⟩ -- Enter these 'French quotes' with `\<` and  `\>`
```

This proof term can be used within a tactic block.

```
example (h₁ : p) (h₂ : q) : p ∧ q :=
begin
  exact and.intro h₁ h₂ -- Or `exact ⟨h₁, h₂⟩`.
end
```

2.4 Implication

2.4.1 Implication elimination

Forward: given $h_1 : p \to q$ and $h_2 : p$, we have a proof of q.

Backward: given $h : p \to q$, to prove q, it suffices to prove p.

The `apply` tactic uses implication elimination backward.

```
example (h : p → q) (k : p) : q :=
begin
  apply h, -- This is a backward proof that changes the goal to proving p.
  exact k,
end
```

Given `h₁ : p → q` and `h₂ : p`, the expression `h₁ h₂` is a proof term for `q`. This is forward implication elimination.

```
example (h₁ : p → q) (h₂ : p) : q :=
h₁ h₂ -- h₁ h₂ is the result of implication elimination on h₁ and h₂.
```

As usual, this proof term can be used within a tactic block using the `exact` tactic.

```
example (h₁ : p → q) (h₂ : p) : q :=
begin
  exact h₁ h₂,
end
```

2.4.2 Implication introduction

Implication introduction: to prove $p \to q$, it suffices to assume $h : p$ and derive q.

Tactic-style, if the goal is to prove `p → q`, then `intro h` introduces an assumption `h : p` into the context and replaces the goal with one of proving `q`.

```
example (k : q) : p → q :=
begin
  intro h, -- This is equivalent to 'Assume h : p' in mathematics.
  exact k, -- We close the goal using our proof of q.
end
```

The term style proof is similar, using `assume` instead of `intro`.

```
example (k : q) : p → q :=
assume h,
  k
```

If desired, you can make the type of `h` explicit, when giving a term-style proof.

```
example (k : q) : p → q :=
assume h : p,
  k
```

2.5 Disjunction (or)

2.5.1 Disjunction introduction

There are two disjunction introduction rules, left and right.

Forward

- (**Left or introduction**) given $h : p$, we have a proof of $p \vee q$.
- (**Right or introduction**) given $h : q$, we have a proof of $p \vee q$.

Backward

- (**Left or introduction**) to prove p, it suffices to prove $p \vee q$.
- (**Right or introduction**) to prove q, it suffices to prove $p \vee q$.

The `left` and `right` tactics represent backward left or introduction and right or introduction, respectively.

```
example (h : p) : p ∨ q :=
begin
  left, -- This changes the goal, by left or introduction, to proving p
  exact h,
end
```

Forward, given `h : p`, the expression `or.inl h` is a proof term for `p ∨ q`. Likewise, if `h : q`, the expression `or.inr h` is a proof term for `p ∨ q`.

```
example (h : p) : p ∨ q :=
or.inl h
```

2.5.2 Disjunction elimination

Forward: given $h_1 : p \vee q$, $h_2 : p \to r$, and $h_3 : q \to r$, we have a proof of r.

Backward: given $h : p \vee q$, to prove r, it suffices to (1) assume $hp : p$ and deduce r and (2) assume $hq : q$ and deduce r.

Given `h : p ∨ q`, the `cases` tactic applied as `cases h with hp hq` replaces the goal of proving `r` with two subgoals: (1) to prove `r` with an additional assumption `hp : p` and (2) to prove `r` with an additional assumption `hq : q`.

In the example code below, we show two different methods of closing the resulting subgoals, corresponding, in turn, to term-style and tactic-style implication elimination.

```
example (h : p ∨ q) (h₂ : p → r) (h₃ : q → r) : r :=
begin
  cases h with hp hq,
  { exact h₂ hp, }, -- `h₂ hp` is implication elimination to give `r`.
  { apply h₃, exact hq, }, -- A tactic-style implication elimination.
end
```

Here is a more typical example of disjunction elimination.

```
example (h₁ : (p ∧ r) ∨ (r ∧ q)) : r :=
begin
  cases h₁ with h₂ h₂,
  { exact h₂.right, }, -- In this subproof, `h₂ : p ∧ r`. The subgoal is `r`.
  { exact h₂.left, }, -- In this subproof, `h₂ : r ∧ q`. The subgoal is `r`.
end
```

Given `h₁ : p ∨ q`, `h₂ : p → r`, `h₃ : q → r`, the function `or.elim` applied to `h₁`, `h₂`, and `h₃` gives a proof-term for``r``.

```
example (h₁ : p ∨ q) (h₂ : p → r) (h₃ : q → r) : r :=
or.elim h₁ h₂ h₃
```

Here is a term-style proof of the previous result.

```
example (h₁ : (p ∧ r) ∨ (r ∧ q)) : r :=
or.elim h₁
  (assume h₂ : p ∧ r, h₂.right) -- A term-style proof of `p ∧ r → r`
  (assume h₂ : r ∧ q, h₂.left) -- A term-style proof of `r ∧ q → r`
```

2.6 If and only if (iff)

2.6.1 Iff elimination

There are two iff elimination rules, left and right.

- (**Left iff elimination**) given $h : p \leftrightarrow q$, we have a proof of $p \to q$.
- (**Right iff elimination**) given $h : p \leftrightarrow q$, we have a proof of $q \to p$.

Note the similarity with this and conjunction elimination.

Given `h : p ↔ q`, the `cases` tactic, when applied as `cases h with h₁ h₂`, decomposes the hypothesis `h` into two hypotheses, `h₁ : p → q` and `h₂ : q → p`. This is the same as left and right iff elimination simultaneously.

```
example (h : p ↔ q) : p → q :=
begin
  cases h with h₁ h₂,
  exact h₁,
end
```

Likewise, given `h : p ↔ q`, `iff.elim_left h` is a proof term for `p → q` and `iff.elim_right h` is a proof term for `q → p`.

```
example (h : p ↔ q) : p → q :=
iff.elim_left h
```

2.6.2 Iff introduction

Forward: given $h_1 : p \to q$ and $h_2 : q \to p$, we have a proof of $p \leftrightarrow q$.

Backward: to prove $p \leftrightarrow q$, it suffices to prove $p \to q$ and $q \to p$.

The `split` tactic applies iff introduction backward.

```
example (h₁ : p → q) (h₂ : q → p) : p ↔ q :=
begin
  split, -- This replaces the goal `p ↔ q` with 1. p → q and 2. q → p.
  { exact h₁, }, -- Closes the goal `p → q`.
  { exact h₂, }, -- Closes the goal `q → p`.
end
```

The `iff.intro` function, applied to `h₁ : p → q` and `h₂ : q → p`, gives a proof term for `p ∧ q`. This is a forward application of iff introduction.

```
example (h₁ : p → q) (h₂ : q → p) : p ↔ q :=
iff.intro h₁ h₂
```

2.7 False and negation

2.7.1 False elimination

The symbol $\bot$, referred to as false or contradiction or arbitrary contradiction, is referred to in one fundamental rule of inference, *ex falso sequitur quodlibet*, also called *ex falso* or false elimination. This rule states that anything follows from false.

Forward: given $h : \bot$, we have a proof of p.

Backward: to prove p, it suffices to prove $\bot$.

The `exfalso` tactic represents backward false elimination.

```
example (h : false) : p :=
begin
  exfalso, -- This changes the goal from `p` to `false`.
  exact h, -- We close the goal with `h`.
end
```

Given `h : false`, the expression `false.elim h` is a proof term for `p`.

```
example (h : false) : p :=
false.elim h
```

2.7.2 False introduction

The expression $\neg p$ is a shorthand for $p \to \bot$. The rule of false introduction is thus merely implication elimination in another guise.

Forward: given $h_1 : \neg p$ and $h_2 : p$, we have a proof of $\bot$.

Backward: given $h : \neg p$, to prove $\bot$, it suffices to prove p.

The `apply` tactic uses false introduction backward.

```
example (h : ¬p) (k : p) : false :=
begin
  apply h, -- This changes the goal to proving `p`.
  exact k,
end
```

Given `h₁ : ¬p` and `h₂ : p`, the expression `h₁ h₂` is a proof term for `false`. This is forward false introduction.

```
example (h₁ : ¬p) (h₂ : p) : false :=
h₁ h₂
```

2.7.3 Negation introduction

As $\neg p$ is shorthand for $p \to \bot$, the rule of negation introduction is really just implication introduction.

Negation introduction: to prove $\neg p$, it suffices to assume $h : p$ and derive $\bot$.

Tactic-style, if the goal is to prove `¬p`, then `intro h` introduces an assumption `h : p` into the context and replaces the goal with one of proving `false`.

```
example (k : false) : ¬p :=
begin
  intro h, -- This is equivalent to 'assume h : p' in mathematics.
  exact k, -- We close the goal using our proof of `false`.
end
```

The term-style proof is similar.

```
example (k : false) : ¬p :=
assume h : p,
  k
```

2.8 Summary

2.8.1 Tactic-style

Table 2.2 summaries the Lean tactics that represent rules of propositional logic.

Table 2.2: Tactic-Style

Connective	Introduction	Elimination
$\wedge$, conjunction	`split`	`cases h with` h_1 h_2
$\vee$, disjunction	`left` `right`	`cases h with` k_1 k_2
$\to$, implication	`intro h`	`apply h`
$\leftrightarrow$, iff	`split`	`cases h with` h_1 h_2
`false`, false	`exfalso`	`apply h`
$\neg$, negation	`intro h`	N/A

2.8.2 Term-style

Table 2.3 summaries the Lean functions that represent rules of propositional logic and produce proof terms.

Table 2.3: Term-Style

Connective	Introduction	Elimination
$\wedge$, conjunction	`and.intro` h_1 h_2 or $\langle h_1, h_2 \rangle$	`h.left` `h.right`
$\vee$, disjunction	`or.inl h` `or.inr h`	`or.elim` h_1 h_2 h_3
$\to$, implication	`assume h : P,` *followed by a proof term for* `Q`.	h_1 h_2
$\leftrightarrow$, iff	`iff.intro` h_1 h_2	`iff.elim_left h` `iff.elim_right h`
`false`, false	h_1 h_2	`false.elim h`
$\neg$, negation	`assume h : P,` *followed by proof of* `false`.	N/A

CHAPTER

THREE

PREDICATE LOGIC

3.1 Types

Every term in mathematics has a type. For instance, $\mathbb{Z}$ is the type of integers. A statement such as $5 : \mathbb{Z}$ is called a (typing) judgment. This statement can be read '5 has type $\mathbb{Z}$' or '5 is an integer'.

We can work with arbitrary terms and types. The judgment $x : U$ is read 'x has type U'.

Propositional variables, as discussed in Section 2.1, are terms of type `Prop`.

Suppose we have `P : Prop` and `Q : Prop`. What is meant by `h : P ∧ Q`? When a proposition, such as `P ∧ Q` is viewed as a type, it is the type of proofs of that proposition. Thus the judgment `h : P ∧ Q` should be interpreted as '`h` is a proof of `P ∧ Q`'.

A new type can be introduced into Lean using the following syntax.

```
variable U : Type* -- This declares `U` to be a type.
```

3.2 Functions and definitions

A function (also called a definition) is a mathematical object with a name, zero or more inputs, a body, and a body type.

Below, we define three functions in Lean. The last function has name `avg`. It takes two inputs, `x` and `y`, both of type $\mathbb{Z}$, and has a body `(x + y)/2`, also of type $\mathbb{Z}$ (note that integer division automatically rounds in Lean).

```
def double (x : ℤ) : ℤ := 2 * x
def square (y : ℤ) : ℤ := y * y
def avg (x y : ℤ) : ℤ := (x + y)/2
```

Some functions can be evaluated. For example, with the above definitions, you could find the average of 10 and 6 as follows.

```
#eval avg 10 6
```

Here, `avg 10 6` represents the result of replacing `x` with `10` and `y` with `6` in the body of the defintion of `avg`.

One may check the type of any Lean object using the `#check` directive. On entering the following commands, Lean will respond with `double : ℤ → ℤ` and `avg : ℤ → ℤ → ℤ`.

```
#check double
#check avg
```

Here, $\mathbb{Z} \to \mathbb{Z}$ is the type of functions that take one integer input and produce one integer output, while $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ is the type of functions that take two integer inputs and produce one integer output. Can you guess the type of `square`?

More generally, if `U` and `V` are types, we can declare, without defining them, functions from `U` to `V`. The following declares a function `f` of type `U → V`. That is, `f` is a function from `U` to `V`.

Here, `f x` denotes the result of applying `f` to `x`. Naturally, the type of the result is `V`.

```
variables U V : Type*
variable x : U
variable f : U → V
#check f
#check f x
```

3.3 Currying functions

This section is optional reading.

I lied earlier when I wrote that $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ is the type of functions that take two integer inputs and produce one integer output. To begin, we should clarify whether $\mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ means $\mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$ or $(\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z}$. By convention, the former meaning, $\mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$ is used. This is chosen to harmonise with the notion of function application, as we'll see at the end of this section.

In truth, $\mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$ is therefore the type of functions that take one integer argument and returns *a function* of type $\mathbb{Z} \to \mathbb{Z}$

Thus `avg 5` is a function that takes takes an input, say `y`, and returns `(5 + y)/2`. This is called *partial application* of `avg`. To make this more transparent, we define a new function `avg'` to be the partial application of `avg` at 5.

```
def avg' := avg 5

#eval avg' 17 -- This outputs 11.
```

It transpires that this is the most natural way to think of functions of several variables when proving theorems.

Alternatively, the function `avgu` defined below is actually a function that takes a pair of variables (as indicated by the input type $\mathbb{Z} \times \mathbb{Z}$) and returns an integer.

```
def avgu : ℤ × ℤ → ℤ
| (x, y) := (x + y)/2

#eval avgu (10,6)  -- This displays 8.
```

Producing `avg`, a function that produces a function, from `avgu`, a function of many variabes, is called *currying*, after American mathematician Haskell Curry. The reverse process is called *uncurrying*.

One may consider functions of more than two variables.

```
def avg_three_u : (ℤ × ℤ × ℤ) → ℤ
| (x, y, z) := (x + y + z)/3
#check avg_three_u -- `avg_three_u` has type `ℤ × ℤ × ℤ → ℤ`
#eval avg_three_u (10, 5, 6) -- This is 7.
```

Consider the curried version of this function, which we call `avg_three`.

```
def avg_three (x y z : ℤ) : ℤ := (x + y + z)/2

#check avg_three  -- `avg_three` has type `ℤ → ℤ → ℤ → ℤ`, i.e., `ℤ → (ℤ → (ℤ →␣
↪ℤ))`
#check (avg_three 10)       -- `avg_three 10` has type `ℤ → (ℤ → ℤ)`.
#check (avg_three 10 5)     -- `avg_three 10 5` has type `ℤ → ℤ`.
#check (avg_three 10 5 6)   -- `avg_three 10 5 6` has type `ℤ`, i.e. is an integer.
```

`avg_three` has type `ℤ → (ℤ → (ℤ → ℤ))`. That is, it takes an integer input and outputs a term of type `ℤ → (ℤ → ℤ)`. But a term of this type *is* a function that takes an integer input and outputs a term of type `ℤ → ℤ`. In its turn this is a function that takes an integer input and outputs a term of type `ℤ`.

We see this through successive partial applications of `avg_three`.

Indeed, when we write something like `avg 10 6`, we really mean `(avg 10) 6`. That is we take the function `avg 10` and apply it to `6`.

Likeiwse, `avg_three 10 5 6` really means `(((avg_three 10) 5) 6`. Note how the bracketing convention for function application is the opposite of the convention for function types.

3.4 Predicates

A predicate is a function whose body type is `Prop`. Below, we define the predicate `even` so that `even x` is the proposition `∃ m : ℤ, x = 2 *m`. The symbol `∃` is read 'there exists'. So this proposition can be interpreted as, 'there exists an integer m such that $x = 2m$'.

```
def even (x : ℤ) : Prop := ∃ m : ℤ, x = 2 *m
#check even
#check even 5
```

The result of `#check` assures us that `even` has type `ℤ → Prop`. It is a function that takes one input of type `ℤ` and has a body of type `Prop`. Moreover `even 5` has type `Prop`.

```
#check even
#check even 5
```

Predicates can take more than one input. The following predicate takes two integer inputs `a` and `b` and has body `∃ m : ℤ, b = a * m`. In familiar language, it represents the notion that `a` divides (i.e. is a factor of) `b`.

```
def divides (a b : ℤ) : Prop := ∃ m : ℤ, b = a * m
```

When working abstractly, we can declare, without defininig it, a predicate on an arbitrary type.

```
variable U : Type*        -- Declare a type, `U`.
variable x : U            -- Declare a term `x`, of type `U`.
variable P : U → Prop     -- Declare a predicate `P` on `U`.

#check P x                -- `P x` has type `Prop`
#check P                  -- `P` has type `U → Prop`
```

Here, `P x` is the result of applying `P` to `x`. It has type `Prop`, while `P` itself has type `U → Prop`.

We may define abstract predicates on more than one type.

```
variables (U : Type*) (V : Type*) -- Declare types `U` and `V`.
variables (x : U) (y : V)         -- Declare terms `x` of type `U` and `y` of type␣
↪`V`.

variable Q : U → V → Prop    -- Declare a predicate `Q` on `U` and `V`.

#check Q      -- `Q` is a predicate with type `U → V → Prop`.
#check Q x    -- `Q x` is a predicate with type `V → Prop`.
#check Q x y -- `Q x y` has type `Prop`, i.e. is a proposition.
```

The next two paragraphs are technical and may be omitted if you have not read Section 3.3.

When viewed through the lens of currying functions, the predicate `Q` can be thought of as a function that takes an input of type `U` and outputs a function of type `V → Prop`.

The observant reader will note that this contradicts by previous definition that a predicate is a function with body type `Prop`. That's because I lied to keep things simple. Really, I mean that for a function to be a predicate, its *uncurried* version should have body type `Prop`. The uncurried version of `Q` has type `U × V → Prop`, so indeed its body type is `Prop`.

3.5 Universal quantification

The universal quantifier, written $\forall$ is one of the two operators of predicate logic. It is read 'for all', 'for every', or 'for each'. Informally, $\forall x, P(x)$ is the assertion that $P(x)$ holds for every x.

Usually, the type of x in the above expression can be inferred from the type of P. To be explicit, we can use a type ascription $x : U$ as in the expression $\forall (x : U), P(x)$.

Formally, the meaning of the universal quantifier is defined by two rules of inference.

3.5.1 For all elimination

Rule 66 (For all elimination, forward)**.** *Let U be a type and let P be a predicate on U. Given $h : \forall x, P(x)$ and given $u : U$, we have $P(u)$.*

In Lean, if `P : U → Prop` is a predicate, given `h : ∀ x, P x` and given `u : U`, the expression `h u` is a proof term for `P u`. Note the similarity between this and the Lean notation for implication elimination.

```
variables (U : Type*) (P : U → Prop) (u : U)

example (h : ∀ x, P x) : P u :=
by exact h u
```

Alternatively, the `specialize` tactic applies to a universally quantified statement `h : ∀ x, P x`. Writing `specialize h u` replaces `h` with `h : P u`.

```
example (h : ∀ x, P x) : P u :=
begin
  specialize h u,    -- By for all elimination on `h` and `u`, we have `h : P u`
  show P u, from h,  -- We show `P u` by reiteration on `h`.
end
```

Rule 67 (For all elimination, backward)**.** *Let U be a type and let P be a predicate on U. To prove $P(u)$, $h : \forall x, P(x)$ and given $u : U$, we have $P(u)$.*

In Lean, we invoke backward for all elimination using the `apply` tactic, just as we did for backward implication elimination. Below, Lean is clever enough to close the goal immediately after `apply h` as `u : U` is in the context.

```
example (h : ∀ x, P x) : P u :=
by apply h
```

Let's do something a little more interesting.

Example 68. *Let S and T be prediates on a type U. Given $h_1 : \forall x, S(x)$, $h_2 : \forall y, S(y) \to T(y)$ and $u : U$, we have $T(u)$.*

We give first a forward proof.

```
variables (U : Type*) (S T : U → Prop) (u : U)

example (h₁ : ∀ x, S x) (h₂ : ∀ y, S y → T y) (u : U) : T u :=
begin
  have h₃ : S u, from h₁ u,  -- We have `h₃ : S u` by for all elim. on `h₁` and `u`.
  have h₄ : S u → T u, from h₂ u, -- We have `h₄ : S u → T u` by for all elim. on␣
↪`h₂` and `u`
  show T u, from h₄ h₃, -- We show `T u` by implication elimination on `h₄` and `h₃`.
end
```

The same proof can be written more concisely using `specialize`.

```
example (h₁ : ∀ x, S x) (h₂ : ∀ y, S y → T y) (u : U) : T u :=
begin
  specialize h₁ u,  -- We have `h₁ : S u` by for all elim. on `h₁` and `u`.
  specialize h₂ u, -- We have `h₂ : S u → T u` by for all elim. on `h₂` and `u`
  show T u, from h₂ h₁, -- We show `T u` by implication elimination on `h₂` and `h₁`.
end
```

In the following backward Lean proof, `apply h₂` invokes for all elimination followed by implication elimination on the hypothesis `h₂`.

```
example (h₁ : ∀ x, S x) (h₂ : ∀ y, S y → T y) (u : U) : T u :=
begin
  apply h₂, -- By for all elim. on `h₂` and `u`, followed by imp. elim., it suffices␣
↪to prove `S u`.
  apply h₁, -- The result follows by for all elim. on `h₁` and `u`.
end
```

In the next example, we construct a predicate using two others.

Below, we have predicates `S` and `T` on a type `U`. The function that takes `x : U` to `(S x) ∧ (T x)` is also a predicate. We assume the universally quantified statement `h : ∀ x, (S x) ∧ (T x)`. By for all elimination applied to `h` and `u : U`, we have `(S u) ∧ (T u)`. We can extract `S u` from this by left conjunction elimination.

```
example (h : ∀ x, (S x) ∧ (T x)) : S u :=
begin
  have h₂ : (S u) ∧ (T u), from h u, -- We have `h₂ : (S u) ∧ (T u)` by for all␣
↪elimination on `h` and `u`.
  show (S u), from h₂.left, -- We show `S u` by left conjunction elimination on `h₂`.
end
```

For a more familiar example, we'll show $(\forall x : \mathbb{Z}, x^2 \geq 0) \to (-4)^2 \geq 0$.

Proof. Assume $h : \forall x : \mathbb{Z}, x^2 \geq 0$. It suffices to prove $(-4)^2 \geq 0$. But $-4 : \mathbb{Z}$. The result follows by for all elimination on h and -4. □

In the Lean code below, we need to use the type ascription `4 : ℤ`. The reason is that Lean, by default, interprets numerals as terms of type ℕ. It then balks at -4.

```
example : (∀ x : ℤ, x^2 ≥ 0) → ((-(4 : ℤ))^2 ≥ 0) :=
begin
  intro h,
  exact h (-4),
end
```

3.5.2 For all introduction

Rule 69 (For all introduction)**.** *Let P be a predicate on a type U. To prove $\forall x, P(x)$ is to assume $u : U$ and derive $P(u)$.*

Again, note the similarity between this rule and implication introduction.

All the results we've seen so far that begin with, 'Let P and Q be propositions' can be replaced with universally quantified statements that don't specify the names of the propositions.
Theorem 70 (Commutativity of conjunction (IV))**.** *We have* $\forall P : \text{Prop}, \forall Q : \text{Prop}, P \wedge Q \leftrightarrow Q \wedge P$.

Proof. Assume R and S are propositions. It suffices to show $R \wedge S \leftrightarrow S \wedge R$. But this follows by Theorem 25. □

In Lean, we use `intro` to denote for all introduction (as we do for implication introduction).

```
example : ∀ p q : Prop, p ∧ q ↔ q ∧ p :=begin
  intros r s, -- Assume `r : Prop` and `s : Prop`. It suffices to prove `r ∧ s ↔ s ∧␣
↪r`.
  split; -- By iff intro., it suffices to prove 1. `r ∧ s → s ∧ r` and 2. `s ∧ r → r␣
↪∧ s`. We'll use the same proof in each case.
  { intro h, exact ⟨h.2, h.1⟩, }, -- Assume the antecedent, `h`. The goal is closed by␣
↪and intro. on `h.1` and `h.2`
end
```

Our final example uses both for all introduction and for all elimination.
Example 71. *Let P and Q be predicates on a type U. We have*

$$(\forall x, P(x) \wedge Q(x)) \to (\forall y, Q(y) \wedge P(y)).$$

Here is the Lean proof, with the matheatical proof given in the comments.

```
variables (U : Type*) (P Q : U → Prop)

example : (∀ x, P x ∧ Q x) → (∀ y, Q y ∧ P y) :=
begin
  intro h, -- Assume `h : ∀ x, P x ∧ Q x`. By `→` intro., it suffices to prove `∀ y,␣
↪Q y ∧ P y`.
  intro u, -- Assume `u : U`. By `∀` intro, it suffices to prove `Q u ∧ P u`.
  rw and_comm,  -- By commutativity of conjunction, it suffices to prove `P u ∧ Q u`.
  exact h u, -- This follows by `∀` elim. on `h` and `u`.
end
```

3.6 Existential quantification

The existential quantifier, written $\exists$ is read 'there exists', 'there is', or 'for some'. Informally, $\exists x, P(x)$ is the assertion that there is some u for which $P(u)$ holds.

As with the universal quantifier, we can make the types explicit via a type ascription, writing $\exists (x : U), P(x)$, for example.

Formally, the meaning of the universal quantifier is defined by two rules of inference.

3.6.1 Exists introduction

Rule 72 (Exists introduction, forward)**.** *Let P be a predicate on a type U. Given $u : U$ and $h : P(u)$, we have a proof of $\exists x, P(x)$.*
Example 73. *Given $5 : \mathbb{N}$ and $h : 5 \geq 3$, we have $\exists x, x \geq 3$.*

The proof is simply an application of exists introduction to $5 : \mathbb{N}$ and h.

In Lean, if `u : U` and `h : P u`, then `exists.intro u h` is a proof of `∃ x, P x`

```
example (h : 5 ≥ 3) : ∃ x, x ≥ 3 :=
by exact exists.intro 5 h
```

To be more concise, we can use the anonymous constructor notation.

```
example (h : 5 ≥ 3) : ∃ x, x ≥ 3 :=
by exact ⟨5, h⟩
```

Rule 74 (Exists introduction, backward)**.** *Let P be a predicate on a type U. Given $u : U$, to prove $\exists x, P(x)$, it suffices to prove $P(u)$.*
Example 75. *Let P and Q be predictes on a type U. Given $u : U$, $h_1 : P(u) \to Q(u)$ and $h_2 : P(u)$, we have a proof of $\exists x, Q(x)$.*

Proof. By exists introduction on u, it suffices to prove $Q(u)$. But this follows by implication elimination on h_1 and h_2. □

In Lean, the `use` tactic indicates backward exists introduction. We use this to give a Lean proof of the example above.

```
example (u : U) (h₁ : P u → Q u) (h₂ : P u) : ∃ x, Q x :=
begin
  use u,        -- By `∃` intro. on `u`, it suffices to prove `Q u`
  exact h₁ h₂, -- This follows by `→` elim. on `h₁` and `h₂`.
end
```

Here is a forward Lean proof of the same result.

```
example (u : U) (h₁ : P u → Q u) (h₂ : P u) : ∃ x, Q x :=
begin
  have h₃ : Q u, from h₁ h₂, -- We have `Q u` from `→` elim. on `h₁` and `h₂`.
  exact exists.intro u h₃,   -- The result follows from exists intro. on `u` and `h₃`.
end
```

3.6.2 Exists elimination

Suppose you want to prove, ‘A person in Britain has competed in the Olympics’ and you know, ‘there exists a person in Britain who has won a gold medal at the Olympics’

3.7 Negating quantifiers

3.8 Mixing quantifiers

3.9 Functions and equality