3110 Coq Tactics Cheatsheet

When proving theorems in Coq, knowing what tactics you have at your disposal is vital. In fact, sometimes it's impossible to complete a proof if you don't know the right tactic to use!

We provide this tactics cheatsheet as a reference. It contains all the tactics used in the lecture slides, notes, and lab solutions.

Quick Overview

Here is a brief overview of the tactics that we've covered. Click on any of the links for more details about how to use them.

Solving simple goals:

- assumption: Solves the goal if it is already assumed in the context.
- reflexivity: Solves the goal if it is a trivial equality.
- trivial: Solves a variety of easy goals.
- auto: Solves a greater variety of easy goals.
- discriminate: Solves the goal if it is a trivial inequality and solves any goal if the context contains a false equality.
- exact: Solves a goal if you know the exact proof term that proves the goal.
- contradiction: Solves any goal if the context contains `False` or contradictory hypotheses.

Transforming goals:

- intros / intro: Introduces variables appearing with `forall` as well as the premises (left-hand side) of implications.
- simpl: Simplifies the goal or hypotheses in the context.
- unfold: Unfolds the definitions of terms.
- apply: Uses implications to transform goals and hypotheses.
- rewrite: Replaces a term with an equivalent term if the equivalence of the terms has already been proven.
- inversion: Deduces equalities that must be true given an equality between two constructors.
- left / right: Replaces a goal consisting of a disjunction `P \/ Q` with just `P` or `Q`.
- replace: Replace a term with a equivalent term and generates a subgoal to prove that the equality holds.

Breaking apart goals and hypotheses:

- split: Replaces a goal consisting of a conjunction `P /\ Q` with two subgoals `P` and `Q`.
- destruct (and/or): Replaces a hypothesis `P /\ Q` with two hypotheses `P` and `Q`. Alternatively, if the hypothesis is a disjunction `P \/ Q`, generates two subgoals, one where `P` holds and one where `Q` holds.
- destruct (case analysis): Generates a subgoal for every constructor of an inductive type.
- induction: Generates a subgoal for every constructor of an inductive type and provides an induction hypothesis for recursively defined constructors.

Solving specific types of goals:

- ring: Solves goals consisting of addition and multiplication operations.
- tauto: Solves goals consisting of tautologies that hold in constructive logic.
- field: Solves goals consisting of addition, subtraction (the additive inverse), multiplication, and division (the multiplicative inverse).

Tacticals (tactics that act on other tactics):

- ; (semicolon): Applies the tactic on the right to all subgoals produced by the tactic on the left.
- try: Attempts to apply the given tactic but does not fail even if the given tactic fails.
- || (or): Tries to apply the tactic on the left; if that fails, tries to apply the tactic on the right.
- all:: Applies the given tactic to all remaining subgoals.
- repeat: Applies the given tactic repeatedly until it fails.

Solving simple goals

The following tactics prove simple goals. Generally, your aim when writing Coq proofs is to transform your goal until it can be solved using one of these tactics.

assumption

If the goal is already in your context, you can use the `assumption` tactic to immediately prove the goal.

Example:

```
Theorem p_implies_p : forall P : Prop,
  P -> P.
Proof.
  intros P P_holds.
```

```
1 subgoal
P : Prop
P_holds : P
-------------------(1/1)
P
```

After introducing the hypothesis `P_holds` (which says that `P` is true) into the context, we can use `assumption` to complete the proof.

```
Theorem p_implies_p : forall P : Prop,
  P -> P.
Proof.
  intros P P_holds.
  assumption.
```

```
No more subgoals.
```

reflexivity

If the goal contains an equality that looks obviously true, the `reflexivity` tactic can finish off the proof, doing some basic simplification if needed.

Example:

```
Theorem forty_two : 41 + 1 = 42.
Proof.
```

```
1 subgoal
------------------(1/1)
41 + 1 = 42
```

Both the left and right sides of the equality in the goal are clearly equal (after simplification), so we use `reflexivity`.

```
Theorem forty_two : 41 + 1 = 42.
Proof.
  reflexivity.
```

```
No more subgoals.
```

trivial

The `trivial` tactic can solve a variety of simple goals. It introduces variables and hypotheses into the context and then tries to use various other tactics under the hood to solve the goal.

Any goal that can be solved with `assumption` or `reflexivity` can also be solved using `trivial`.

Unlike most of the other tactics in this section, `trivial` will not fail even if it cannot solve the goal.

Example:

```
Theorem p_implies_p : forall P : Prop,
  P -> P.
Proof.
```

```
1 subgoal
------------------(1/1)
P -> P
```

Previously, we proved this theorem using `intros` and `assumption`; however, `trivial` can actually take care of both those steps in one fell swoop.

```
Theorem p_implies_p : forall P : Prop,
  P -> P.
Proof.
  trivial.
```

```
No more subgoals.
```

auto

The `auto` tactic is a more advanced version of `trivial` that performs a recursive proof search.

Any goal that can be solved with `trivial` can also be solved using `auto`.

Like `trivial`, `auto` never fails even if it cannot do anything.

Example:

```
Theorem modus_tollens: forall (P Q : Prop),
  (P -> Q) -> ~Q -> ~P.
Proof.
```

```
1 subgoal
-------------------(1/1)
forall P Q : Prop, (P -> Q) -> ~ Q -> ~ P
```

This proof is too complicated for `trivial` to handle on its own, but it can be solved with `auto`!

```
Theorem modus_tollens: forall (P Q : Prop),
  (P -> Q) -> ~Q -> ~P.
Proof.
  auto.
```

```
No more subgoals.
```

discriminate

The `discriminate` tactic proves that different constructors of an inductive type cannot be equal. In other words, if the goal is an inequality consisting of two different constructors, `discriminate` will solve the goal.

`discriminate` also has another use: if the context contains a equality between two different constructors (i.e. a false assumption), you can use `discriminate` to prove any goal.

Example 1:

```
Inductive element :=
| grass : element
| fire : element
| water : element.

Theorem fire_is_not_water : fire <> water.
Proof.
```

```
1 subgoal
-------------------(1/1)
fire <> water
```

You may be surprised to learn that `auto` cannot solve this simple goal! However, `discriminate` takes care of this proof easily.

```
Inductive element :=
| grass : element
| fire : element
| water : element.

Theorem fire_is_not_water : fire <> water.
Proof.
  discriminate.
```

```
No more subgoals.
```

Example 2:

```
Theorem false_implies_anything : forall P : Prop,
  0 = 1 -> P.
Proof.
  intros P zero_equals_one.
```

```
1 subgoal
P : Prop
zero_equals_one : 0 = 1
-------------------(1/1)
P
```

Recall that the natural numbers in Coq are defined as an inductive type with constructors `O` (zero) and `S` (successor of). The constructors on both sides of the false equality `0 = 1` are different, so we can use `discriminate` to prove our goal that any proposition `P` holds.

```
Theorem false_implies_anything : forall P : Prop,
  0 = 1 -> P.
Proof.
  intros P zero_equals_one.
  discriminate.
```

```
No more subgoals.
```

exact

If you know the exact proof term that proves the goal, you can provide it directly using the `exact` tactic.

Example:

```
Theorem everything : 42 = 42.
Proof.
```

```
1 subgoal
-------------------(1/1)
42 = 42
```

Suppose we know that `eq_refl 42` is a term with the type `42 = 42`. Then, we prove that there exists a value that inhabits this type by supplying the term directly using `exact`, which proves the theorem.

(We could have also used `reflexivity` or other tactics to prove this goal.)

```
Theorem everything : 42 = 42.
Proof.
  exact (eq_refl 42).
```

```
No more subgoals.
```

contradiction

If there is a hypothesis that is equivalent to `False` or two contradictory hypotheses in the context, you can use the `contradiction` tactic to prove any goal.

Example:

```
Theorem law_of_contradiction : forall (P Q :
Prop),
  P /\ ~P -> Q.
Proof.
  intros P Q P_and_not_P.
  destruct P_and_not_P as [P_holds not_P].
```

```
1 subgoal
P, Q : Prop
P_holds : P
not_P : ~ P
--------------------(1/1)
Q
```

After destructing the hypothesis `P /\ ~P`, we obtain two hypotheses `P` and `~P` that contradict each other, so we use `contradiction` to complete the proof.

```
Theorem law_of_contradiction : forall (P Q :
Prop),
  P /\ ~P -> Q.
Proof.
  intros P Q P_and_not_P.
  destruct P_and_not_P as [P_holds not_P].
  contradiction.
```

```
No more subgoals.
```

Transforming goals

While proving a theorem, you will typically need to transform your goal to introduce assumptions into the context, simplify the goal, make use of assumptions, and so on. The following tactics allow you to make progress toward solving a goal.

intros / intro

If there are universally quantified variables in the goal (i.e. `forall`), you can introduce those variables into the context using the `intros` tactic. You can also use `intros` to introduce all propositions on the left side of an implication as assumptions.

If `intros` is used by itself, Coq will introduce all the variables and hypotheses that it can, and it will assign names to them automatically. You can provide your own names (or introduce fewer things) by supplying those names in order. See Example 2.

`intros` also has a sister tactic `intro` that introduces just one thing.

`intros` used by itself will never fail, even if there's nothing to introduce. If you supply some names to `intros`, however, it will fail if a name is already in use or if there's not enough stuff left to introduce.

Example 1:

```
Theorem modus_tollens : forall (P Q : Prop),
  (P -> Q) -> ~Q -> ~P.
Proof.
```

```
1 subgoal
-------------------(1/1)
forall P Q : Prop, (P -> Q) -> ~ Q -> ~ P
```

We can introduce the two variables `P` and `Q`, as well as the two hypotheses `(P -> Q)` and `~Q` using `intros`.

```
Theorem modus_tollens : forall (P Q : Prop),
  (P -> Q) -> ~Q -> ~P.
Proof.
  intros.
```

```
1 subgoal
P, Q : Prop
H : P -> Q
H0 : ~ Q
-------------------(1/1)
~ P
```

Example 2:

The names that Coq chose for the hypotheses `H` and `H0` aren't very descriptive. We can provide more descriptive names instead. Note that we also have to give names to the two variables after the `forall` because `intros` introduces things in order.

```
Theorem modus_tollens : forall (P Q : Prop),
  (P -> Q) -> ~Q -> ~P.
Proof.
  intros P Q P_implies_Q not_Q.
```

```
1 subgoal
P, Q : Prop
P_implies_Q : P -> Q
not_Q : ~ Q
-------------------(1/1)
~ P
```

simpl

The `simpl` tactic reduces complex terms to simpler forms. You'll find that it's not always necessary to use `simpl` because other tactics (e.g. `discriminate`) can do the simplification themselves, but it's often helpful to try `simpl` to help you figure out what you, as the writer of the proof, should do next.

You can also use `simpl` on a hypothesis in the context with the syntax `simpl in <hypothesis>`.

`simpl` will never fail, even if no simplification can be done.

Example:

```
Theorem switch_to_honors : 2110 + 2 = 2112.
Proof.
```

```
1 subgoal
-------------------(1/1)
2110 + 2 = 2112
```

Let's simplify that goal with `simpl`.

```
Theorem switch_to_honors : 2110 + 2 = 2112.
Proof.
  simpl.
```

```
1 subgoal
-------------------(1/1)
2112 = 2112
```

unfold

The `unfold` tactic replaces a defined term in the goal with its definition.

You can also use `unfold` on a hypothesis in the context with the syntax `unfold <term> in <hypothesis>`.

Example 1:

```
Definition plus_two (x : nat) : nat :=
  x + 2.

Theorem switch_to_honors_again :
  plus_two 2110 = 2112.
Proof.
```

```
1 subgoal
-------------------(1/1)
plus_two 2110 = 2112
```

This time, nothing happens if we try `simpl`. However, we can `unfold` and transform the goal into something that we can then simplify.

```
Theorem switch_to_honors_again :
  plus_two 2110 = 2112.
Proof.
  unfold plus_two.
```

```
1 subgoal
-------------------(1/1)
2110 + 2 = 2112
```

Example 2:

```
Theorem demorgan : forall (P Q : Prop),
  ~(P \/ Q) -> ~P /\ ~Q.
Proof.
  intros P Q not_P_or_Q.
  unfold not.
```

```
1 subgoal
P, Q : Prop
not_P_or_Q : ~ (P \/ Q)
-------------------(1/1)
(P -> False) /\ (Q -> False)
```

We'd like to unfold the `~(P \/ Q)` in our context as well, so we use `unfold..in..`.

(In thie case, we could have also applied `unfold` before `intros` to unfold all the negations at once.)

```
Theorem demorgan : forall (P Q : Prop),
  ~(P \/ Q) -> ~P /\ ~Q.
Proof.
  intros P Q not_P_or_Q.
  unfold not.
  unfold not in not_P_or_Q.
```

```
1 subgoal
P, Q : Prop
not_P_or_Q : P \/ Q -> False
-------------------(1/1)
(P -> False) /\ (Q -> False)
```

apply

The `apply` tactic has a variety of uses.

If your goal is some proposition `B` and you know that `A -> B`, then in order to prove that `B` holds, it suffices to show `A` holds. `apply` uses this reasoning to transform the goal from `B` to `A`. See Example 1.

`apply` can also be used on hypotheses. If you have some hypothesis that states that `A` holds, as well as another hypothesis `A -> B`, you can use `apply` to transform the first hypothesis into `B`. The syntax is `apply <term> in <hypothesis>` or `apply <term> in <hypothesis> as <new-hypothesis>`. See Example 2.

You can even apply previously proven theorems. See Example 3.

Example 1:

```
Theorem modus_ponens : forall (P Q : Prop),
  (P -> Q) -> P -> Q.
Proof.
  intros P Q P_implies_Q P_holds.
```

```
1 subgoal
P, Q : Prop
P_implies_Q : P -> Q
P_holds : P
-------------------(1/1)
Q
```

Since we know that `P -> Q`, proving that `P` holds would also prove that `Q` holds. Therefore, we use `apply` to transform our goal.

```
Theorem modus_ponens : forall (P Q : Prop),
  (P -> Q) -> P -> Q.
Proof.
  intros P Q P_implies_Q P_holds.
  apply P_implies_Q.
```

```
1 subgoal
P, Q : Prop
P_implies_Q : P -> Q
P_holds : P
-------------------(1/1)
P
```

Example 2:

Alternatively, we notice that `P` holds in our context, and because we know that `P -> Q`, we can apply that implication to our hypothesis that `P` holds to transform it.

```
Theorem modus_ponens : forall (P Q : Prop),
  (P -> Q) -> P -> Q.
Proof.
  intros P Q P_implies_Q P_holds.
  apply P_implies_Q in P_holds.
```

```
1 subgoal
P, Q : Prop
P_implies_Q : P -> Q
P_holds : Q
-------------------(1/1)
P
```

Note that this *replaces* our previous hypothesis (and now its name is no longer very applicable)! To prevent this, we can give our new hypothesis its own name using the `apply..in..as..` syntax.

```
Theorem modus_ponens : forall (P Q : Prop),
  (P -> Q) -> P -> Q.
Proof.
  intros P Q P_implies_Q P_holds.
  apply P_implies_Q in P_holds as Q_holds.
```

```
1 subgoal
P, Q : Prop
P_implies_Q : P -> Q
P_holds : P
Q_holds : Q
-------------------(1/1)
P
```

Example 3:

```
Lemma modus_ponens'' : forall (P Q : Prop),
  P -> (P -> Q) -> Q.
Proof.
  auto.
Qed.

Theorem double_negation : forall (P : Prop),
  P -> ~~P.
Proof.
  unfold not. intro P.
```

```
1 subgoal
P : Prop
-------------------(1/1)
P -> (P -> False) -> False
```

We notice that our goal is just an instance of `P -> (P -> Q) -> Q`, which we already proved is true. Therefore, we can use `apply` to apply our lemma, which finishes the proof.

```
Lemma modus_ponens'' : forall (P Q : Prop),
  P -> (P -> Q) -> Q.
Proof.
  auto.
Qed.

Theorem double_negation : forall (P : Prop),
  P -> ~~P.
Proof.
  unfold not. intro P.
  apply modus_ponens''.
```

```
No more subgoals.
```

rewrite

Given some known equality `a = b`, the `rewrite` tactic lets you replace `a` with `b` or vice versa in a goal or hypothesis

The syntax is `rewrite -> <equality>` to replace `a` with `b` in the goal or `rewrite <- <equality>` to replace `b` with `a`. Note that `rewrite <equality>` is identical to `rewrite -> <equality>`.

You can also rewrite terms in hypotheses with the `rewrite..in..` syntax.

Example:

```
Theorem add_comm : forall (x y : nat),
  x + y = y + x.
Proof.
  intros. induction x.
  - trivial.
  - simpl.
```

```
1 subgoal
x, y : nat
IHx : x + y = y + x
------------------(1/1)
S (x + y) = y + S x
```

We can try using `auto` to see if Coq can figure out the rest of the proof for us, but it can't because it doesn't know that addition is commutative (that's what we're trying to prove!).

However, we can apply our inductive hypothesis `x + y = y + x` by rewriting the `x + y` in the goal as `y + x` using `rewrite`:

```
Theorem add_comm : forall (x y : nat),
  x + y = y + x.
Proof.
  intros. induction x.
  - trivial.
  - simpl. rewrite -> IHx.
```

```
1 subgoal
x, y : nat
IHx : x + y = y + x
------------------(1/1)
S (y + x) = y + S x
```

Now you can finish the proof by simply using `trivial` or `auto`.

inversion

Suppose you have a hypothesis `S m = S n`, where `m` and `n` are `nats`. The `inversion` tactic allows you to conclude that `m = n`. In general, if you have a hypothesis that states an equality between two constructors and the constructors are the same, `inversion` helps you figure out that all the arguments to those constructors must be equal as well, and it tries to rewrite the goal using that information.

Example:

```
Theorem succ_eq_implies_eq : forall (x y : nat),
  S x = S y -> x = y.
Proof.
  intros x y succ_eq.
```

```
1 subgoal
x, y : nat
succ_eq : S x = S y
------------------(1/1)
x = y
```

Since `S x = S y`, we use `inversion` to extract a new hypothesis that states `x = y`. `inversion` actually goes one step further and rewrites the goal using that equality.

```
Theorem succ_eq_implies_eq : forall (x y : nat),
  S x = S y -> x = y.
Proof.
  intros x y succ_eq.
  inversion succ_eq.
```

```
1 subgoal
x, y : nat
succ_eq : S x = S y
H0 : x = y
------------------(1/1)
y = y
```

left / right

If the goal is a disjunction `A \/ B`, the `left` tactic replaces the goal with the left side of the disjunction `A`, and the `right` tactic replaces the goal with the right side `B`.

Example 1:

```
Theorem or_left : forall (P Q : Prop),
  P -> P \/ Q.
Proof.
  intros P Q P_holds.
```

```
1 subgoal
P, Q : Prop
P_holds : P
------------------(1/1)
P \/ Q
```

Since we know that `P` holds, it makes sense to change the goal to the left side of the disjunction using `left`.

```
Theorem or_left : forall (P Q : Prop),
  P -> P \/ Q.
Proof.
  intros P Q P_holds.
  left.
```

```
1 subgoal
P, Q : Prop
P_holds : P
------------------(1/1)
P
```

Example 2:

```
Theorem or_right : forall (P Q : Prop),
  Q -> P \/ Q.
Proof.
  intros P Q Q_holds.
```

```
1 subgoal
P, Q : Prop
Q_holds : Q
------------------(1/1)
P \/ Q
```

This time, we know that `Q` holds, so we replace the goal with its right side using `right`.

```
Theorem or_right : forall (P Q : Prop),
  Q -> P \/ Q.
Proof.
  intros P Q Q_holds.
  right.
```

```
1 subgoal
P, Q : Prop
Q_holds : Q
------------------(1/1)
Q
```

replace

The `replace` tactic allows you to replace a term in the goal with another term and produces a new subgoal that asks you to prove that those two terms are equal. The syntax is `replace <term> with <term>`.

Example:

```
Theorem one_x_one : forall (x : nat),
  1 + x + 1 = 2 + x.
Proof.
  intro. simpl.
```

```
1 subgoal
x : nat
------------------(1/1)
S (x + 1) = S (S x)
```

We believe that `x + 1` and `S x` are equal, so we can use `replace` to assert that this equality is true and then prove it later.

```
Theorem one_x_one : forall (x : nat),
  1 + x + 1 = 2 + x.
Proof.
  intro. simpl.
  replace (x + 1) with (S x).
```

```
2 subgoals
x : nat
------------------(1/2)
S (S x) = S (S x)
------------------(2/2)
S x = x + 1
```

Breaking apart goals and hypotheses

The following tactics break apart goals (or hypotheses) into several simpler subgoals (or hypotheses).

split

If the goal is a conjunction `A /\ B`, the `split` tactic replaces the goal with two subgoals `A` and `B`.

Example:

```
Theorem implies_and : forall (P Q R : Prop),
  P -> (P -> Q) -> (P -> R) -> (Q /\ R).
Proof.
  intros P Q R P_holds.
  intros P_implies_Q P_implies_R.
```

```
1 subgoal
P, Q, R : Prop
P_holds : P
P_implies_Q : P -> Q
P_implies_R : P -> R
-------------------(1/1)
Q /\ R
```

In order to make progress in the proof, we use `split` to break up `Q /\ R` into two subgoals.

```
Theorem implies_and : forall (P Q R : Prop),
  P -> (P -> Q) -> (P -> R) -> (Q /\ R).
Proof.
  intros P Q R P_holds.
  intros P_implies_Q P_implies_R.
  split.
```

```
2 subgoals
P, Q, R : Prop
P_holds : P
P_implies_Q : P -> Q
P_implies_R : P -> R
-------------------(1/2)
Q
-------------------(2/2)
R
```

destruct (and / or)

If there is a hypothesis containing a conjunction or a disjunction in the context, you can use the `destruct` tactic to break them apart.

A hypothesis `A /\ B` means that both `A` and `B` hold, so it can be destructed into two new hypotheses `A` and `B`. You can also use the `destruct..as [...]` syntax to give your own name to these new hypotheses. See Example 1.

On the other hand, a hypothesis `A \/ B` means that at least one of `A` and `B` holds, so in order to make use of this hypothesis, you must prove that the goal holds when `A` is true (and `B` may not be) and when `B` is true (and `A` may not be). You can also use the `destruct..as [... | ...]` syntax to provide your own names to the hypotheses that are generated (note the presence of the the vertical bar). See Example 2.

Example 1:

```
Theorem and_left : forall (P Q : Prop),
  (P /\ Q) -> P.
Proof.
  intros P Q P_and_Q.
```

```
1 subgoal
P, Q : Prop
P_and_Q : P /\ Q
-------------------(1/1)
P
```

Since there's a conjunction `P /\ Q` in our context, using `destruct` on it will give us both `P` and `Q` as separate hypotheses.

```
Theorem and_left : forall (P Q : Prop),
  (P /\ Q) -> P.
Proof.
  intros P Q P_and_Q.
  destruct P_and_Q.
```

```
1 subgoal
P, Q : Prop
H : P
H0 : Q
------------------(1/1)
P
```

The names that Coq chose for the new hypotheses aren't very descriptive, so let's provide our own.

```
Theorem and_left : forall (P Q : Prop),
  (P /\ Q) -> P.
Proof.
  intros P Q P_and_Q.
  destruct P_and_Q as [P_holds Q_holds].
```

```
1 subgoal
P, Q : Prop
P_holds : P
Q_holds : Q
------------------(1/1)
P
```

Example 2:

```
Theorem or_comm : forall (P Q : Prop),
  P \/ Q -> Q \/ P.
Proof.
  intros P Q P_or_Q.
```

```
1 subgoal
P, Q : Prop
P_or_Q : P \/ Q
------------------(1/1)
Q \/ P
```

We can `destruct` the hypothesis `P \/ Q` to replace our current goal with two new subgoals `P \/ Q` with different contexts: one in which `P` holds and one in which `Q` holds.

```
Theorem or_comm : forall (P Q : Prop),
  P \/ Q -> Q \/ P.
Proof.
  intros P Q P_or_Q.
  destruct P_or_Q as [P_holds | Q_holds].
```

```
2 subgoals
P, Q : Prop
P_holds : P
------------------(1/2)
Q \/ P
------------------(2/2)
Q \/ P
```

After we've proven the first subgoal, we observe that, in the context for the second subgoal, we have the hypothesis that `Q` holds instead.

```
Theorem or_comm : forall (P Q : Prop),
  P \/ Q -> Q \/ P.
Proof.
  intros P Q P_or_Q.
  destruct P_or_Q as [P_holds | Q_holds].
  - right. assumption.
  -
```

```
1 subgoal
P, Q : Prop
Q_holds : Q
-------------------(1/1)
Q \/ P
```

destruct (case analysis)

The `destruct` tactic can also be used for more general case analysis by destructing on a term or variable whose type is an inductive type.

Example:

```
Inductive element :=
| grass : element
| fire : element
| water : element.

Definition weakness (e : element) : element :
=
  match e with
  | grass => fire
  | fire => water
  | water => grass
  end.

Theorem never_weak_to_self : forall (e : elem
ent),
  weakness e <> e.
Proof.
```

```
1 subgoal
-------------------(1/1)
forall e : element, weakness e <> e
```

In order to proceed with this proof, we need to prove that it holds for each constructor of `element` case-by-case, so we use `destruct`.

```
Theorem never_weak_to_self : forall (e : elem
ent),
  weakness e <> e.
Proof.
  destruct e.
```

```
3 subgoals
-------------------(1/3)
weakness grass <> grass
-------------------(2/3)
weakness fire <> fire
-------------------(3/3)
weakness water <> water
```

induction

Using the `induction` tactic is the same as using the `destruct` tactic, except that it also introduces induction hypotheses as appropriate.

Once again, you can use the `induction..as [...]` syntax to give names to the terms and hypotheses produced in the different cases.

See lecture, notes, and lab 22 for more on induction.

Example:

```
Theorem n_plus_n : forall (n : nat),
  n + n = n * 2.
Proof.
  induction n as [| x IH].
```

```
2 subgoals
-------------------(1/2)
0 + 0 = 0 * 2
-------------------(2/2)
S x + S x = S x * 2
```

The base case `0` doesn't produce anything new, so we don't need to provide any names there. The inductive case `S x` produces a new term `x` and a new hypothesis, so we give those names. The vertical bar separates the two cases.

After proving the base case, we move on to the inductive case. Hey, Coq came up with the correct induction hypothesis for us. Thanks, Coq!

```
Theorem n_plus_n : forall (n : nat),
  n + n = n * 2.
Proof.
  induction n as [| x IH].
  - reflexivity.
  - simpl.
```

```
1 subgoal
x : nat
IH : x + x = x * 2
-------------------(1/1)
S (x + S x) = S (S (x * 2))
```

From here, we can make use of the induction hypothesis with `rewrite` and then apply `auto` to knock out the rest of the proof.

```
Theorem n_plus_n : forall (n : nat),
  n + n = n * 2.
Proof.
  induction n as [| x IH].
  - reflexivity.
  - simpl. rewrite <- IH. auto.
```

```
No more subgoals.
```

Solving specific types of goals

The tactics in this section are automated tactics that are specialized for solving certain types of goals.

ring

The `ring` tactic can solve any goal that contains only addition and multiplication operations.

You must first use the command `Require Import Arith.` in order to use `ring`.

Example:

```
Require Import Arith.

Theorem foil : forall a b c d,
  (a + b) * (c + d) = a*c + b*c + a*d + b*d.
Proof.
  intros. ring.
```

```
No more subgoals.
```

It would be pretty painful to prove this using simpler tactics, but fortunately `ring` is here to save the day.

tauto

The `tauto` tactic can solve any goal that's a tautology (in constructive logic). A tautology is a logical formula that's always true, regardless of the values of the variables in it.

Example:

```
Theorem demorgan : forall (P Q : Prop),
  ~(P \/ Q) -> ~P /\ ~Q.
Proof.
  tauto.
```

```
No more subgoals.
```

DeMorgan's law is a tautology, so it can be proven by applying `tauto`.

field

The `field` tactic can solve any goal that contains addition, subtraction (the additive inverse), multiplication, and division (the multiplicative inverse).

Note that `field` cannot be used on the natural numbers or integers, because integer division is not the inverse of multiplication (e.g. `(1 / 2) * 2` does not equal 1).

You must first use the command `Require Import Field.` in order to use `field`.

See lecture notes 22 (http://www.cs.cornell.edu/courses/cs3110/2017fa/l/22-coq-induction/notes.v) for more information on `field`.

Tacticals

The following *tacticals* are "higher-order tactics" that operate on tactics.

; (semicolon)

The `;` tactical applies the tactic on the right side of the semicolon to all the subgoals produced by tactic on the left side.

Example:

```
Theorem and_comm : forall (P Q : Prop),
  P /\ Q -> Q /\ P.
Proof.
  intros P Q P_and_Q.
  destruct P_and_Q.
  split.
  - assumption.
  - assumption.
Qed.
```

The two subgoals generated by `split` were solved using the same tactic. We can use `;` to make the code more concise.

```
Theorem and_comm : forall (P Q : Prop),
  P /\ Q -> Q /\ P.
Proof.
  intros P Q P_and_Q.
  destruct P_and_Q.
  split; assumption.
Qed.
```

try

Many tactics will fail if they are not applicable. The `try` tactical lets you attempt to use a tactic and allows the tactic to go through even if it fails. This can be particularly useful when chaining tactics together using `;`.

Example:

```
Inductive element :=
| grass : element
| fire : element
| water : element.

Definition weakness (e : element) : element :
=
  match e with
  | grass => fire
  | fire => water
  | water => grass
  end.

Theorem fire_weak_implies_grass :
  forall (e : element),
    weakness e = fire -> e = grass.
Proof.
  destruct e.
```

```
3 subgoals
-------------------(1/3)
weakness grass = fire -> grass = grass
-------------------(2/3)
weakness fire = fire -> fire = grass
-------------------(3/3)
weakness water = fire -> water = grass
```

We'd like to use `discriminate` to take care of the second and third subgoals, but we can't simply write `destruct e; discriminate.` because `discriminate` will fail when Coq tries to apply it to the first subgoal. This is where the `try` tactic comes in handy.

```
Theorem fire_weak_implies_grass :
  forall (e : element),
    weakness e = fire -> e = grass.
Proof.
  destruct e; try discriminate.
```

```
1 subgoal
-------------------(1/1)
weakness grass = fire -> grass = grass
```

|| (or)

The `||` tactical first tries the tactic on the left side; if it fails, then it applies the tactic on the right side.

Example:

```
Theorem fire_weak_implies_grass :
  forall (e : element),
    weakness e = fire -> e = grass.
Proof.
  destruct e; try discriminate.
```

```
1 subgoal
-------------------(1/1)
weakness grass = fire -> grass = grass
```

Let's use this theorem from the last section again. `discriminate` took care of the other two subgoals, and we know that `trivial` can solve this one. In other words, we apply either `discriminate` or `trivial` to the subgoals generated by `destruct e`, so we can use `||` to shorten the proof.

```
Theorem fire_weak_implies_grass :
  forall (e : element),
    weakness e = fire -> e = grass.
Proof.
  destruct e; discriminate || trivial.
```

```
No more subgoals.
```

all:

The `all:` tactical applies a tactic to all the remaining subgoals in the proof.

Example:

```
Theorem fire_weak_implies_grass :
  forall (e : element),
    weakness e = fire -> e = grass.
Proof.
  destruct e.
  all: discriminate || trivial.
```

```
1 subgoal
-------------------(1/1)
weakness grass = fire -> grass = grass
```

An alternative proof for the previous theorem using `all:`.

repeat

The `repeat` tactical repeatedly applies a tactic until it fails.

Note that `repeat` will never fail, even if it applies the given tactic zero times.

Example:

```
Require Import Arith.

Theorem add_assoc_4 : forall (a b c d : nat),
  (a + b + c) + d = a + (b + c + d).
Proof.
  intros.
```

```
1 subgoal
a, b, c, d : nat
-------------------(1/1)
a + b + c + d = a + (b + c + d)
```

Coq provides a theorem `Nat.add_assoc : forall n m p : nat, n + (m + p) = n + m + p` in the `Arith` library that we can make use of two times for this proof.

```
Require Import Arith.

Theorem add_assoc_4 : forall (a b c d : nat),
  (a + b + c) + d = a + (b + c + d).
Proof.
  intros.
  repeat rewrite -> Nat.add_assoc.
```

```
1 subgoal
a, b, c, d : nat
-------------------(1/1)
a + b + c + d = a + b + c + d
```

Acknowledgement: Inspired by the Coq Tactic Index (https://pjreddie.com/coq-tactics/) by Joseph Redmon.