Martin comment: search for an example that you cannot use simple induction to prove and try to prove it using strong induction and then see if you can do it in coq. also try printing out one of the examples of induction (reduction). and think of it as a function that takes now instead of just a variable a list but the problem is that lists are infinite so it may work if you want to work with just limited number of previous states instead of all previous ones. so use vectors instead!

TABLE OF CONTENTS INDEX ROADMAP

BASICS

FUNCTIONAL PROGRAMMING IN COQ

Introduction

The functional programming style is founded on simple, everyday mathematical intuition: If a procedure or method has no side effects, then (ignoring efficiency) all we need to understand about it is how it maps inputs to outputs — that is, we can think of it as just a concrete method for computing a mathematical function. This is one sense of the word "functional" in "functional programming." The direct connection between programs and simple mathematical objects supports both formal correctness proofs and sound informal reasoning about program behavior.

The other sense in which functional programming is "functional" is that it emphasizes the use of functions (or methods) as *first-class* values — i.e., values that can be passed as arguments to other functions, returned as results, included in data structures, etc. The recognition that functions can be treated as data gives rise to a host of useful and powerful programming idioms.

Other common features of functional languages include *algebraic data types* and *pattern matching*, which make it easy to construct and manipulate rich data structures, and sophisticated *polymorphic type systems* supporting abstraction and code reuse. Coq offers all of these features.

The first half of this chapter introduces the most essential elements of Coq's functional programming language, called *Gallina*. The second half introduces some basic *tactics* that can be used to prove properties of Coq programs.

Data and Functions

Enumerated Types

One notable aspect of Coq is that its set of built-in features is *extremely* small. For example, instead of providing the usual palette of atomic data types (booleans, integers, strings, etc.), Coq offers a powerful mechanism for defining new data types from scratch, with all these familiar types as instances.

Naturally, the Coq distribution comes preloaded with an extensive standard library providing definitions of booleans, numbers, and many common data structures like lists and hash tables. But there is nothing magic or primitive about these library definitions. To illustrate this, we will explicitly recapitulate all the definitions we need in this course, rather than just getting them implicitly from the library.

Days of the Week

To see how this definition mechanism works, let's start with a very simple example. The following declaration tells Coq that we are defining a new set of data values — a *type*.

data type

```
Inductive day : Type :=
  | monday : day
  | tuesday : day
  | wednesday : day
  | thursday : day
  | friday : day
  | saturday : day
  | sunday : day.
```

The type is called `day`, and its members are `monday`, `tuesday`, etc. The second and following lines of the definition can be read "`monday` is a `day`, `tuesday` is a `day`, etc."

Having defined `day`, we can write functions that operate on days.

function

```
Definition next_weekday (d:day) : day :=
  match d with
  | monday ⇒ tuesday
  | tuesday ⇒ wednesday
  | wednesday ⇒ thursday
  | thursday ⇒ friday
  | friday ⇒ monday
  | saturday ⇒ monday
  | sunday ⇒ monday
  end.
```

One thing to note is that the argument and return types of this function are explicitly declared. Like most functional programming languages, Coq can often figure out these types for itself when they are not given explicitly — i.e., it can do *type inference* — but we'll generally include them to make reading easier.

Having defined a function, we should check that it works on some examples. There are actually three different ways to do this in Coq. First, we can use the command `Compute` to evaluate a compound expression involving `next_weekday`.

```
Compute (next_weekday friday).
(* ==> monday : day *)
```

```
Compute (next_weekday (next_weekday saturday)).
(* ==> tuesday : day *)
```

(We show Coq's responses in comments, but, if you have a computer handy, this would be an excellent moment to fire up the Coq interpreter under your favorite IDE — either CoqIde or Proof General — and try this for yourself. Load this file, `Basics.v`, from the book's Coq sources, find the above example, submit it to Coq, and observe the result.)

Second, we can record what we *expect* the result to be in the form of a Coq example:

```
Example test_next_weekday:
  (next_weekday (next_weekday saturday)) = tuesday.
```

This declaration does two things: it makes an assertion (that the second weekday after `saturday` is `tuesday`), and it gives the assertion a name that can be used to refer to it later. Having made the assertion, we can also ask Coq to verify it, like this:

```
Proof. simpl. reflexivity. Qed.
```

The details are not important for now (we'll come back to them in a bit), but essentially this can be read as "The assertion we've just made can be proved by observing that both sides of the equality evaluate to the same thing, after some simplification."

Third, we can ask Coq to *extract*, from our `Definition`, a program in some other, more conventional, programming language (OCaml, Scheme, or Haskell) with a high-performance compiler. This facility is very interesting, since it gives us a way to go from proved-correct algorithms written in Gallina to efficient machine code. (Of course, we are trusting the correctness of the OCaml/Haskell/Scheme compiler, and of Coq's extraction facility itself, but this is still a big step forward from the way most software is developed today.) Indeed, this is one of the main uses for which Coq was developed. We'll come back to this topic in later chapters.

Homework Submission Guidelines

If you are using Software Foundations in a course, your instructor may use automatic scripts to help grade your homework assignments. In order for these scripts to work correctly (so that you get full credit for your work!), please be careful to follow these rules:

- The grading scripts work by extracting marked regions of the `.v` files that you submit. It is therefore important that you do not alter the "markup" that delimits exercises: the Exercise header, the name of the exercise, the "empty square bracket" marker at the end, etc. Please leave this markup exactly as you find it.
- Do not delete exercises. If you skip an exercise (e.g., because it is marked Optional, or because you can't solve it), it is OK to leave a partial proof in your `.v` file, but in this case please make sure it ends with `Admitted` (not, for example `Abort`).

- It is fine to use additional definitions (of helper functions, useful lemmas, etc.) in your solutions. You can put these between the exercise header and the theorem you are asked to prove.

Booleans

In a similar way, we can define the standard type `bool` of booleans, with members `true` and `false`.

```
Inductive bool : Type :=
  | true : bool
  | false : bool.
```

Although we are rolling our own booleans here for the sake of building up everything from scratch, Coq does, of course, provide a default implementation of the booleans, together with a multitude of useful functions and lemmas. (Take a look at `Coq.Init.Datatypes` in the Coq library documentation if you're interested.) Whenever possible, we'll name our own definitions and theorems so that they exactly coincide with the ones in the standard library.

Functions over booleans can be defined in the same way as above:

```
Definition negb (b:bool) : bool :=
  match b with
  | true ⇒ false
  | false ⇒ true
  end.

Definition andb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true ⇒ b2
  | false ⇒ false
  end.

Definition orb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | true ⇒ true
  | false ⇒ b2
  end.
```

The last two of these illustrate Coq's syntax for multi-argument function definitions. The corresponding multi-argument application syntax is illustrated by the following "unit tests," which constitute a complete specification — a truth table — for the `orb` function:

```
Example test_orb1: (orb true false) = true.
Proof. simpl. reflexivity. Qed.
Example test_orb2: (orb false false) = false.
Proof. simpl. reflexivity. Qed.
Example test_orb3: (orb false true) = true.
Proof. simpl. reflexivity. Qed.
Example test_orb4: (orb true true) = true.
Proof. simpl. reflexivity. Qed.
```

We can also introduce some familiar syntax for the boolean operations we have just defined. The `Notation` command defines a new symbolic notation for an existing definition.

```
Notation "x && y" := (andb x y).
Notation "x || y" := (orb x y).

Example test_orb5: false || false || true = true.
Proof. simpl. reflexivity. Qed.
```

A note on notation: In `.v` files, we use square brackets to delimit fragments of Coq code within comments; this convention, also used by the `coqdoc` documentation tool, keeps them visually separate from the surrounding text. In the HTML version of the files, these pieces of text appear in a `different font`.

The command `Admitted` can be used as a placeholder for an incomplete proof. We'll use it in exercises, to indicate the parts that we're leaving for you — i.e., your job is to replace `Admitted`s with real proofs.

Exercise: 1 star (nandb)

Remove "`Admitted.`" and complete the definition of the following function; then make sure that the `Example` assertions below can each be verified by Coq. (Remove "`Admitted.`" and fill in each proof, following the model of the `orb` tests above.) The function should return `true` if either or both of its inputs are `false`.

```
Definition nandb (b1:bool) (b2:bool) : bool
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_nandb1: (nandb true false) = true.
(* FILL IN HERE *) Admitted.
Example test_nandb2: (nandb false false) = true.
(* FILL IN HERE *) Admitted.
Example test_nandb3: (nandb false true) = true.
(* FILL IN HERE *) Admitted.
Example test_nandb4: (nandb true true) = false.
(* FILL IN HERE *) Admitted.
```

```
Definition nandb (b1:bool) (b2:bool) : bool :=
  match b1 with
  | false => true
  | true => negb b2
  end.
```

□

Exercise: 1 star (andb3)

Do the same for the `andb3` function below. This function should return `true` when all of its inputs are `true`, and `false` otherwise.

```
Definition andb3 (b1:bool) (b2:bool) (b3:bool) : bool
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_andb31: (andb3 true true true) = true.
(* FILL IN HERE *) Admitted.
Example test_andb32: (andb3 false true true) = false.
(* FILL IN HERE *) Admitted.
```

```
Definition andb3 (b1:bool) (b2:bool) (b3:bool) : bool :=
  match b1 with
  | true => andb b2 b3
  | false => false
  end.
```

```
Example test_andb33: (andb3 true false true) = false.
(* FILL IN HERE *) Admitted.
Example test_andb34: (andb3 true true false) = false.
(* FILL IN HERE *) Admitted.
```

□

Function Types

Every expression in Coq has a type, describing what sort of thing it computes. The `Check` command asks Coq to print the type of an expression.

```
Check true.
(* ===> true : bool *)
Check (negb true).
(* ===> negb true : bool *)
```

Functions like `negb` itself are also data values, just like `true` and `false`. Their types are called *function types*, and they are written with arrows.

```
Check negb.
(* ===> negb : bool -> bool *)
```

The type of `negb`, written `bool → bool` and pronounced "`bool` arrow `bool`," can be read, "Given an input of type `bool`, this function produces an output of type `bool`." Similarly, the type of `andb`, written `bool → bool → bool`, can be read, "Given two inputs, both of type `bool`, this function produces an output of type `bool`."

Compound Types

The types we have defined so far are examples of "enumerated types": their definitions explicitly enumerate a finite set of elements, each of which is just a bare constructor. Here is a more interesting type definition, where one of the constructors takes an argument:

```
Inductive rgb : Type :=
  | red : rgb
  | green : rgb
  | blue : rgb.

Inductive color : Type :=
  | black : color
  | white : color
  | primary : rgb → color.
```

Let's look at this in a little more detail.

Every inductively defined type (`day`, `bool`, `rgb`, `color`, etc.) contains a set of *constructor expressions* built from *constructors* like `red`, `primary`, `true`, `false`, `monday`, etc. The definitions of `rgb` and `color` say how expressions in the sets `rgb` and `color` can be built:

- `red`, `green`, and `blue` are the constructors of `rgb`;
- `black`, `white`, and `primary` are the constructors of `color`;

- the expression `red` belongs to the set `rgb`, as do the expressions `green` and `blue`;
- the expressions `black` and `white` belong to the set `color`;
- if `p` is an expression belonging to the set `rgb`, then `primary p` (pronounced "the constructor `primary` applied to the argument `p`") is an expression belonging to the set `color`; and
- expressions formed in these ways are the *only* ones belonging to the sets `rgb` and `color`.

We can define functions on colors using pattern matching just as we have done for `day` and `bool`.

```
Definition monochrome (c : color) : bool :=
  match c with
  | black ⇒ true
  | white ⇒ true
  | primary p ⇒ false
  end.
```

Since the `primary` constructor takes an argument, a pattern matching `primary` should include either a variable (as above) or a constant of appropriate type (as below).

```
Definition isred (c : color) : bool :=
  match c with
  | black ⇒ false
  | white ⇒ false
  | primary red ⇒ true
  | primary _ ⇒ false
  end.
```

The pattern `primary _` here is shorthand for "`primary` applied to any `rgb` constructor except `red`." (The wildcard pattern `_` has the same effect as the dummy pattern variable `p` in the definition of `monochrome`.)

Modules

Coq provides a *module system*, to aid in organizing large developments. In this course we won't need most of its features, but one is useful: If we enclose a collection of declarations between `Module X` and `End X` markers, then, in the remainder of the file after the `End`, these definitions are referred to by names like `X.foo` instead of just `foo`. We will use this feature to introduce the definition of the type `nat` in an inner module so that it does not interfere with the one from the standard library (which we want to use in the rest because it comes with a tiny bit of convenient special notation).

```
Module NatPlayground.
```

Numbers

An even more interesting way of defining a type is to allow its constructors to take arguments from the very same type — that is, to allow the rules describing its

elements to be *inductive*.

For example, we can define (a unary representation of) natural numbers as follows:

```
Inductive nat : Type :=
  | O : nat
  | S : nat → nat.
```

The clauses of this definition can be read:

- O is a natural number (note that this is the letter "O," not the numeral "0").
- S can be put in front of a natural number to yield another one — if n is a natural number, then S n is too.

Again, let's look at this in a little more detail. The definition of nat says how expressions in the set nat can be built:

- O and S are constructors;
- the expression O belongs to the set nat;
- if n is an expression belonging to the set nat, then S n is also an expression belonging to the set nat; and
- expressions formed in these two ways are the only ones belonging to the set nat.

The same rules apply for our definitions of day, bool, color, etc.

The above conditions are the precise force of the Inductive declaration. They imply that the expression O, the expression S O, the expression S (S O), the expression S (S (S O)), and so on all belong to the set nat, while other expressions built from data constructors, like true, andb true false, S (S false), and O (O (O S)) do not.

A critical point here is that what we've done so far is just to define a *representation* of numbers: a way of writing them down. The names O and S are arbitrary, and at this point they have no special meaning — they are just two different marks that we can use to write down numbers (together with a rule that says any nat will be written as some string of S marks followed by an O). If we like, we can write essentially the same definition this way:

```
Inductive nat' : Type :=
  | stop : nat'
  | tick : nat' → nat'.
```

The *interpretation* of these marks comes from how we use them to compute.

We can do this by writing functions that pattern match on representations of natural numbers just as we did above with booleans and days — for example, here is the predecessor function:

```
Definition pred (n : nat) : nat :=
  match n with
    | O ⇒ O
    | S n' ⇒ n'
  end.
```

The second branch can be read: "if `n` has the form `S n'` for some `n'`, then return `n'`."

```
End NatPlayground.
```

Because natural numbers are such a pervasive form of data, Coq provides a tiny bit of built-in magic for parsing and printing them: ordinary arabic numerals can be used as an alternative to the "unary" notation defined by the constructors `S` and `O`. Coq prints numbers in arabic form by default:

```
Check (S (S (S (S O)))).
  (* ===> 4 : nat *)

Definition minustwo (n : nat) : nat :=
  match n with
    | O ⇒ O
    | S O ⇒ O
    | S (S n') ⇒ n'
  end.

Compute (minustwo 4).
  (* ===> 2 : nat *)
```

The constructor `S` has the type `nat → nat`, just like `pred` and functions like `minustwo`:

```
Check S.
Check pred.
Check minustwo.
```

These are all things that can be applied to a number to yield a number. However, there is a fundamental difference between the first one and the other two: functions like `pred` and `minustwo` come with *computation rules* — e.g., the definition of `pred` says that `pred 2` can be simplified to `1` — while the definition of `S` has no such behavior attached. Although it is like a function in the sense that it can be applied to an argument, it does not *do* anything at all! It is just a way of writing down numbers. (Think about standard arabic numerals: the numeral `1` is not a computation; it's a piece of data. When we write `111` to mean the number one hundred and eleven, we are using `1`, three times, to write down a concrete representation of a number.)

For most function definitions over numbers, just pattern matching is not enough: we also need recursion. For example, to check that a number `n` is even, we may need to recursively check whether `n-2` is even. To write such functions, we use the keyword `Fixpoint`.

recursive

```
Fixpoint evenb (n:nat) : bool :=
  match n with
  | O ⇒ true
  | S O ⇒ false
  | S (S n') ⇒ evenb n'
  end.
```

We can define `oddb` by a similar `Fixpoint` declaration, but here is a simpler definition:

```
Definition oddb (n:nat) : bool := negb (evenb n).

Example test_oddb1: oddb 1 = true.
Proof. simpl. reflexivity. Qed.
Example test_oddb2: oddb 4 = false.
Proof. simpl. reflexivity. Qed.
```

can't i just do it by function application
i.e.: negb evenb?

(You will notice if you step through these proofs that `simpl` actually has no effect on the goal — all of the work is done by `reflexivity`. We'll see more about why that is shortly.)

Naturally, we can also define multi-argument functions by recursion.

```
Module NatPlayground2.

Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
    | O ⇒ m
    | S n' ⇒ S (plus n' m)
  end.
```

Adding three to two now gives us five, as we'd expect.

```
Compute (plus 3 2).
```

The simplification that Coq performs to reach this conclusion can be visualized as follows:

is there a command that can show us this simplification? write a proof for it. too? maybe, search. DO IT!!

```
(*  plus (S (S (S O))) (S (S O))
==> S (plus (S (S O)) (S (S O)))
      by the second clause of the match
==> S (S (plus (S O) (S (S O))))
      by the second clause of the match
==> S (S (S (plus O (S (S O)))))
      by the second clause of the match
==> S (S (S (S O)))
      by the first clause of the match
*)
```

As a notational convenience, if two or more arguments have the same type, they can be written together. In the following definition, `(n m : nat)` means just the same as if we had written `(n : nat) (m : nat)`.

```
Fixpoint mult (n m : nat) : nat :=
  match n with
    | O ⇒ O
    | S n' ⇒ plus m (mult n' m)
  end.

Example test_mult1: (mult 3 3) = 9.
Proof. simpl. reflexivity. Qed.
```

You can match two expressions at once by putting a comma between them:

```
Fixpoint minus (n m:nat) : nat :=
  match n, m with
  | O , _ ⇒ O
```

```
  | S _ , O ⇒ n
  | S n', S m' ⇒ minus n' m'
  end.
```

Again, the _ in the first line is a *wildcard pattern*. Writing _ in a pattern is the same as writing some variable that doesn't get used on the right-hand side. This avoids the need to invent a variable name.

```
End NatPlayground2.

Fixpoint exp (base power : nat) : nat :=
  match power with
    | O ⇒ S O
    | S p ⇒ mult base (exp base p)
  end.
```

Exercise: 1 star (factorial)

Recall the standard mathematical factorial function:

```
    factorial(0)  =  1
    factorial(n)  =  n * factorial(n-1)     (if n>0)
```

Translate this into Coq.

```
Fixpoint factorial (n:nat) : nat
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_factorial1: (factorial 3) = 6.
(* FILL IN HERE *) Admitted.
Example test_factorial2: (factorial 5) = (mult 10 12).
(* FILL IN HERE *) Admitted.
```

□

We can make numerical expressions a little easier to read and write by introducing *notations* for addition, multiplication, and subtraction.

```
Notation "x + y" := (plus x y)
                      (at level 50, left associativity)
                      : nat_scope.
Notation "x - y" := (minus x y)
                      (at level 50, left associativity)
                      : nat_scope.
Notation "x * y" := (mult x y)
                      (at level 40, left associativity)
                      : nat_scope.

Check ((0 + 1) + 1).
```

(The `level`, `associativity`, and `nat_scope` annotations control how these notations are treated by Coq's parser. The details are not important for our purposes, but interested readers can refer to the optional "More on Notation" section at the end of this chapter.)

Note that these do not change the definitions we've already made: they are simply instructions to the Coq parser to accept `x + y` in place of `plus x y` and, conversely, to

the Coq pretty-printer to display `plus x y` as `x + y`.

When we say that Coq comes with almost nothing built-in, we really mean it: even equality testing for numbers is a user-defined operation! We now define a function `beq_nat`, which tests `natural` numbers for `equality`, yielding a `boolean`. Note the use of nested `matches` (we could also have used a simultaneous match, as we did in `minus`.)

any benefit to using nested? or simultaneous match?

```
Fixpoint beq_nat (n m : nat) : bool :=
  match n with
  | O ⇒ match m with
        | O ⇒ true
        | S m' ⇒ false
        end
  | S n' ⇒ match m with
            | O ⇒ false
            | S m' ⇒ beq_nat n' m'
            end
  end.
```

FYI: doesn't have . at this end

The `leb` function tests whether its first argument is less than or equal to its second argument, yielding a boolean.

```
Fixpoint leb (n m : nat) : bool :=
  match n with
  | O ⇒ true
  | S n' ⇒
      match m with
      | O ⇒ false
      | S m' ⇒ leb n' m'
      end
  end.

Example test_leb1: (leb 2 2) = true.
Proof. simpl. reflexivity. Qed.
Example test_leb2: (leb 2 4) = true.
Proof. simpl. reflexivity. Qed.
Example test_leb3: (leb 4 2) = false.
Proof. simpl. reflexivity. Qed.
```

Exercise: 1 star (blt_nat)

The `blt_nat` function tests `natural` numbers for `less-than`, yielding a `boolean`. Instead of making up a new `Fixpoint` for this one, define it in terms of a previously defined function.

```
Definition blt_nat (n m : nat) : bool
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_blt_nat1: (blt_nat 2 2) = false.
(* FILL IN HERE *) Admitted.
Example test_blt_nat2: (blt_nat 2 4) = true.
(* FILL IN HERE *) Admitted.
Example test_blt_nat3: (blt_nat 4 2) = false.
(* FILL IN HERE *) Admitted.
```

□

Proof by Simplification

Now that we've defined a few datatypes and functions, let's turn to stating and proving properties of their behavior. Actually, we've already started doing this: each `Example` in the previous sections makes a precise claim about the behavior of some function on some particular inputs. The proofs of these claims were always the same: use `simpl` to simplify both sides of the equation, then use `reflexivity` to check that both sides contain identical values.

The same sort of "proof by simplification" can be used to prove more interesting properties as well. For example, the fact that `0` is a "neutral element" for `+` on the left can be proved just by observing that `0 + n` reduces to `n` no matter what `n` is, a fact that can be read directly off the definition of `plus`.

```
Theorem plus_O_n : ∀ n : nat, 0 + n = n.
Proof.
  intros n. simpl. reflexivity. Qed.
```

(You may notice that the above statement looks different in the `.v` file in your IDE than it does in the HTML rendition in your browser, if you are viewing both. In `.v` files, we write the ∀universal quantifier using the reserved identifier "forall." When the `.v` files are converted to HTML, this gets transformed into an upside-down-A symbol.)

This is a good place to mention that `reflexivity` is a bit more powerful than we have admitted. In the examples we have seen, the calls to `simpl` were actually not needed, because `reflexivity` can perform some simplification automatically when checking that two sides are equal; `simpl` was just added so that we could see the intermediate state — after simplification but before finishing the proof. Here is a shorter proof of the theorem:

```
Theorem plus_O_n' : ∀ n : nat, 0 + n = n.
Proof.
  intros n. reflexivity. Qed.
```

Moreover, it will be useful later to know that `reflexivity` does somewhat *more* simplification than `simpl` does — for example, it tries "unfolding" defined terms, replacing them with their right-hand sides. The reason for this difference is that, if reflexivity succeeds, the whole goal is finished and we don't need to look at whatever expanded expressions `reflexivity` has created by all this simplification and unfolding; by contrast, `simpl` is used in situations where we may have to read and understand the new goal that it creates, so we would not want it blindly expanding definitions and leaving the goal in a messy state.

The form of the theorem we just stated and its proof are almost exactly the same as the simpler examples we saw earlier; there are just a few differences.

First, we've used the keyword `Theorem` instead of `Example`. This difference is mostly a matter of style; the keywords `Example` and `Theorem` (and a few others, including `Lemma`, `Fact`, and `Remark`) mean pretty much the same thing to Coq.

Second, we've added the quantifier `∀n:nat`, so that our theorem talks about *all* natural numbers `n`. Informally, to prove theorems of this form, we generally start by saying "Suppose `n` is some number..." Formally, this is achieved in the proof by `intros n`, which moves `n` from the quantifier in the goal to a *context* of current assumptions.and it gets the assumption from the theorem, right?

The keywords `intros`, `simpl`, and `reflexivity` are examples of *tactics*. A tactic is a command that is used between `Proof` and `Qed` to guide the process of checking some claim we are making. We will see several more tactics in the rest of this chapter and yet more in future chapters.

Other similar theorems can be proved with the same pattern.

```
Theorem plus_1_l : ∀ n:nat, 1 + n = S n.
Proof.
  intros n. reflexivity. Qed.

Theorem mult_0_l : ∀ n:nat, 0 * n = 0.
Proof.
  intros n. reflexivity. Qed.
```

The `_l` suffix in the names of these theorems is pronounced "on the left."

It is worth stepping through these proofs to observe how the context and the goal change. You may want to add calls to `simpl` before `reflexivity` to see the simplifications that Coq performs on the terms before checking that they are equal.

Proof by Rewriting

This theorem is a bit more interesting than the others we've seen:

```
Theorem plus_id_example : ∀ n m:nat,
  n = m →
  n + n = m + m.
```

Instead of making a universal claim about all numbers `n` and `m`, it talks about a more specialized property that only holds when `n = m`. The arrow symbol is pronounced "implies."

As before, we need to be able to reason by assuming we are given such numbers `n` and `m`. We also need to assume the hypothesis `n = m`. The `intros` tactic will serve to move all three of these from the goal into assumptions in the current context.

Since `n` and `m` are arbitrary numbers, we can't just use simplification to prove this theorem. Instead, we prove it by observing that, if we are assuming `n = m`, then we can replace `n` with `m` in the goal statement and obtain an equality with the same

expression on both sides. The tactic that tells Coq to perform this replacement is called `rewrite`.

```
Proof.
  (* move both quantifiers into the context: *)
  intros n m.
  (* move the hypothesis into the context: *)
  intros H.
  (* rewrite the goal using the hypothesis: *)
  rewrite → H.
  reflexivity. Qed.
```

why left with right, and not the other way around? are there cases that matters which direction we choose?
YES, the ex at the bottom of this page illustrate that the direction may matter!! interestingly even after changing direction it still worked!!

The first line of the proof moves the universally quantified variables `n` and `m` into the context. The second moves the hypothesis `n = m` into the context and gives it the name `H`. The third tells Coq to rewrite the current goal (`n + n = m + m`) by replacing the left side of the equality hypothesis `H` with the right side.

FYI: so i'll have:
m +m = m+ m
right? YES

(The arrow symbol in the `rewrite` has nothing to do with implication: it tells Coq to apply the rewrite from left to right. To rewrite from right to left, you can use `rewrite <-`. Try making this change in the above proof and see what difference it makes.)

FYI: if I use <- then i'll have:
n + n = n + n

Exercise: 1 star (plus_id_exercise)

Remove "`Admitted.`" and fill in the proof.

```
Theorem plus_id_exercise : ∀ n m o : nat,
  n = m → m = o → n + m = m + o.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

The `Admitted` command tells Coq that we want to skip trying to prove this theorem and just accept it as a given. This can be useful for developing longer proofs, since we can state subsidiary lemmas that we believe will be useful for making some larger argument, use `Admitted` to accept them on faith for the moment, and continue working on the main argument until we are sure it makes sense; then we can go back and fill in the proofs we skipped. Be careful, though: every time you say `Admitted` you are leaving a door open for total nonsense to enter Coq's nice, rigorous, formally checked world!

We can also use the `rewrite` tactic with a previously proved theorem instead of a hypothesis from the context. If the statement of the previously proved theorem involves quantified variables, as in the example below, Coq tries to instantiate them by matching with the current goal.

```
Theorem mult_0_plus : ∀ n m : nat,
  (0 + n) * m = n * m.
Proof.
  intros n m.
  rewrite → plus_O_n.
  reflexivity. Qed.
```

-> : n * m = n *m
<- : (0 + n)*m = 0+n*m
and it works!!

AHA! **Exercise: 2 stars (mult_S_1)**

```
Theorem mult_S_1 : ∀ n m : nat,
  m = S n →
  m * (1 + n) = m * m.
Proof.
  (* FILL IN HERE *) Admitted.

  (* (N.b. This proof can actually be completed with tactics other than
     rewrite, but please do use rewrite for the sake of the exercise.) *)
```

FYI: the order of tactics really matter!!!

□

Proof by Case Analysis

Of course, not everything can be proved by simple calculation and rewriting: In general, unknown, hypothetical values (arbitrary numbers, booleans, lists, etc.) can block simplification. For example, if we try to prove the following fact using the `simpl` tactic as above, we get stuck. (We then use the `Abort` command to give up on it for the moment.)

```
Theorem plus_1_neq_0_firsttry : ∀ n : nat,
  beq_nat (n + 1) 0 = false.
Proof.
  intros n.
  simpl. (* does nothing! *)
Abort.
```

The reason for this is that the definitions of both `beq_nat` and `+` begin by performing a `match` on their first argument. But here, the first argument to `+` is the unknown number `n` and the argument to `beq_nat` is the compound expression `n + 1`; neither can be simplified.

To make progress, we need to consider the possible forms of `n` separately. If `n` is `O`, then we can calculate the final result of `beq_nat (n + 1) 0` and check that it is, indeed, `false`. And if `n = S n'` for some `n'`, then, although we don't know exactly what number `n + 1` yields, we can calculate that, at least, it will begin with one `S`, and this is enough to calculate that, again, `beq_nat (n + 1) 0` will yield `false`.

The tactic that tells Coq to consider, separately, the cases where `n = O` and where `n = S n'` is called `destruct`.

```
Theorem plus_1_neq_0 : ∀ n : nat,
  beq_nat (n + 1) 0 = false.
Proof.
  intros n. destruct n as [| n'].
  - reflexivity.
  - reflexivity. Qed.
```

The `destruct` generates *two* subgoals, which we must then prove, separately, in order to get Coq to accept the theorem. The annotation "`as [ | n' ]`" is called an *intro*

pattern. It tells Coq what variable names to introduce in each subgoal. In general, what goes between the square brackets is a *list of lists* of names, separated by `|`. In this case, the first component is empty, since the `O` constructor is nullary (it doesn't have any arguments). The second component gives a single name, `n'`, since `S` is a unary constructor.

The `-` signs on the second and third lines are called *bullets*, and they mark the parts of the proof that correspond to each generated subgoal. The proof script that comes after a bullet is the entire proof for a subgoal. In this example, each of the subgoals is easily proved by a single use of `reflexivity`, which itself performs some simplification — e.g., the first one simplifies `beq_nat (S n') 0` to `false` by first rewriting `(S n' + 1)` to `S (n' + 1)`, then unfolding `beq_nat`, and then simplifying the `match`.

how? for case analysis, if I don't use the bullets, how would it work?

Marking cases with bullets is entirely optional: if bullets are not present, Coq simply asks you to prove each subgoal in sequence, one at a time. But it is a good idea to use bullets. For one thing, they make the structure of a proof apparent, making it more readable. Also, bullets instruct Coq to ensure that a subgoal is complete before trying to verify the next one, preventing proofs for different subgoals from getting mixed up. These issues become especially important in large developments, where fragile proofs lead to long debugging sessions.

There are no hard and fast rules for how proofs should be formatted in Coq — in particular, where lines should be broken and how sections of the proof should be indented to indicate their nested structure. However, if the places where multiple subgoals are generated are marked with explicit bullets at the beginning of lines, then the proof will be readable almost no matter what choices are made about other aspects of layout.

This is also a good place to mention one other piece of somewhat obvious advice about line lengths. Beginning Coq users sometimes tend to the extremes, either writing each tactic on its own line or writing entire proofs on one line. Good style lies somewhere in the middle. One reasonable convention is to limit yourself to 80-character lines.

The `destruct` tactic can be used with any inductively defined datatype. For example, we use it next to prove that boolean negation is involutive — i.e., that negation is its own inverse.

```
Theorem negb_involutive : ∀ b : bool,
  negb (negb b) = b.
Proof.
  intros b. destruct b.
  - reflexivity.
  - reflexivity. Qed.
```

Note that the `destruct` here has no `as` clause because none of the subcases of the `destruct` need to bind any variables, so there is no need to specify any names. (We could also have written `as [ | ]`, or `as [ ]`.) In fact, we can omit the `as` clause from *any*

`destruct` and Coq will fill in variable names automatically. This is generally considered bad style, since Coq often makes confusing choices of names when left to its own devices.

It is sometimes useful to invoke `destruct` inside a subgoal, generating yet more proof obligations. In this case, we use different kinds of bullets to mark goals on different "levels." For example:

```
Theorem andb_commutative : ∀ b c, andb b c = andb c b.
Proof.
  intros b c. destruct b.
  - destruct c.
    + reflexivity.
    + reflexivity.
  - destruct c.
    + reflexivity.
    + reflexivity.
Qed.
```

Each pair of calls to `reflexivity` corresponds to the subgoals that were generated after the execution of the `destruct c` line right above it.

Besides `-` and `+`, we can use `*` (asterisk) as a third kind of bullet. We can also enclose sub-proofs in curly braces, which is useful in case we ever encounter a proof that generates more than three levels of subgoals:

```
Theorem andb_commutative' : ∀ b c, andb b c = andb c b.
Proof.
  intros b c. destruct b.
  { destruct c.
    { reflexivity. }
    { reflexivity. } }
  { destruct c.
    { reflexivity. }
    { reflexivity. } }
Qed.
```

Since curly braces mark both the beginning and the end of a proof, they can be used for multiple subgoal levels, as this example shows. Furthermore, curly braces allow us to reuse the same bullet shapes at multiple levels in a proof:

```
Theorem andb3_exchange :
  ∀ b c d, andb (andb b c) d = andb (andb b d) c.
Proof.
  intros b c d. destruct b.
  - destruct c.
    { destruct d.
      - reflexivity.
      - reflexivity. }
    { destruct d.
      - reflexivity.
      - reflexivity. }
  - destruct c.
    { destruct d.
      - reflexivity.
```

```
    - reflexivity. }
  { destruct d.
    - reflexivity.
    - reflexivity. }
Qed.
```

Before closing the chapter, let's mention one final convenience. As you may have noticed, many proofs perform case analysis on a variable right after introducing it:

```
intros x y. destruct y as [|y].
```

This pattern is so common that Coq provides a shorthand for it: we can perform case analysis on a variable when introducing it by using an intro pattern instead of a variable name. For instance, here is a shorter proof of the `plus_1_neq_0` theorem above.

```
Theorem plus_1_neq_0' : ∀ n : nat,
  beq_nat (n + 1) 0 = false.
Proof.
  intros [|n].
  - reflexivity.
  - reflexivity. Qed.
```

If there are no arguments to name, we can just write `[]`.

```
Theorem andb_commutative'' :
  ∀ b c, andb b c = andb c b.
Proof.
  intros [] [].
  - reflexivity.
  - reflexivity.
  - reflexivity.
  - reflexivity.
Qed.
```

the order of cases depends on the order of your cases in data type def!! so if you change that your proof is gonna break!!! one of the huge problems of coq!

Exercise: 2 stars (andb_true_elim2)

Prove the following claim, marking cases (and subcases) with bullets when you use `destruct`.

```
Theorem andb_true_elim2 : ∀ b c : bool,
  andb b c = true → c = true.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

really cool!!! to see how the assumption or hypothesis can be wrong or false and yet the conclusion follows.

Exercise: 1 star (zero_nbeq_plus_1)

```
Theorem zero_nbeq_plus_1 : ∀ n : nat,
  beq_nat 0 (n + 1) = false.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

More on Notation (Optional)

(In general, sections marked Optional are not needed to follow the rest of the book, except possibly other Optional sections. On a first reading, you might want to skim these sections so that you know what's there for future reference.)

Recall the notation definitions for infix plus and times:

```
Notation "x + y" := (plus x y)
                       (at level 50, left associativity)
                       : nat_scope.
Notation "x * y" := (mult x y)
                       (at level 40, left associativity)
                       : nat_scope.
```

For each notation symbol in Coq, we can specify its *precedence level* and its *associativity*. The precedence level `n` is specified by writing `at level n`; this helps Coq parse compound expressions. The associativity setting helps to disambiguate expressions containing multiple occurrences of the same symbol. For example, the parameters specified above for `+` and `*` say that the expression `1+2*3*4` is shorthand for `(1+((2*3)*4))`. Coq uses precedence levels from 0 to 100, and *left*, *right*, or *no* associativity. We will see more examples of this later, e.g., in the `Lists` chapter.

Each notation symbol is also associated with a *notation scope*. Coq tries to guess what scope is meant from context, so when it sees `S(O*O)` it guesses `nat_scope`, but when it sees the cartesian product (tuple) type `bool*bool` (which we'll see in later chapters) it guesses `type_scope`. Occasionally, it is necessary to help it out with percent-notation by writing `(x*y)%nat`, and sometimes in what Coq prints it will use `%nat` to indicate what scope a notation is in.

Notation scopes also apply to numeral notation (`3`, `4`, `5`, etc.), so you may sometimes see `0%nat`, which means `O` (the natural number `0` that we're using in this chapter), or `0%Z`, which means the Integer zero (which comes from a different part of the standard library).

Pro tip: Coq's notation mechanism is not especially powerful. Don't expect too much from it!

Fixpoints and Structural Recursion (Optional)

Here is a copy of the definition of addition:

```
Fixpoint plus' (n : nat) (m : nat) : nat :=
  match n with
  | O ⇒ m
  | S n' ⇒ S (plus' n' m)
  end.
```

When Coq checks this definition, it notes that `plus'` is "decreasing on 1st argument." What this means is that we are performing a *structural recursion* over the argument `n` — i.e., that we make recursive calls only on strictly smaller values of `n`. This implies that all calls to `plus'` will eventually terminate. Coq demands that some argument of *every* `Fixpoint` definition is "decreasing."

KEEP IN MIND that coq is a lang for describing how to construct proof trees!

This requirement is a fundamental feature of Coq's design: In particular, it guarantees that every function that can be defined in Coq will terminate on all inputs. However, because Coq's "decreasing analysis" is not very sophisticated, it is sometimes necessary to write functions in slightly unnatural ways.

Exercise: 2 stars, optional (decreasing)

look at subset def that you have in lists.v file

To get a concrete sense of this, find a way to write a sensible `Fixpoint` definition (of a simple function on numbers, say) that *does* terminate on all inputs, but that Coq will reject because of this restriction.

```
(* FILL IN HERE *)
```

□

More Exercises

Exercise: 2 stars (boolean_functions)

Use the tactics you have learned so far to prove the following theorem about boolean functions.

```
Theorem identity_fn_applied_twice :
  ∀ (f : bool → bool),
  (∀ (x : bool), f x = x) →
  ∀ (b : bool), f (f b) = b.
Proof.
  (* FILL IN HERE *) Admitted.
```

Now state and prove a theorem `negation_fn_applied_twice` similar to the previous one but where the second hypothesis says that the function `f` has the property that `f x = negb x`.

any other solution than repeating the above?

```
(* FILL IN HERE *)
```

□

Exercise: 3 stars, optional (andb_eq_orb)

Prove the following theorem. (Hint: This one can be a bit tricky, depending on how you approach it. You will probably need both `destruct` and `rewrite`, but destructing everything in sight is not the best way.)

```
Theorem andb_eq_orb :
  ∀ (b c : bool),
  (andb b c = orb b c) →
  b = c.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (binary)

Consider a different, more efficient representation of natural numbers using a binary rather than unary system. That is, instead of saying that each natural number is either zero or the successor of a natural number, we can say that each binary number is either

- zero,
- twice a binary number, or
- one more than twice a binary number.

(a) First, write an inductive definition of the type `bin` corresponding to this description of binary numbers.

(Hint: Recall that the definition of `nat` above,

```
Inductive nat : Type :=
  | O : nat
  | S : nat → nat.
```

says nothing about what `O` and `S` "mean." It just says "`O` is in the set called `nat`, and if `n` is in the set then so is `S n`." The interpretation of `O` as zero and `S` as successor/plus one comes from the way that we *use* `nat` values, by writing functions to do things with them, proving things about them, and so on. Your definition of `bin` should be correspondingly simple; it is the functions you will write next that will give it mathematical meaning.)

One caveat: If you use `O` or `S` as constructor names in your definition, it will confuse the auto-grader script in `BasicsTest.v`. Please choose different names.

(b) Next, write an increment function `incr` for binary numbers, and a function `bin_to_nat` to convert binary numbers to unary numbers.

(c) Write five unit tests `test_bin_incr1`, `test_bin_incr2`, etc. for your increment and binary-to-unary functions. (A "unit test" in Coq is a specific `Example` that can be proved with just `reflexivity`, as we've done for several of our definitions.) Notice that incrementing a binary number and then converting it to unary should yield the same result as first converting it to unary and then incrementing.

```
(* FILL IN HERE *)
```

□