BASICS

FUNCTIONAL PROGRAMMING IN COQ

Introduction

The functional style of programming 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 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 *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 native functional programming language, called *Gallina*. The second half introduces some basic *tactics* that can be used to prove properties of Gallina 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 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, this course we will explicitly recapitulate (almost) all the definitions we need, rather than getting them from the standard 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 set of data values -- a *type*.

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

The new type is called `day`, and its members are `monday`, `tuesday`, etc.

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

```
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 point 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 next check that it works on some examples. There are actually three different ways to do the examples 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 it 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 just now, 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."

Third, we can ask Coq to *extract*, from our `Definition`, a program in another, more conventional, programming language (OCaml, Scheme, or Haskell) with a high-performance compiler. This facility is very interesting, since it gives us a path 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 (and give you 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; 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.
- If you introduce a helper lemma that you end up being unable to prove, hence end it with `Admitted`, then make sure to also end the main theorem in which you use it with `Admitted`, not `Qed`. That will help you get partial credit, in case you use that main theorem to solve a later exercise.

You will also notice that each chapter (like `Basics.v`) is accompanied by a *test script* (`BasicsTest.v`) that automatically calculates points for the finished homework problems in the chapter. These scripts are mostly for the auto-grading tools, but you may also want to use them to double-check that your file is well formatted before handing it in. In a terminal window, either type "`make BasicsTest.vo`" or do the following:

```
coqc -Q . LF Basics.v
coqc -Q . LF BasicsTest.v
```

See the end of this chapter for more information about how to interpret the output of test scripts.

There is no need to hand in `BasicsTest.v` itself (or `Preface.v`).

If your class is using the Canvas system to hand in assignments...

- If you submit multiple versions of the assignment, you may notice that they are given different names. This is fine: The most recent submission is the one that will be graded.
- To hand in multiple files at the same time (if more than one chapter is assigned in the same week), you need to make a single submission with all the files at once using the button "Add another file" just above the comment box.

The `Require Export` statement on the next line tells Coq to use the `String` module from the standard library. We'll use strings ourselves in later chapters, but we need to `Require` it here so that the grading scripts can use it for internal purposes.

```
From Coq Require Export String.
```

Booleans

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

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

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.
```

(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. Whenever possible, we'll name our own definitions and theorems so that they exactly coincide with the ones in the standard library.)

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 infix 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`.

Exercise: 1 star, standard (nandb)

The command `Admitted` can be used as a placeholder for an incomplete proof. We 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.

Remove "`Admitted.`" and complete the definition of the following function; then make sure that the `Example` assertions below can each be verified by Coq. (I.e., fill in each proof, following the model of the `orb` tests above, and make sure Coq accepts it.) 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.
```

□

Exercise: 1 star, standard (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.
Example test_andb33: (andb3 true false true) = false.
(* FILL IN HERE *) Admitted.
Example test_andb34: (andb3 true true false) = false.
(* FILL IN HERE *) Admitted.
```

□

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 *)
```

If the expression after `Check` is followed by a colon and a type, Coq will verify that the type of the expression matches the given type and halt with an error if not.

```
Check true
    : bool.
Check (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
    : 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, each of type `bool`, this function produces an output of type `bool`."

New Types from Old

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

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

Inductive color : Type :=
  | black
  | white
  | primary (p : rgb).
```

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

Every inductively defined type (`day`, `bool`, `rgb`, `color`, etc.) describes a set of *constructor expressions* built from *constructors*.

- We start with an infinite set of *constructors*. E.g., `red`, `primary`, `true`, `false`, `monday`, etc. are constructors.
- *Constructor expressions* are formed by applying a constructor to zero or more other constructors or constructor expressions. E.g.,
  - `red`
  - `true`
  - `primary`
  - `primary red`
  - `red primary`
  - `red true`
  - `primary (primary primary)`

- etc.

- An `Inductive` definition carves out a subset of the whole space of constructor expressions and gives it a name, like `bool`, `rgb`, or `color`.

In particular, the definitions of `rgb` and `color` say which constructor expressions belong to the sets `rgb` and `color`:

- `red`, `green`, and `blue` belong to the set `rgb`;
- `black` and `white` belong to the set `color`;
- if `p` is a constructor expression belonging to the set `rgb`, then `primary p` (pronounced "the constructor `primary` applied to the argument `p`") is a constructor expression belonging to the set `color`; and
- constructor 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 did 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 -- note that we can choose its name freely) 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 "the constructor `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. 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 limit the scope of definitions, so that we are free to reuse names.

```
Module Playground.
  Definition b : rgb := blue.
End Playground.

Definition b : bool := true.

Check Playground.b : rgb.
Check b : bool.
```

Tuples

```
Module TuplePlayground.
```

A single constructor with multiple parameters can be used to create a tuple type. As an example, consider representing the four bits in a nybble (half a byte). We first define a datatype `bit` that resembles `bool` (using the constructors B_0 and B_1 for the two possible bit values) and then define the datatype `nybble`, which is essentially a tuple of four bits.

```
Inductive bit : Type :=
  | B0
  | B1.

Inductive nybble : Type :=
  | bits (b0 b1 b2 b3 : bit).

Check (bits B1 B0 B1 B0)
    : nybble.
```

The `bits` constructor acts as a wrapper for its contents. Unwrapping can be done by pattern-matching, as in the `all_zero` function which tests a nybble to see if all its bits are B_0. We use underscore (_) as a *wildcard pattern* to avoid inventing variable names that will not be used.

```
Definition all_zero (nb : nybble) : bool :=
  match nb with
    | (bits B0 B0 B0 B0) ⇒ true
    | (bits _ _ _ _) ⇒ false
  end.

Compute (all_zero (bits B1 B0 B1 B0)).
(* ===> false : bool *)
Compute (all_zero (bits B0 B0 B0 B0)).
(* ===> true : bool *)

End TuplePlayground.
```

Numbers

We put this section in a module so that our own definition of natural numbers does not interfere with the one from the standard library. In the rest of the book, we'll want to use the standard library's.

```
Module NatPlayground.
```

All the types we have defined so far -- both "enumerated types" such as `day`, `bool`, and `bit` and tuple types such as `nybble` built from them -- are finite. The natural numbers, on the other hand, are an infinite set, so we'll need to use a slightly richer form of type declaration to represent them.

There are many representations of numbers to choose from. We are most familiar with decimal notation (base 10), using the digits 0 through 9, for example, to form the number 123. You may have encountered hexadecimal notation (base 16), in which the same number is represented as 7B, or octal (base 8), where it is 173, or binary (base 2), where it is 1111011. Using an enumerated type to represent digits, we could use any of these as our representation natural numbers. Indeed, there are circumstances where each of these choices would be useful.

The binary representation is valuable in computer hardware because the digits can be represented with just two distinct voltage levels, resulting in simple circuitry. Analogously, we wish here to choose a representation that

makes *proofs* simpler.

In fact, there is a representation of numbers that is even simpler than binary, namely unary (base 1), in which only a single digit is used (as one might do to count days in prison by scratching on the walls). To represent unary numbers with a Coq datatype, we use two constructors. The capital-letter `O` constructor represents zero. When the `S` constructor is applied to the representation of the natural number n, the result is the representation of n+1, where `S` stands for "successor" (or "scratch" if one is in prison). Here is the complete datatype definition.

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

With this definition, 0 is represented by `O`, 1 by `S O`, 2 by `S (S O)`, and so on.

Informally, the clauses of the definition can be read:

- `O` is a natural number (remember 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:

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

These conditions are the precise force of the `Inductive` declaration. They imply that the constructor expression `O`, the constructor expression `S O`, the constructor expression `S (S O)`, the constructor expression `S (S (S O))`, and so on all belong to the set `nat`, while other constructor expressions, 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
  | tick (foo : 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'`."

The following `End` command closes the current module, so `nat` will refer back to the type from the standard library. As mentioned earlier, it comes with special notation (as decimal numbers) unlike the above redefinition of `nat`.

```
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 decimal numerals can be used as an alternative to the "unary" notation defined by the constructors `S` and `O`. Coq prints numbers in decimal 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 functions such as `pred` and `minustwo`:

```
Check S : nat→nat.
Check pred : nat→nat.
Check minustwo : nat→nat.
```

These are all things that can be applied to a number to yield a number. However, there is a fundamental difference between `S` and the other two: functions like `pred` and `minustwo` are defined by giving *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 decimal 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.)

Now let's go on and define some more functions over numbers.

For most interesting computations involving numbers, simple 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. Such functions are introduced with the keyword `Fixpoint` instead of `Definition`.

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

We could define `oddb` by a similar `Fixpoint` declaration, but here is a simpler way:

```
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.
```

(You may 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 discuss 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).
(* ===> 5 : nat *)
```

The steps of simplification that Coq performs can be visualized as follows:

```
(*      plus 3 2
   i.e. 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 (S O))))
          by the first clause of the match
   i.e. 5  *)
```

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.

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, standard (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.
□
```

Again, we can make numerical expressions 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) : nat.
```

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

Note that these declarations 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 is a user-defined operation! Here is a function `eqb`, which tests natural numbers for `eq`uality, yielding a `b`oolean. Note the use of nested `match`es (we could also have used a simultaneous match, as we did in `minus`.)

```
Fixpoint eqb (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' ⇒ eqb n' m'
    end
  end.
```

Similarly, 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.
```

We'll be using these (especially `eqb`) a lot, so let's give them infix notations.

```
Notation "x =? y" := (eqb x y) (at level 70) : nat_scope.
Notation "x <=? y" := (leb x y) (at level 70) : nat_scope.

Example test_leb3': (4 <=? 2) = false.
Proof. simpl. reflexivity. Qed.
```

We now have two symbols that look like equality: `=` and `=?`. We'll have much more to say about the differences and similarities between them later. For now, the main thing to notice is that `x = y` is a logical *claim* -- a "proposition" -- that we can try to prove, while `x =? y` is an *expression* whose value (either `true` or `false`) we can compute.

Exercise: 1 star, standard (ltb)

The `ltb` function tests natural numbers for `l`ess-`t`han, yielding a `b`oolean. Instead of making up a new `Fixpoint` for this one, define it in terms of a previously defined function. (It can be done with just one previously defined function, but you can use two if you want.)

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

Notation "x <? y" := (ltb x y) (at level 70) : nat_scope.

Example test_ltb1: (ltb 2 2) = false.
(* FILL IN HERE *) Admitted.
Example test_ltb2: (ltb 2 4) = true.
(* FILL IN HERE *) Admitted.
Example test_ltb3: (ltb 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. 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 the standard upside-down-A symbol.)

This is a good place to mention that `reflexivity` is a bit more powerful than we have acknowledged. 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 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. Note that we could have used another identifier instead of `n` in the `intros` clause, (though of course this might be confusing to human readers of the proof):

```
Theorem plus_O_n'' : ∀ n : nat, 0 + n = n.
```

```
Proof.
  intros m. reflexivity. Qed.
```

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 many 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

The following theorem is a bit more interesting than the ones 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.
```

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.

(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.)

Exercise: 1 star, standard (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!

The `Check` command can also be used to examine the statements of previously declared lemmas and theorems. The two examples below are lemmas about multiplication that are proved in the standard library. (We will see how to prove them ourselves in the next chapter.)

```
Check mult_n_O.
(* ===> forall n : nat, 0 = n * 0 *)

Check mult_n_Sm.
(* ===> forall n m : nat, n * m + n = n * S m *)
```

We can 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_n_0_m_0 : ∀ p q : nat,
  (p × 0) + (q × 0) = 0.
Proof.
  intros p q.
  rewrite <- mult_n_O.
  rewrite <- mult_n_O.
  reflexivity. Qed.
```

Exercise: 1 star, standard (mult_n_1)

Use those two lemmas about multiplication that we just checked to prove the following theorem. Hint: recall that `1` is `S O`.

```
Theorem mult_n_1 : ∀ p : nat,
  p × 1 = p.
Proof.
  (* FILL IN HERE *) Admitted.
□
```

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,
  (n + 1) =? 0 = false.
Proof.
  intros n.
  simpl. (* does nothing! *)
Abort.
```

The reason for this is that the definitions of both `eqb` 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 `eqb` 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 `(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` represents, we can calculate that, at least, it will begin with one `S`, and this is enough to calculate that, again, `(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,
  (n + 1) =? 0 = false.
Proof.
  intros n. destruct n as [| n'] eqn:E.
  - 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.

In each subgoal, Coq remembers the assumption about `n` that is relevant for this subgoal -- either `n = 0` or `n = S n'` for some n'. The `eqn:E` annotation tells `destruct` to give the name `E` to this equation. Leaving off the `eqn:E` annotation causes Coq to elide these assumptions in the subgoals. This slightly streamlines proofs where the assumptions are not explicitly used, but it is better practice to keep them for the sake of documentation, as they can help keep you oriented when working with the subgoals.

The `-` signs on the second and third lines are called *bullets*, and they mark the parts of the proof that correspond to the two generated subgoals. The part of the proof script that comes after a bullet is the entire proof for the corresponding 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 second one simplifies `(S n' + 1) =? 0` to `false` by first rewriting `(S n' + 1)` to `S (n' + 1)`, then unfolding `eqb`, and then simplifying the `match`.

Marking cases with bullets is 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, improving readability. 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 -- e.g., 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 a single line. Good style lies somewhere in the middle. One reasonable guideline 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 eqn:E.
  - 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. 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 eqn:Eb.
  - destruct c eqn:Ec.
    + reflexivity.
    + reflexivity.
  - destruct c eqn:Ec.
    + 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) or any repetition of a bullet symbol (e.g. `--` or `***`) as a bullet. We can also enclose sub-proofs in curly braces:

```
Theorem andb_commutative' : ∀ b c, andb b c = andb c b.
Proof.
  intros b c. destruct b eqn:Eb.
```

```
    { destruct c eqn:Ec.
      { reflexivity. }
      { reflexivity. } }
    { destruct c eqn:Ec.
      { 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. The choice of braces, bullets, or a combination of the two is purely a matter of taste.

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

Exercise: 2 stars, standard (andb_true_elim2)

Prove the following claim, marking cases (and subcases) with bullets when you use `destruct`. Hint: delay introducing the hypothesis until after you have an opportunity to simplify it.

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

□

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] eqn:E.
```

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. (You'll also note one downside of this shorthand: we lose the equation recording the assumption we are making in each subgoal, which we previously got from the `eqn:E` annotation.)

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

If there are no constructor arguments that need names, we can just write `[]` to get the case analysis.

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

Exercise: 1 star, standard (zero_nbeq_plus_1)

```
Theorem zero_nbeq_plus_1 : ∀ n : 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 product 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`, `42`, 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."

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, standard, optional (decreasing)

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. (If you choose to turn in this optional exercise as part of a homework assignment, make sure you comment out your solution so that it doesn't cause Coq to reject the whole file!)

```
(* FILL IN HERE *)
```
□

More Exercises

Exercise: 1 star, standard (identity_fn_applied_twice)

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.
```
□

Exercise: 1 star, standard (negation_fn_applied_twice)

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`.

```
(* FILL IN HERE *)

(* Do not modify the following line: *)
Definition manual_grade_for_negation_fn_applied_twice : option (nat×string) := None.
```

(The last definition is used by the autograder.) □

Exercise: 3 stars, standard, 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, standard (binary)

We can generalize our unary representation of natural numbers to the more efficient binary representation by treating a binary number as a sequence of constructors B_0 and B_1 (representing 0s and 1s), terminated by a `Z`. For comparison, in the unary representation, a number is a sequence of `S` constructors terminated by an `O`.

For example:

decimal	binary	unary
0	Z	O
1	B_1 Z	S O
2	B_0 (B_1 Z)	S (S O)
3	B_1 (B_1 Z)	S (S (S O))
4	B_0 (B_0 (B_1 Z))	S (S (S (S O)))
5	B_1 (B_0 (B_1 Z))	S (S (S (S (S O))))
6	B_0 (B_1 (B_1 Z))	S (S (S (S (S (S O)))))
7	B_1 (B_1 (B_1 Z))	S (S (S (S (S (S (S O))))))
8	B_0 (B_0 (B_0 (B_1 Z)))	S (S (S (S (S (S (S (S O)))))))

Note that the low-order bit is on the left and the high-order bit is on the right -- the opposite of the way binary numbers are usually written. This choice makes them easier to manipulate.

```
Inductive bin : Type :=
  | Z
  | B₀ (n : bin)
  | B₁ (n : bin).
```

Complete the definitions below of an increment function `incr` for binary numbers, and a function `bin_to_nat` to convert binary numbers to unary numbers.

```
Fixpoint incr (m:bin) : bin
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *). Admitted.

Fixpoint bin_to_nat (m:bin) : nat
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *). Admitted.
```

The following "unit tests" of your increment and binary-to-unary functions should pass after you have defined those functions correctly. Of course, unit tests don't fully demonstrate the correctness of your functions! We'll return to that thought at the end of the next chapter.

```
Example test_bin_incr1 : (incr (B1 Z)) = B0 (B1 Z).
(* FILL IN HERE *) Admitted.

Example test_bin_incr2 : (incr (B0 (B1 Z))) = B1 (B1 Z).
(* FILL IN HERE *) Admitted.

Example test_bin_incr3 : (incr (B1 (B1 Z))) = B0 (B0 (B1 Z)).
(* FILL IN HERE *) Admitted.

Example test_bin_incr4 : bin_to_nat (B0 (B1 Z)) = 2.
(* FILL IN HERE *) Admitted.

Example test_bin_incr5 :
        bin_to_nat (incr (B1 Z)) = 1 + bin_to_nat (B1 Z).
(* FILL IN HERE *) Admitted.

Example test_bin_incr6 :
        bin_to_nat (incr (incr (B1 Z))) = 2 + bin_to_nat (B1 Z).
(* FILL IN HERE *) Admitted.
□
```

Testing Your Solutions

Each SF chapter comes with a test file containing scripts that check whether you have solved the required exercises. If you're using SF as part of a course, your instructors will likely be running these test files to autograde your solutions. You can also use these test files, if you like, to make sure you haven't missed anything.

Important: This step is *optional*: if you've completed all the non-optional exercises and Coq accepts your answers, this already shows that you are in good shape.

The test file for this chapter is `BasicsTest.v`. To run it, make sure you have saved `Basics.v` to disk. Then do this:

```
coqc -Q . LF Basics.v
coqc -Q . LF BasicsTest.v
```

If you accidentally deleted an exercise or changed its name, then `make BasicsTest.vo` will fail with an error that tells you the name of the missing exercise. Otherwise, you will get a lot of useful output:

- First will be all the output produced by `Basics.v` itself. At the end of that you will see `COQC BasicsTest.v`.
- Second, for each required exercise, there is a report that tells you its point value (the number of stars or some fraction thereof if there are multiple parts to the exercise), whether its type is ok, and what assumptions it relies upon.

  If the *type* is not `ok`, it means you proved the wrong thing: most likely, you accidentally modified the theorem statement while you were proving it. The autograder won't give you any points for that, so make sure to correct the theorem.

  The *assumptions* are any unproved theorems which your solution relies upon. "Closed under the global context" is a fancy way of saying "none": you have solved the exercise. (Hooray!) On the other hand, a list of axioms means you haven't fully solved the exercise. (But see below regarding "Allowed Axioms.") If the exercise name itself is in the list, that means you haven't solved it; probably you have `Admitted` it.
- Third, you will see the maximum number of points in standard and advanced versions of the assignment. That number is based on the number of stars in the non-optional exercises.
- Fourth, you will see a list of "Allowed Axioms". These are unproved theorems that your solution is permitted to depend upon. You'll probably see something about `functional_extensionality` for this chapter; we'll cover what that means in a later chapter.
- Finally, you will see a summary of whether you have solved each exercise. Note that summary does not include the critical information of whether the type is ok (that is, whether you accidentally changed the theorem statement): you have to look above for that information.

Exercises that are manually graded will also show up in the output. But since they have to be graded by a human, the test script won't be able to tell you much about them.

```
(* 2020-09-09 20:51 *)
```