True

We formalize the proposition, *true* as an inductively defined type, as follows:

```
inductive true : Prop
| intro : true
```

This proposition has one *introduction* rule: a proof called *true.intro*. It is always available a proof of *true*.

A proof of true carries no useful information. There is no way to *use* a proof of *true*. In other words, there is no useful *elimination* rule for a proposition of this form.

False

We formalize the proposition, *false* as an inductively defined type, as follows:

Try it in Lean online!

```
inductive false : Prop
```

The *false* proposition has *no* constructors, thus no proofs. In other words, there are no *introduction* rules for *false*.

Although one cannot construct a proof of false from a set of consistent assumptions, it's always possible to obtain a proof of false by *assuming* that one exists or by reaching a contradiction in one's reasoning.

Here for example are two equivalent functions where, in the body of the function one has assumed a proof of false.

Try it in Lean online!

```
def a_function (f : false) : 0 = 1 :=
_

def a_function'' : false → 0 = 1 :=
λ (f : false), _

def a_function' : false → 0 = 1
| f := _
```

This assumption is clear when one looks at the context at the points where underscores appear in the function definitions.

Try it in Lean online!

```
f : false
⊢ 0 = 1
```

While *false* has no introduction rule, it does have an elimination rule. Given a proof of false as an assumption or as a consequence of a contradition, one can conclude a proof immediately by applying this rule. We call it the *false elimination* rule. In Lean it is called *false.elim*. When applied to a proof of *false*, this rule ends the proof with no remaining goals to be proved.

Try it in Lean online!

```
def a_function (f : false) : 0 = 1 :=
false.elim f

def a_function'' : false → 0 = 1 :=
λ (f : false), false.elim f

def a_function' : false → 0 = 1
| f := false.elim f

def a_function''' (f : false) : 0 = 1 :=
match f with
end
```

a=b

Given any *type*, α, equality is a reflexive, symmetric, and transitive binary relation on objects of this type. It is thus also said to be an *equivalence relation* on objects of this type.

Given a type, α, this relation is defined by a predicate, *eq* with two argument terms, such as *a* and *b*, of type α. The expression *eq a b* is thus a proposition, usually written as *a = b*.

This proposition is taken to assert that *a* and *b* are equal in value. Examples of equality propositions include *0 = 0* and *0 = 1*.

We formalize the concept of equality as a polymorphic inductive type as follows:

Try it in Lean online!

```
inductive eq {α : Type} (a : α) : α → Prop
| refl : eq a
```

The first line states that this polmorphic type has one type argument, α; and that it takes two values of this type: one named *a*, and a second unnamed argument (after the colon).

The constructor, *eq.refl*, formalizes the single introduction rule for equality. It establishes that given any value, *a* of type α, the term, *eq.refl a*, will be accepted as a proof of *a = a*.

Because this is the only constructor (introduction rule) for equality, it is not possible to construct a proof of *a = b*, where *a* and *b*are different values.

It is possible to prove *a = b* even if *a* and *b* are different terms, as long as they reduce to the same values. So, for example, it is possible to construct a proof of *2 + 2 = 4*. It's just *eq.refl 4*.

Try it in Lean online!

```
example : 2 + 2 = 4 :=
  eq.refl 4
```

The eliimination rule for *using* proof of equalities is called *by the substitutability of equals*. It is formalized in Lean by the rule, *eq.subst*. This rule states that if you know (if you have a proof) that *a = b*, and if you know (have a proof) that *a* has some property, *P* (i.e., that the proposition *P a* is true), then you can deduce/conclude that *b* also has property, *P* (i.e., that *P b* is true).

This rule allows you to subsitute *b* for *a* in any proof in which *a* appears, such as a proof of *P a*, yielding a proof for *b*, such as a proof of *P b*.

Try it in Lean online!

```
example : ∀ (α : Type) (P : α → Prop) (a b : α),
            (a = b) → P a → P b :=
begin
assume (α : Type),
assume (P : α → Prop),
assume (a b : α),
assume (heq : a = b),
assume (pa : P a),
exact (eq.subst heq pa)
end

example : ∀ (α : Type) (P : α → Prop) (a b : α),
            (a = b) → P a → P b :=
  λ α P a b heq pa,
    eq.subst heq pa
```

PAndQ

Given any *type*, α, equality is a reflexive, symmetric, and transitive binary relation on objects of this type. It is thus also said to be an *equivalence relation* on objects of this type.

Given a type, α, this relation is defined by a predicate, *eq* with two argument terms, such as *a* and *b*, of type α. The expression *eq a b* is thus a proposition, usually written as *a = b*.

This proposition is taken to assert that *a* and *b* are equal in value. Examples of equality propositions include *0 = 0* and *0 = 1*.

We formalize the concept of equality as a polymorphic inductive type as follows:

Try it in Lean online!

```
inductive eq {α : Type} (a : α) : α → Prop
| refl : eq a
```

The first line states that this polmorphic type has one type argument, α; and that it takes two values of this type: one named *a*, and a second unnamed argument (after the colon).

The constructor, *eq.refl*, formalizes the single introduction rule for equality. It establishes that given any value, *a* of type α, the term, *eq.refl a*, will be accepted as a proof of *a = a*.

Because this is the only constructor (introduction rule) for equality, it is not possible to construct a proof of *a = b*, where *a* and *b*are different values.

It is possible to prove *a = b* even if *a* and *b* are different terms, as long as they reduce to the same values. So, for example, it is possible to construct a proof of *2 + 2 = 4*. It's just *eq.refl 4*.

Try it in Lean online!

```
example : 2 + 2 = 4 :=
  eq.refl 4
```

The eliimination rule for *using* proof of equalities is called *by the substitutability of equals*. It is formalized in Lean by the rule, *eq.subst*. This rule states that if you know (if you have a proof) that *a = b*, and if you know (have a proof) that *a* has some property, *P* (i.e., that the proposition *P a* is true), then you can deduce/conclude that *b* also has property, *P* (i.e., that *P b* is true).

This rule allows you to subsitute *b* for *a* in any proof in which *a* appears, such as a proof of *P a*, yielding a proof for *b*, such as a proof of *P b*.

Try it in Lean online!

```
example : ∀ (α : Type) (P : α → Prop) (a b : α),
            (a = b) → P a → P b :=
begin
assume (α : Type),
assume (P : α → Prop),
```

```
assume (a b : α),
assume (heq : a = b),
assume (pa : P a),
exact (eq.subst heq pa)
end

example : ∀ (α : Type) (P : α → Prop) (a b : α),
            (a = b) → P a → P b :=
  λ α P a b heq pa,
    eq.subst heq pa
```

P imp Q

Given any two propositions, *P* and *Q*, one can form the proposition, $P \rightarrow Q$. We intend for it to mean that if there is a proof *(p : P)*, that is, if *P* is true), then a proof of*Q* can be constructing, showing that *Q* must be true as well. In short, the proposition $P \rightarrow Q$, an implication, is true if whenever *P* is true, so is *Q*.

To prove that an implication, $P \rightarrow Q$, is true it suffices to show that if one is given or assumes a proof of *P*, then one can construct and return a proof of *Q*. In other words, to prove $P \rightarrow Q$ one in effect constructs a function that takes a proof (p : P) as an argument. Within the body of the function it is assumed that *p* is some arbitrary proof of *P*. If in that context, one can return a proof of *Q* then one has shown that if there is (or is assumed to be) a proof of *P*then there is a proof of *Q*; so if P is true then so is Q. And that is what it means for the implication to be true.

This is the introduction rule for arrow: assume one is given a value (proof) of the premise, and show that in that context one can return a proof (value) of the conclusion (type).

Here's a simple example. The assertion is that if any natural number, n, is equal to 3, then n^2 is equal to 9. See the following tactic script. The proof is in four steps: first use forall introduction: assume that n is an arbitrary natural number, then prove the rest. Second, we use arrow introduction: assume that n = 3, then prove what remains. Third, use equals elimination, by using the assumed equality, n = 3, to rewrite n as 3 in the goal. The rewrite tactic uses substitutability of equals (eq.subst) "under the hood"). All that then remains is to prove now is that 3 ^ 2 = 9, and this is done by appealing to the symmetric property of the equality relation, using eq.refl to finish the proof.

Try it in Lean online!

```
example : ∀ (n : ℕ), n = 3 → n^2 = 9 :=
begin
intros,
rewrite a,
exact eq.refl 9,
end
```

To use a proof, h, of an implication, $P \to Q$, we recall that h is formalized as a *function*, one that if applied to a proof of P will return a proof of Q. So, to use a proof, h, of $P \to Q$, we also have to have a proof of $(p : P)$. In that case we can use h by applying it to p, as in (h p), to obtain a proof of Q.

We can see this idea clearly in the following simple proof that implies is transitive.

Try it in Lean online!

```
example : ∀ (P Q R : Prop), (P → Q) → (Q → R) → (P → R) :=
begin
assume P Q R,
assume pq qr,
assume p,
exact qr (pq p)
end
```

To prove the proposition, we first assume that P Q and R are arbitrary propositions (forall introduction). Next we assume that (P → Q) (arrow introduction) and that (Q → R) (arrow introduction again). Now in this context, what remains to be proved is (P → R). We once again use arrow introduction to assume that we have a proof (p : P). Now we have two functions (proofs of (P → Q) and (Q → R)) along with a proof (p : P).

To finish the proof, we first apply the proof of (P → Q) to p to get a proof of Q, then we apply the proof of (Q → R) to that proof of Q to get a proof of R. That completes the proof. QED.

P Equiv Q

Given any two propositions, P and Q, one can form the proposition, $P \leftrightarrow Q$. It is pronounced as "P is equivalent to Q" or "P if and only if Q." In mathematical writing, it is often written as "P iff Q", with an extra "f" in the "iff".

We intend for it to mean that if P is true then so is Q, and vice versa: if Q is true then so is P. In other words it's intended to meant that both (P → Q) and (Q → P) are true.

The usual presentation of a proof of $P \leftrightarrow Q$ is thus in two parts: first one proves (P → Q), then one proves (Q → P). With both such proofs in hand, one can then deduce a proof of $P \leftrightarrow Q$. This is the introduction rule for iff.

These ideas are formalized in Lean's polymorphic logical type, *iff P Q*. Here is its definition.

Try it in Lean online!

```
structure iff (a b : Prop) : Prop :=
```

```
intro :: (mp : a → b) (mpr : b → a)
```

It takes two propositions, P and Q as arguments (in the Lean libraries they are called a and b; the definition above is straight from the Lean library). Given these arguments, iff forms the proposition, *iff P Q*, i.e., $P \leftrightarrow Q$. The notation $P \leftrightarrow Q$ is just shorthand for *iff P Q*.

The iff connective provides one proof constructor (introduction rule), *iff.intro*. The constructor takes two arguments: one, called mpr (for reasons irrelevant here) must be a proof of $(P \to Q)$; the other, mpr, a proof of $(Q \to P)$. If *pq* is a proof of $(P \to Q)$ and *qp* is a proof of $(Q \to P)$, then the term, *iff.intro pq qp* type checks as a proof of *iff P Q*.

Just as a proof of a conjunction is essentially a pair of proofs, so is a proof of *iff P Q*. The elimination rules for *iff* are thus essentially the same as for *and P Q*. There are two such rules. The *iff.elim_left* rule applied to a proof of *iff P Q* returns a proof of $(P \to Q)$, while the *iff.elim_right* rule returns a proof of $(Q \to P)$.

Not P

If *P* is any proposition, then so is *not P*, usually written as $\neg P$. What such a proposition is meant to assert is that P is not true. In constructive logic for P to be not true means that there is no proof of P; for if there were, then P would be true.

We formalize the meaning of $\neg P$ by defining it to mean $P \to false$. Here is the precise definition of *not* in Lean: *def not : Prop → Prop := λ (a : Prop), a → false*. The not operator is formalized as a function. When applied to any proposition, P, it reduces to the proposition, $P \to false$.

To show that $\neg P$ is true means to show that $P \to false$ is true. To prove this implication, assume that there is a proof (p : P), and in this context show that a contradiction (a proof of false) can be derived.

This is the proof strategy called *proof by negation*. If your goal is to prove $\neg P$, assume P and show that doing so leads to a contradiction.

In computational terms, to prove $P \to false$, one shows that there is a function of this type by defining such a function. The mere existence of such a function shows that if there were a proof of P then there is a proof of false. Because false is an uninhabited type, it has no proofs, thus the existence of such function shows there cannot be a proof of P. That is what it means for $\neg P$ to be true.

Consider, for example, the proposition, $\neg\, 0 = 1$. To prove it, we reduce the expression to $0 = 1 \to false$. Now we apply arrow elimination by assuming we have a proof of $0 = 1$, then we show that in that context a proof of false can be returned.

Having assumed that there is a proof, h, of *0 = 1*, if we can show that there is a proof of false for for each possible form (case) of h, then we've shown that a proof of false can be derived from h no matter its form. So how many cases are there?

The answer is that there are no cases at all! The only way to form of proof an an equality is with eq.refl, but it takes only one argument and so can never form a proof that two different values (such as zero and one) are equal. With no cases left to prove, the overall proof is done!

This reasoning is clear in the empty pattern match (with zero cases) in the following proof.

Try it in Lean online!

```
example : ¬ 0 = 1 :=
λ h,
    match h with
    end
```

Here is the equivalent proof as a tactic script. What the cases tactic does is to give one subgoal for each each constructor application that could have produced h. One then needs to show that *in each case*the goal follows. With no further cases to consider, and so the proof is done.

Try it in Lean online!

```
example : ¬ 0 = 1 :=
begin
assume h : 0 = 1,
cases h,
end
```

Of course one can never actually use this proof/function to obtain a proof of false, because, as we've just shown, there can be no proof of 0 = 1 to which one could apply this function in the first place.

Proof by negation is the introduction rule for ¬ (negation). One uses it to prove propositions of the form, ¬ *P*.

The elimination rule for negation in predicate logic could also be called double negation elimination. To prove some proposition, *P*, one can do it in two steps. First, show that assuming ¬ *P* leads to a contradiction. This proves ¬ *(¬ P)*.Second, use the rule of negation elimination to derive a proof of*P* from the proof of ¬ *(¬ P)*.

This proof strategy is called *proof by contradiction*. Like proof by negation, it begins by assuming the opposite of the goal to be proved and then shows that this assumption leads to a contradiction, proving the negation of the assumption. Unlike proof by negation, proof by contradiction, however, relies on the rule of negation elimination. Carefully compare these two proof strategies to understand this crucial difference.

The subtle thing about negation elimination is that it is not valid in*constructive* predicate logic! So proof by contradition is not a valid proof strategy. An attempt to prove ¬ *(¬ P) → P* in Lean leads to failure. Consider that what ¬ *(¬ P) → P* means, by the definition of not, is *(P → false) → false*. There is no way to "extract" a proof of *P* from of proof of this implication.

For negation elimination to be valid, which is to say, to make Lean into a classical as opposed to constructive predicate logic, it suffices to accept an additional axiom into constructive logic: the so-called *law of the excluded middle*. This axiom allows us to assume, for *any* proposition, *P*, that *P ∨ ¬ P* is true: that there is a proof of either *P* or of ¬ *P*.

We now formalize the axiom of the excluded middle and then show that we can use it to prove the validity of negation elimination. Here's the axiom:

Try it in Lean online!

```
axiom em : ∀ (P : Prop), P ∨ ¬ P
```

Here's the proof.

Try it in Lean online!

```
axiom em : ∀ (P : Prop), P ∨ ¬ P
theorem neg_elim : ∀ (P : Prop), ¬ ¬ P → P :=
begin
assume P,
assume nnp,
cases (em P) with p np,
exact p,
-- next: false elim "by case analysis!"
-- false.elim (nnp np) would work, too
-- as would the "contradiction" tactic
cases (nnp np),
end
```

Accepting the axiom of the excluded middle turns Lean into another logic, converting it from a constructive predicate logic into a classical predicate logic. Most mathematicians work with classical predicate logic.

Constructive logic emerged from work on the foundations of constructive mathematics, a part of mathematics that rejects proofs of the existence of things without having such things in hand. Note that the constructive definition of a proof of *or P (¬P)* requires a proof of either*P* or of ¬ *P*, while in classical logic one accepts that*or P (¬P)* is true without having a proof of either.

P or Q

Given any two propositions, *P* and *Q*, one can form the larger proposition, *P ∨ Q*. The connective, *∨*, is short for the polymorphic logical type, *or P Q*, where *P*> and *Q* are propositions, terms of type Prop, thus also logical types.

We mean for *P ∨ Q* to assert that at least one of *P* or*Q* are true.

We formalize *∨* as a polymorphic proposition, *or*: that takes two propositions (logical types) as arguments, and that provides two proof constructors (proof building axioms). The first, called *inl*, takes a proof of *P* as an argument and constructs a proof of *or P Q*. The second, *inr*, takes a proof of *Q* and constructs a proof of *or P Q*.

Try it in Lean online!

```
inductive or (P Q : Prop) : Prop
| inl (p : P) : or
| inr (q : Q) : or
```

For purposes of our example, we assume that *P* and *Q* are arbitrary propositions, and that we have proof, *p* of *P*. We then construct a proof of *or P Q* from a proof of *P*.
Try it in Lean online!

```
axioms P Q : Prop
axioms (p : P)

example : P ∨ Q :=
or.inl p

example : P ∨ Q :=
begin
exact (or.inl p)
end
```

Here's another form of the same proof, where we introduce *P* and *Q* using *∀*.

Try it in Lean online!

```
example : ∀ (P Q : Prop), P → P ∨ Q :=
  λ (P Q : Prop) (p : P),
      or.inl p
```

There are two constructors (or.inl and or.inr) that can be used to construct a proof of *or P Q*, when using a proof, h, one must consider both *cases*: (or.inl p) and (or.inr q), where p and q are new names for the arguments that would have to have been given to the respective constructors for the proof to have been formed in the first place. We can then use these names and the objects to which they refer in subsequent reasoning.

Unlike and elimination, where we extract the fields, p and q, from h; with or elimination, we have to consider two cases separately. In the first case, the proof contains a proof of P. In the second case, it contains a proof of Q. So if we want to show that some situation, R, is implied by P or Q, we have to show that R is implied by P and that R is also implied by Q.

Our rule for using a proof of P or Q, or.elim, thus takes three arguments: h, as well as a proof that P implies R and a proof that Q implies R. The rule applied to all three such arguments forms a proof of R. We reproduce its implementation here so that you can see exactly how it works. In particular, be sure to note the case analysis, where we decide which form of proof we've got and destructure it accordingly.

Try it in Lean online!

```
def or_elim : ∀ {P Q R : Prop}, P ∨ Q → (P → R) → (Q → R) → R
| P Q R (or.inl p) pr qr := pr p
| P Q R (or.inr q) pr qr := qr q
```

The following theorem states the rule and then proves it using Lean's library-provided *or.intro* proof rule. Because there are two constructors, a given proof could have one of two forms. Given a proof, we use pattern matching to reason about each case. In the first case, or.inl must have been applied to a proof of P, and destructuring it using pattern matching gives us a name for that proof. We can then use that named object in constructing other proofs.

Try it in Lean online!

```
def or_cases : ∀ {P Q R : Prop}, P ∨ Q → (P → R) → (Q → R) → R :=
begin
assume P Q R,
assume h,
assume pr qr,
apply or_elim h _ _,
assume p,
exact (pr p),
assume q,
exact (qr q),
end
```

Try it in Lean online!

```
def or_cases : ∀ {P Q R : Prop}, P ∨ Q → (P → R) → (Q → R) → R :=
begin
assume P Q R,
assume h,
assume pr qr,
cases h with p q,
exact (pr p),
exact (qr q),
end
```

A great example is a proof that *or is commutative*. In other words, for any propositions, *P* and *Q*, *P ∨ Q* is equivalent to *Q ∨ P*. Stated formally, *∀ (P Q : Prop), P ∨ Q ↔ Q ∨ P*. This is a *theorem* in predicate logic, which we can now prove.

To construct a proof of a bi-implication, one must construct proofs of the implications in each direction. We will do this with two separate*lemmas*. Lemmas are proofs of lesser propositions, constructed to enable the construction of a proof of a major theorem.

Here, the major theorem is $\forall (P\ Q : Prop),\ P \vee Q \leftrightarrow Q \vee P$. The two lemmas will be proofs of the implications in both directions: first with the arrow from left to right, then from right to left. Once we have proofs of these lemmas we will combine them using *iff.intro* to prove the major theorem.

We present proofs of the left to right implication in several forms. We start with a proof term. We use lambda to introduce argments/assumptions that P and Q are propositions and that h is a proof of P or Q. We then need to show that Q or P is true, given that P or Q is. This requires that we show that Q or P is true no matter what constructor was used to construct h, to show that P or Q is true. So we do pattern matching on h. There are two cases. In the first case, inl must have been applied to a proof, let's call it p, of P; and in this case we can construct a proof of Q or P using inr p. In the second case, h must be of the form inr q. That is, P or Q is true because Q is. In this case we obtain a proof of q, again by destructuring h in this case to obtain a proof, we call it q, of Q; and we then use q to construct a proof of Q or P using inr.

Try it in Lean online!

```
lemma or_commutes_left_term: ∀ {P Q: Prop}, P ∨ Q → Q ∨ P :=
λ (P Q : Prop),
    λ (h : P ∨ Q),
        -- two cases to consider
        match h with
        | (or.inl p) := or.inr p
        | (or.inr q) := or.inl q
        end
```

Here is the equivalent proof as a proof script.

Try it in Lean online!

```
lemma or_commutes_left_script: ∀ {P Q: Prop}, P ∨ Q → Q ∨ P :=
begin
assume P Q,
assume h,
-- case analysis!
cases h with p q,
exact or.inr p,
exact or.inl q,
end
```

Here is the equivalent proof given by cases.

Try it in Lean online!

```
lemma or_commutes_left_cases : ∀ {P Q : Prop}, P ∨ Q → Q ∨ P
-- two case to consider
```

```
| P Q (or.inl p) := or.inr p
| P Q (or.inr q) := or.inl q
```

Here is our preferred (but not required) form. By naming P and Q before the colon we avoid having to introduce names for them using a forall, and the overall proof presentation is further simplified. Moreover, this form of the proof makes is very clear how we have formalized the proof of the lemma: as a polymorphic function! P and Q are basically type arguments. Given *any* such propositions, P and Q we have a function, that if given a proof of (P or Q) will and reduce to a proof of (Q or P).

Try it in Lean online!

```
lemma or_commutes_left {P Q : Prop} : P ∨ Q → Q ∨ P
-- two case to consider
| (or.inl p) := or.inr p
| (or.inr q) := or.inl q
```

The proof of the implication in the right to left direction is completely symmetric, so we only present the most concise form of the proof.

Try it in Lean online!

```
theorem or_commutes_right {P Q : Prop} : Q ∨ P → P ∨ Q
| (or.inl q) := or.inr q
| (or.inr p) := or.inl p
```

Now with proofs of both lemmas in hand, we case use these proofs as arguments to *iff.intro* to construct a proof of the theorem. QED.

Try it in Lean online!

```
theorem or_commutes (P Q : Prop) : P ∨ Q ↔ Q ∨ P :=
iff.intro or_commutes_left or_commutes_right
```

For All

If *P* is a type and *Q* is a proposition, then ∀ (p : P), Q is also a proposition: a "univereally quantified" proposition, also called a "universal generalization."

Try it in Lean online!

```
axioms (P : Type) (Q : Prop)
#check ∀ (p : P), Q     -- Prop
```

Such a proposition is intended to assert that the proposition Q (often formed by applying a predicate to a value of type P) is true for *any* value, (p : P). Note that if there are no values of P, that is, if P is an uninhabited type, such as false, then Q needs to be

true for no values at all for the overall proposition to be true. In other words, a proposition that is universally quantified over an empty set is trivially true.

The key idea behind a proof of such a proposition is that one assumes that one is given some arbitrary but specific value of the quantified type (just as any function is assumed to be given an arbitrary but specific value of its argument); and then, in that context, one shows that a proof of Q (a return value of type Q) can be produced. In formal terms, to prove ∀ (p : P), Q one shows that if given a value of type P, one can return a value of type Q.

If proving ∀ (p : P), Q sounds a lot like showing that there is a function of type P → Q, that's because that's exactly what is involved. In fact, P → Q is just an infix notation for ∀ (p : P), Q. Think about that.

So, if you are asked to construct a proof of ∀ (p : P), Q, treat it exactly as if you were being asked to prove P → Q: Assume there is a value/proof of P and in this context construct a value/proof of type Q. The result is a function of type P → Q. A proof of a "forall", just like a proof of an implication, P → Q, *is* a function of type P → Q!

Similarly, to use of proof of ∀ (p : P), Q, treat is exactly as if were a function of type P → Q, because it is exactly a function of this type! This is the elimination principle for universal generalizations.

Here's an example in which we first prove, the universal generalization, ∀ (n : ℕ), n = n, and then apply it to a particular value, 5, to obtain a proof of 5 = 5, in particular.

Try it in Lean online!

```
theorem eq_refl_nat : ∀ (n : ℕ), n = n :=
λ n, eq.refl n

#reduce eq_refl_nat 5

theorem eq_refl_nat' (n : ℕ) : n = n :=
eq.refl n
```

The third part of the example is an equivalent proof of the forall proposition. This time it is presented in a form that should make it entirely clear that the proof *is* a function. Compared with the first proof, all we have really done is to give the argument a name before the colon rather than using a ∀ binding after the colon.

In constructive logic, as should now be clear, proofs are programs. In the case of ∀ and →, proofs are functions. In the case of ∧, ↔, and ∃, proofs are ordered pairs. In the case of ∨, a proof is a "variant", which is to say a value in one of several possible forms (here "inl pf" or "inr pf").

There Exists

If *P* is a type and *Q* is a proposition, then ∃ (p : P), Q is also a proposition: an "existentially quantified" proposition.

Try it in Lean online!

```
axioms (P Q : Prop)
#check ∃ (p : P), Q   -- Prop: a proposition
```

Such a proposition is intended to assert that there is some value, (p : P), for which the proposition, Q, is true. Q does not have to be, but in practice usually is, formed by application of a predicate to p.

Here's an example where Q is not formed by application of a predicate: ∃ (p : P), true. In this case, any value (p : P) will make Q true because Q is true in any case, but for the overall proposition to be be, there does have to exist some such value, p, to be given as a "witness" to the truth of Q. In other words, P cannot be an empty/uninhabited type.

Assuming we've previously defined a predicate, even : ℕ → Prop, here's an example where Q is formed by application of a predicate to the value bound by the ∃: ∃ (n : ℕ), even n. This proposition asserts that there is some value, n, a natural number, that satisfies the predicate, even n. In plain English, "there is some even number."

In the constructive logic of Lean, a proof of such a proposition is basically a pair, (w, pf), where w is a specific value of type P and where pf is a proof of Q. In other words, to construct a proof of ∃ (p : P), Q, you have to give two values: some value w of type P, and a proof that Q is true for that specific value w. This shows that there does exist some value of type P that satisfies Q. This is the introduction rule for exists.

The formalization of exists in Lean is a little bit complicated, but it is based on an easy-to-understand polymorphic proposition, Exists. This logical type takes one type argument, α: the type for which a witness must be given; and it takes one additional argument, p, a predicate on values of that type. In our terminology, P is α, and Q is formed by applying p to some value, w, of this type, P. We omit further details of the exact definition of exists in Lean.

Try it in Lean online!

```
inductive Exists {α : Type} (p : α → Prop) : Prop
| intro (w : α) (h : p w) : Exists
```

The key to understanding it, however, is to see that, once again, the polymorphic type defines one proof constructor, the introduction rule for exists, which precisely requires two arguments: (w : α), a value that serves as a "witness", and (h : p w), a proof that w satisfies p, i.e., that the proposition, (p w), is true.

Here are three examples of statements and proofs of existentially quantified propositions introduced above.

Try it in Lean online!

```
/-
Even though Q is always true, this
proposition cannot be proved because
there does not exist any value, f, at
all.
-/
example : ∃ (f : false), true :=
exists.intro _ _
```

Try it in Lean online!

```
/-
Even though there obviously exists
some natural number, no choice of n
will ever make Q (false in this case)
true, so this proposition can't be
proved.
-/
example : ∃ (n : ℕ), false :=
exists.intro 5 _
```

Try it in Lean online!

```
-- We start by defining an even predicate
def even (n : ℕ): Prop := n % 2 = 0

/-
Now we can assert and prove that there
exists some even number. In this case,
we pick 4 as a witness, but any even
number will do. Note that the proof
is by application of the introduction
rule for exists two values: a witness,
4, and a proof that that particular
witness is even. Re-do this example
with a proof script, giving only 4
as an argument to exists.intro, then
look to see what is the remaining goal.
It will then be clear that (eq.refl 0)
is a proof of a proposition "about 4"
(that it is even).
-/
example : exists (n : ℕ), even n :=
exists.intro 4 (eq.refl 0)
```

The next two examples involve multiple existentially quantified variables. To prove them, introduce each quantified variable in turn, and then provides a proof of the predicate over the several variables. The first proof is given as a tactic script, and the second as an exact proof term. In English, one would simply state that "We take the

triple, (3,4,5) as a witness, and by simple algebra and the reflexive property of equality show that this triple satisfies the predicate"

Try it in Lean online!

```
example : exists (a b c : ℕ), a^2 + b^2 = c^2 :=
begin
apply exists.intro 3,
apply exists.intro 4,
apply exists.intro 5,
exact (eq.refl 25),
end
```

Try it in Lean online!

```
example : exists (a b c : ℕ), a^2 + b^2 = c^2 :=
exists.intro 3 (
    exists.intro 4 (
        exists.intro 5 (eq.refl 25)
    )
)
```

The elimination rule for exists simple reflects the fact that a proof of ∃ (p : P), Q is pair, (w, pf). The elimination rule uses pattern matching to destructures such a pair, binding names to its elements (such as "w" and "pf") so that these names can be used in building a new proof. The first proof in the following example makes it clear that the elimination is just pattern matching. There's only one constructor, so there is only one case to consider.

Here's the example. Informally, it states that if there is a Ball that is both heavy and round, then there is a ball that is heavy. This is an implication, so we will first assume the premise. Doing this gives us a proof of an existentially quantified proposition to use.

We use it in constructing proofs in two styles. The first proof is in the form of an exact proof term that uses explicit pattern matching. The second proof is in the form of a tactic script. Correlating these two proofs should make it clear that all that the cases tactic is doing is pattern matching on the assumed proof, h. The cases tactic is a sort of all-purpose elimination operation!

Try it in Lean online!

```
axiom Ball : Type
axioms (Heavy : Ball → Prop) (Round : Ball → Prop)

example : (∃ (b : Ball), Round b ∧ Heavy b) →
         (∃ (b : Ball), Round b) :=
λ h,
match h with
| exists.intro w pf := exists.intro w pf.left
end

example : (∃ (b : Ball), Round b ∧ Heavy b) →
         (∃ (b : Ball), Round b) :=
```

```
begin
assume h,
cases h with w pf,
apply exists.intro w pf.left
end
```