Martin's comment on type annotations:
the type annotations are really similar to
haskell except that haskell doesn't require you
to write them explicitly
forall X … <=> a -> …
<=> means equivalent
so forall X is basically like an additional type
argument
you need to pass type annotations
you can't cut corners in coq from lambda cal
in coq they're explicit and this is the whole
picture
when talking about polymorphism you do
need type annotations
you're basically transforming types using the
same formalism
so you can transform values and types (two
levels) but with the same formalism
(nil nat) is a type application

in Haskell:
:k (:kind) is applied on types
and gives you the type annotation
so you can only apply it on
type constructors and not
data constructors!!

```
*Basics> :k GenList
GenList :: * -> *
*Basics> :k Maybe
Maybe :: * -> *
*Basics> :k (->)
(->) :: TYPE q -> TYPE r -> *
```
WHAAAT!!!!!!

```
*Basics> :k Eq
Eq :: * -> Constraint
*Basics> :k (,)
(,) :: * -> * -> *
```


POLY

POLYMORPHISM AND HIGHER-ORDER FUNCTIONS

I led!!!

```
(* Suppress some annoying warnings from Coq: *)
Set Warnings "-notation-overridden,-parsing".
Require Export Lists.
```

FULL: Polymorphism

In this chapter we continue our development of basic concepts of functional programming. The critical new ideas are *polymorphism* (abstracting functions over the types of the data they manipulate) and *higher-order functions* (treating functions as data). We begin with polymorphism.

Polymorphic Lists

For the last couple of chapters, we've been working just with lists of numbers. Obviously, interesting programs also need to be able to manipulate lists with elements from other types — lists of strings, lists of booleans, lists of lists, etc. We *could* just define a new inductive datatype for each of these, for example…

```
Inductive boollist : Type :=
  | bool_nil : boollist
  | bool_cons : bool → boollist → boollist.
```

… but this would quickly become tedious, partly because we have to make up different constructor names for each datatype, but mostly because we would also need to define new versions of all our list manipulating functions (`length`, `rev`, etc.) for each new datatype definition.

To avoid all this repetition, Coq supports *polymorphic* inductive type definitions. For example, here is a *polymorphic list* datatype.

```
Inductive list (X:Type) : Type :=
  | nil : list X
  | cons : X → list X → list X.
```

This is exactly like the definition of `natlist` from the previous chapter, except that the `nat` argument to the `cons` constructor has been replaced by an arbitrary type `X`, a binding for `X` has been added to the header, and the occurrences of `natlist` in the types of the constructors have been replaced by `list X`. (We can re-use the constructor names `nil` and `cons` because the earlier definition of `natlist` was inside of a `Module` definition that is now out of scope.)

it's a function that takes a type and r eturns a list which is an inductive definition

What sort of thing is `list` itself? One good way to think about it is that `list` is a *function* from `Types` to `Inductive` definitions; or, to put it another way, `list` is a function from `Types` to `Types`. For any particular type `X`, the type `list X` is an `Inductively` defined set of lists whose elements are of type `X`.

```
Check list.
(* ===> list : Type -> Type *)
```

The parameter `X` in the definition of `list` becomes a parameter to the constructors `nil` and `cons` — that is, `nil` and `cons` are now polymorphic constructors, that need to be supplied with the type of the list they are building. As an example, `nil nat` constructs the empty list of type `nat`.

```
Check (nil nat).
(* ===> nil nat : list nat *)
```

Similarly, `cons nat` adds an element of type `nat` to a list of type `list nat`. Here is an example of forming a list containing just the natural number 3.

```
Check (cons nat 3 (nil nat)).
(* ===> cons nat 3 (nil nat) : list nat *)
```

What might the type of `nil` be? We can read off the type `list X` from the definition, but this omits the binding for `X` which is the parameter to `list`. `Type → list X` does not explain the meaning of `X`. `(X : Type) → list X` comes closer. Coq's notation for this situation is `∀X : Type, list X`.

```
Check nil.
(* ===> nil : forall X : Type, list X *)
```

X is a type var that is treated as a parameter!

Similarly, the type of `cons` from the definition looks like `X → list X → list X`, but using this convention to explain the meaning of `X` results in the type `∀X, X → list X → list X`.

```
Check cons.
(* ===> cons : forall X : Type, X -> list X -> list X *)
```

(Side note on notation: In .v files, the "forall" quantifier is spelled out in letters. In the generated HTML files and in the way various IDEs show .v files (with certain settings of their display controls), ∀is usually typeset as the usual mathematical "upside down A," but you'll still see the spelled-out "forall" in a few places. This is just a quirk of typesetting: there is no difference in meaning.)

Having to supply a type argument for each use of a list constructor may seem an awkward burden, but we will soon see ways of reducing that burden.

```
Check (cons nat 2 (cons nat 1 (nil nat))).
```

(We've written `nil` and `cons` explicitly here because we haven't yet defined the `[ ]` and `::` notations for the new version of lists. We'll do that in a bit.)

We can now go back and make polymorphic versions of all the list-processing functions that we wrote before. Here is `repeat`, for example:

```
Fixpoint repeat (X : Type) (x : X) (count : nat) : list X :=
  match count with
  | 0 ⇒ nil X
  | S count' ⇒ cons X x (repeat X x count')
  end.
```

As with `nil` and `cons`, we can use `repeat` by applying it first to a type and then to an element of this type (and a number):

```
Example test_repeat1 :
  repeat nat 4 2 = cons nat 4 (cons nat 4 (nil nat)).
Proof. reflexivity. Qed.
```

To use `repeat` to build other kinds of lists, we simply instantiate it with an appropriate type parameter:

```
Example test_repeat2 :
  repeat bool false 1 = cons bool false (nil bool).
Proof. reflexivity. Qed.

Module MumbleGrumble.
```

Exercise: 2 stars (mumble_grumble)

Consider the following two inductively defined types.

```
Inductive mumble : Type :=
  | a : mumble
  | b : mumble → nat → mumble
  | c : mumble.

Inductive grumble (X:Type) : Type :=
  | d : mumble → grumble X
  | e : X → grumble X.
```

Which of the following are well-typed elements of `grumble X` for some type `X`?

- `d (b a 5)` type error!
- `d mumble (b a 5)`
- `d bool (b a 5)`
- `e bool true`
- `e mumble (b c 0)`
- `e bool (b c 0)` type error!
- `c`

```
  (* FILL IN HERE *)
□
  End MumbleGrumble.
```

Type Annotation Inference

Let's write the definition of `repeat` again, but this time we won't specify the types of any of the arguments. Will Coq still accept it?

```
Fixpoint repeat' X x count : list X :=
  match count with
  | 0 ⇒ nil X
  | S count' ⇒ cons X x (repeat' X x count')
  end.
```

Indeed it will. Let's see what type Coq has assigned to `repeat'`:

```
Check repeat'.
(* ===> forall X : Type, X -> nat -> list X *)
Check repeat.
(* ===> forall X : Type, X -> nat -> list X *)
```

It has exactly the same type as `repeat`. Coq was able to use *type inference* to deduce what the types of `X`, `x`, and `count` must be, based on how they are used. For example, since `X` is used as an argument to `cons`, it must be a `Type`, since `cons` expects a `Type` as its first argument; matching `count` with `0` and `S` means it must be a `nat`; and so on.

This powerful facility means we don't always have to write explicit type annotations everywhere, although explicit type annotations are still quite useful as documentation and sanity checks, so we will continue to use them most of the time. You should try to find a balance in your own code between too many type annotations (which can clutter and distract) and too few (which forces readers to perform type inference in their heads in order to understand your code).

Type Argument Synthesis

To use a polymorphic function, we need to pass it one or more types in addition to its other arguments. For example, the recursive call in the body of the `repeat` function above must pass along the type `X`. But since the second argument to `repeat` is an element of `X`, it seems entirely obvious that the first argument can only be `X` — why should we have to write it explicitly?

Fortunately, Coq permits us to avoid this kind of redundancy. In place of any type argument we can write the "implicit argument" _, which can be read as "Please try to figure out for yourself what belongs here." More precisely, when Coq encounters a _, it will attempt to *unify* all locally available information — the type of the function being applied, the types of the other arguments, and the type expected by the context in which the application appears — to determine what concrete type should replace the _.

This may sound similar to type annotation inference — indeed, the two procedures rely on the same underlying mechanisms. Instead of simply omitting the types of some arguments to a function, like

```
repeat' X x count : list X :=
```

we can also replace the types with _

```
repeat' (X : _) (x : _) (count : _) : list X :=
```

to tell Coq to attempt to infer the missing information.

Using implicit arguments, the `repeat` function can be written like this:

```
Fixpoint repeat'' X x count : list X :=
  match count with
  | 0 ⇒ nil _
  | S count' ⇒ cons _ x (repeat'' _ x count')
  end.
```

In this instance, we don't save much by writing _ instead of `X`. But in many cases the difference in both keystrokes and readability is nontrivial. For example, suppose we want to write down a list containing the numbers 1, 2, and 3. Instead of writing this...

```
Definition list123 :=
  cons nat 1 (cons nat 2 (cons nat 3 (nil nat))).
```

...we can use argument synthesis to write this:

```
Definition list123' :=
  cons _ 1 (cons _ 2 (cons _ 3 (nil _))).
```

Implicit Arguments

We can go further and even avoid writing _'s in most cases by telling Coq *always* to infer the type argument(s) of a given function.

The `Arguments` directive specifies the name of the function (or constructor) and then lists its argument names, with curly braces around any arguments to be treated as implicit. (If some arguments of a definition don't have a name, as is often the case for constructors, they can be marked with a wildcard pattern _.)

```
Arguments nil {X}.
Arguments cons {X} _ _.
Arguments repeat {X} x count.
```

Now, we don't have to supply type arguments at all:

```
Definition list123'' := cons 1 (cons 2 (cons 3 nil)).
```

Alternatively, we can declare an argument to be implicit when defining the function itself, by surrounding it in curly braces instead of parens. For example:

```
Fixpoint repeat''' {X : Type} (x : X) (count : nat) : list X :=
  match count with
  | 0 ⇒ nil
```

```
  | S count' ⇒ cons x (repeat''' x count')
  end.
```

(Note that we didn't even have to provide a type argument to the recursive call to `repeat'''`; indeed, it would be invalid to provide one!)

We will use the latter style whenever possible, but we will continue to use explicit `Argument` declarations for `Inductive` constructors. The reason for this is that marking the parameter of an inductive type as implicit causes it to become implicit for the type itself, not just for its constructors. For instance, consider the following alternative definition of the `list` type:

```
Inductive list' {X:Type} : Type :=
  | nil' : list'
  | cons' : X → list' → list'.
```

Because `X` is declared as implicit for the *entire* inductive definition including `list'` itself, we now have to write just `list'` whether we are talking about lists of numbers or booleans or anything else, rather than `list' nat` or `list' bool` or whatever; this is a step too far.

Let's finish by re-implementing a few other standard list functions on our new polymorphic lists...

```
Fixpoint app {X : Type} (l1 l2 : list X)
             : (list X) :=
  match l1 with
  | nil ⇒ l2
  | cons h t ⇒ cons h (app t l2)
  end.

Fixpoint rev {X:Type} (l:list X) : list X :=
  match l with
  | nil ⇒ nil
  | cons h t ⇒ app (rev t) (cons h nil)
  end.

Fixpoint length {X : Type} (l : list X) : nat :=
  match l with
  | nil ⇒ 0
  | cons _ l' ⇒ S (length l')
  end.

Example test_rev1 :
  rev (cons 1 (cons 2 nil)) = (cons 2 (cons 1 nil)).
Proof. reflexivity. Qed.

Example test_rev2:
  rev (cons true nil) = cons true nil.
Proof. reflexivity. Qed.

Example test_length1: length (cons 1 (cons 2 (cons 3 nil))) = 3.
Proof. reflexivity. Qed.
```

Supplying Type Arguments Explicitly

One small problem with declaring arguments `Implicit` is that, occasionally, Coq does not have enough local information to determine a type argument; in such cases, we need to tell Coq that we want to give the argument explicitly just this time. For example, suppose we write this:

```
Fail Definition mynil := nil.
```

(The `Fail` qualifier that appears before `Definition` can be used with *any* command, and is used to ensure that that command indeed fails when executed. If the command does fail, Coq prints the corresponding error message, but continues processing the rest of the file.)

Here, Coq gives us an error because it doesn't know what type argument to supply to `nil`. We can help it by providing an explicit type declaration (so that Coq has more information available when it gets to the "application" of `nil`):

```
Definition mynil : list nat := nil.
```

Alternatively, we can force the implicit arguments to be explicit by prefixing the function name with `@`.

```
Check @nil.

Definition mynil' := @nil nat.
```

Using argument synthesis and implicit arguments, we can define convenient notation for lists, as before. Since we have made the constructor type arguments implicit, Coq will know to automatically infer these when we use the notations.

```
Notation "x :: y" := (cons x y)
                     (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y []) ..).
Notation "x ++ y" := (app x y)
                     (at level 60, right associativity).
```

Now lists can be written just the way we'd hope:

```
Definition list123''' := [1; 2; 3].
```

Exercises

Exercise: 2 stars, optional (poly_exercises)

Here are a few simple exercises, just like ones in the `Lists` chapter, for practice with polymorphism. Complete the proofs below.

```
Theorem app_nil_r : ∀ (X:Type), ∀ l:list X,
  l ++ [] = l.
Proof.
  (* FILL IN HERE *) Admitted.

Theorem app_assoc : ∀ A (l m n:list A),
  l ++ m ++ n = (l ++ m) ++ n.
```

```
Proof.
  (* FILL IN HERE *) Admitted.

Lemma app_length : ∀ (X:Type) (l1 l2 : list X),
  length (l1 ++ l2) = length l1 + length l2.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars, optional (more_poly_exercises)

Here are some slightly more interesting ones...

```
Theorem rev_app_distr: ∀ X (l1 l2 : list X),
  rev (l1 ++ l2) = rev l2 ++ rev l1.
Proof.
  (* FILL IN HERE *) Admitted.

Theorem rev_involutive : ∀ X : Type, ∀ l : list X,
  rev (rev l) = l.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Polymorphic Pairs

Following the same pattern, the type definition we gave in the last chapter for pairs of numbers can be generalized to *polymorphic pairs*, often called *products*:

```
Inductive prod (X Y : Type) : Type :=
| pair : X → Y → prod X Y.

Arguments pair {X} {Y} _ _.
```

As with lists, we make the type arguments implicit and define the familiar concrete notation.

```
Notation "( x , y )" := (pair x y).
```

We can also use the `Notation` mechanism to define the standard notation for product *types*:

```
Notation "X * Y" := (prod X Y) : type_scope.
```

(The annotation `: type_scope` tells Coq that this abbreviation should only be used when parsing types. This avoids a clash with the multiplication symbol.)

It is easy at first to get `(x,y)` and `X*Y` confused. Remember that `(x,y)` is a *value* built from two other values, while `X*Y` is a *type* built from two other types. If `x` has type `X` and `y` has type `Y`, then `(x,y)` has type `X*Y`.

The first and second projection functions now look pretty much as they would in any functional programming language.

```
Definition fst {X Y : Type} (p : X * Y) : X :=
  match p with
```

```
    | (x, y) ⇒ x
    end.

  Definition snd {X Y : Type} (p : X * Y) : Y :=
    match p with
    | (x, y) ⇒ y
    end.
```

The following function takes two lists and combines them into a list of pairs. In other functional languages, it is often called `zip`; we call it `combine` for consistency with Coq's standard library.

```
Fixpoint combine {X Y : Type} (lx : list X) (ly : list Y)
           : list (X*Y) :=
  match lx, ly with
  | [], _ ⇒ []
  | _, [] ⇒ []
  | x :: tx, y :: ty ⇒ (x, y) :: (combine tx ty)
  end.
```

Exercise: 1 star, optional (combine_checks)

Try answering the following questions on paper and checking your answers in Coq:

- What is the type of `combine` (i.e., what does `Check @combine` print?)
- What does

  ```
  Compute (combine [1;2] [false;false;true;true]).
  ```

  print?

□

Exercise: 2 stars, recommended (split)

The function `split` is the right inverse of `combine`: it takes a list of pairs and returns a pair of lists. In many functional languages, it is called `unzip`.

Fill in the definition of `split` below. Make sure it passes the given unit test.

```
Fixpoint split {X Y : Type} (l : list (X*Y))
               : (list X) * (list Y)
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_split:
  split [(1,false);(2,false)] = ([1;2],[false;false]).
Proof.
(* FILL IN HERE *) Admitted.
```

□

Polymorphic Options

One last polymorphic type for now: *polymorphic options*, which generalize `natoption` from the previous chapter:

```
Inductive option (X:Type) : Type :=
  | Some : X → option X
```

```
  | None : option X.

Arguments Some {X} _.
Arguments None {X}.
```

We can now rewrite the `nth_error` function so that it works with any type of lists.

```
Fixpoint nth_error {X : Type} (l : list X) (n : nat)
                   : option X :=
  match l with
  | [] ⇒ None
  | a :: l' ⇒ if beq_nat n O then Some a else nth_error l' (pred
n)
  end.

Example test_nth_error1 : nth_error [4;5;6;7] 0 = Some 4.
+
Example test_nth_error2 : nth_error [[1];[2]] 1 = Some [2].
+
Example test_nth_error3 : nth_error [true] 2 = None.
+
```

Exercise: 1 star, optional (hd_error_poly)

Complete the definition of a polymorphic version of the `hd_error` function from the last chapter. Be sure that it passes the unit tests below.

```
Definition hd_error {X : Type} (l : list X) : option X
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
```

Once again, to force the implicit arguments to be explicit, we can use `@` before the name of the function.

```
Check @hd_error.

Example test_hd_error1 : hd_error [1;2] = Some 1.
 (* FILL IN HERE *) Admitted.
Example test_hd_error2 : hd_error [[1];[2]] = Some [1].
 (* FILL IN HERE *) Admitted.
```
☐

Functions as Data

Like many other modern programming languages — including all functional languages (ML, Haskell, Scheme, Scala, Clojure, etc.) — Coq treats functions as first-class citizens, allowing them to be passed as arguments to other functions, returned as results, stored in data structures, etc.

Higher-Order Functions

Functions that manipulate other functions are often called *higher-order* functions. Here's a simple one:

```
Definition doit3times {X:Type} (f:X→X) (n:X) : X :=
  f (f (f n)).
```

The argument `f` here is itself a function (from `X` to `X`); the body of `doit3times` applies `f` three times to some value `n`.

```
Check @doit3times.
(* ===> doit3times : forall X : Type, (X -> X) -> X -> X *)

Example test_doit3times: doit3times minustwo 9 = 3.
Proof. reflexivity. Qed.

Example test_doit3times': doit3times negb true = false.
Proof. reflexivity. Qed.
```

Filter

Here is a more useful higher-order function, taking a list of `X`s and a *predicate* on `X` (a function from `X` to `bool`) and "filtering" the list, returning a new list containing just those elements for which the predicate returns `true`.

```
Fixpoint filter {X:Type} (test: X→bool) (l:list X)
                : (list X) :=
  match l with
  | [] ⇒ []
  | h :: t ⇒ if test h then h :: (filter test t)
                        else filter test t
  end.
```

For example, if we apply `filter` to the predicate `evenb` and a list of numbers `l`, it returns a list containing just the even members of `l`.

```
Example test_filter1: filter evenb [1;2;3;4] = [2;4].
Proof. reflexivity. Qed.

Definition length_is_1 {X : Type} (l : list X) : bool :=
  beq_nat (length l) 1.

Example test_filter2:
    filter length_is_1
           [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
  = [ [3]; [4]; [8] ].
Proof. reflexivity. Qed.
```

We can use `filter` to give a concise version of the `countoddmembers` function from the Lists chapter.

```
Definition countoddmembers' (l:list nat) : nat :=
  length (filter oddb l).

Example test_countoddmembers'1: countoddmembers' [1;0;3;1;4;5] =
4.
Proof. reflexivity. Qed.
Example test_countoddmembers'2: countoddmembers' [0;2;4] = 0.
Proof. reflexivity. Qed.
Example test_countoddmembers'3: countoddmembers' nil = 0.
Proof. reflexivity. Qed.
```

Anonymous Functions

It is arguably a little sad, in the example just above, to be forced to define the function `length_is_1` and give it a name just to be able to pass it as an argument to `filter`, since we will probably never use it again. Moreover, this is not an isolated example: when using higher-order functions, we often want to pass as arguments "one-off" functions that we will never use again; having to give each of these functions a name would be tedious.

Fortunately, there is a better way. We can construct a function "on the fly" without declaring it at the top level or giving it a name.

```
Example test_anon_fun':
  doit3times (fun n ⇒ n * n) 2 = 256.
Proof. reflexivity. Qed.
```

The expression `(fun n ⇒ n * n)` can be read as "the function that, given a number `n`, yields `n * n`."

Here is the `filter` example, rewritten to use an anonymous function.

```
Example test_filter2':
    filter (fun l ⇒ beq_nat (length l) 1)
           [ [1; 2]; [3]; [4]; [5;6;7]; []; [8] ]
  = [ [3]; [4]; [8] ].
Proof. reflexivity. Qed.
```

Exercise: 2 stars (filter_even_gt$_7$)

Use `filter` (instead of `Fixpoint`) to write a Coq function `filter_even_gt`$_7$ that takes a list of natural numbers as input and returns a list of just those that are even and greater than 7.

```
Definition filter_even_gt7 (l : list nat) : list nat
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_filter_even_gt7_1 :
  filter_even_gt7 [1;2;6;9;10;3;12;8] = [10;12;8].
 (* FILL IN HERE *) Admitted.

Example test_filter_even_gt7_2 :
  filter_even_gt7 [5;2;6;19;129] = [].
 (* FILL IN HERE *) Admitted.
```

☐

Exercise: 3 stars (partition)

Use `filter` to write a Coq function `partition`:

```
partition : ∀ X : Type,
              (X → bool) → list X → list X * list X
```

Given a set X, a test function of type `X → bool` and a `list X`, `partition` should return a pair of lists. The first member of the pair is the sublist of the original list containing the elements that satisfy the test, and the second is the sublist containing those that fail the test. The order of elements in the two sublists should be the same as their order in the original list.

```
Definition partition {X : Type}
                     (test : X → bool)
                     (l : list X)
                   : list X * list X
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_partition1: partition oddb [1;2;3;4;5] = ([1;3;5],
[2;4]).
(* FILL IN HERE *) Admitted.
Example test_partition2: partition (fun x ⇒ false) [5;9;0] = ([],
[5;9;0]).
(* FILL IN HERE *) Admitted.
```

☐

Map

Another handy higher-order function is called `map`.

```
Fixpoint map {X Y:Type} (f:X→Y) (l:list X) : (list Y) :=
  match l with
  | [] ⇒ []
  | h :: t ⇒ (f h) :: (map f t)
  end.
```

It takes a function `f` and a list `l = [n1, n2, n3, ...]` and returns the list `[f n1, f n2, f n3,...]`, where `f` has been applied to each element of `l` in turn. For example:

```
Example test_map1: map (fun x ⇒ plus 3 x) [2;0;2] = [5;3;5].
Proof. reflexivity. Qed.
```

The element types of the input and output lists need not be the same, since `map` takes *two* type arguments, `X` and `Y`; it can thus be applied to a list of numbers and a function from numbers to booleans to yield a list of booleans:

```
Example test_map2:
  map oddb [2;1;2;5] = [false;true;false;true].
Proof. reflexivity. Qed.
```

It can even be applied to a list of numbers and a function from numbers to *lists* of booleans to yield a *list of lists* of booleans:

```
Example test_map3:
    map (fun n ⇒ [evenb n;oddb n]) [2;1;2;5]
  = [[true;false];[false;true];[true;false];[false;true]].
Proof. reflexivity. Qed.
```

Exercises

Exercise: 3 stars (map_rev)

Show that map and rev commute. You may need to define an auxiliary lemma.

```
Theorem map_rev : ∀ (X Y : Type) (f : X → Y) (l : list X),
  map f (rev l) = rev (map f l).
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars, recommended (flat_map)

The function map maps a list X to a list Y using a function of type X → Y. We can define a similar function, flat_map, which maps a list X to a list Y using a function f of type X → list Y. Your definition should work by 'flattening' the results of f, like so:

```
        flat_map (fun n ⇒ [n;n+1;n+2]) [1;5;10]
      = [1; 2; 3; 5; 6; 7; 10; 11; 12].
```

```
Fixpoint flat_map {X Y:Type} (f:X → list Y) (l:list X)
                   : (list Y)
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_flat_map1:
  flat_map (fun n ⇒ [n;n;n]) [1;5;4]
  = [1; 1; 1; 5; 5; 5; 4; 4; 4].
 (* FILL IN HERE *) Admitted.
```

☐

Lists are not the only inductive type that we can write a map function for. Here is the definition of map for the option type:

```
Definition option_map {X Y : Type} (f : X → Y) (xo : option X)
                      : option Y :=
  match xo with
    | None ⇒ None
    | Some x ⇒ Some (f x)
  end.
```

Exercise: 2 stars, optional (implicit_args)

The definitions and uses of filter and map use implicit arguments in many places. Replace the curly braces around the implicit arguments with parentheses, and then fill in explicit type parameters where necessary and use Coq to check that you've done so correctly. (This exercise is not to be turned in; it is probably easiest to do it on a *copy* of this file that you can throw away afterwards.) ☐

Fold

An even more powerful higher-order function is called fold. This function is the inspiration for the "reduce" operation that lies at the heart of Google's map/reduce distributed programming framework.

```
Fixpoint fold {X Y:Type} (f: X→Y→Y) (l:list X) (b:Y)
                         : Y :=
  match l with
  | nil ⇒ b
  | h :: t ⇒ f h (fold f t b)
  end.
```

Intuitively, the behavior of the `fold` operation is to insert a given binary operator `f` between every pair of elements in a given list. For example, `fold plus [1;2;3;4]` intuitively means `1+2+3+4`. To make this precise, we also need a "starting element" that serves as the initial second input to `f`. So, for example,

```
fold plus [1;2;3;4] 0
```

yields

```
1 + (2 + (3 + (4 + 0))).
```

Some more examples:

```
Check (fold andb).
(* ===> fold andb : list bool -> bool -> bool *)

Example fold_example1 :
  fold mult [1;2;3;4] 1 = 24.
+

Example fold_example2 :
  fold andb [true;true;false;true] true = false.
+

Example fold_example3 :
  fold app [[1];[];[2;3];[4]] [] = [1;2;3;4].
+
```

Exercise: 1 star, advanced (fold_types_different)

Observe that the type of `fold` is parameterized by *two* type variables, `X` and `Y`, and the parameter `f` is a binary operator that takes an `X` and a `Y` and returns a `Y`. Can you think of a situation where it would be useful for `X` and `Y` to be different?

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

☐

Functions That Construct Functions

Most of the higher-order functions we have talked about so far take functions as arguments. Let's look at some examples that involve *returning* functions as the results of other functions. To begin, here is a function that takes a value `x` (drawn from some type `X`) and returns a function from `nat` to `X` that yields `x` whenever it is called, ignoring its `nat` argument.

```
Definition constfun {X: Type} (x: X) : nat→X :=
  fun (k:nat) ⇒ x.

Definition ftrue := constfun true.
```

```
Example constfun_example1 : ftrue 0 = true.
+

Example constfun_example2 : (constfun 5) 99 = 5.
+
```

In fact, the multiple-argument functions we have already seen are also examples of passing functions as data. To see why, recall the type of `plus`.

```
Check plus.
(* ==> nat -> nat -> nat *)
```

Each → in this expression is actually a *binary* operator on types. This operator is *right-associative*, so the type of `plus` is really a shorthand for `nat → (nat → nat)` — i.e., it can be read as saying that "`plus` is a one-argument function that takes a `nat` and returns a one-argument function that takes another `nat` and returns a `nat`." In the examples above, we have always applied `plus` to both of its arguments at once, but if we like we can supply just the first. This is called *partial application*.

```
Definition plus3 := plus 3.
Check plus3.

Example test_plus3 : plus3 4 = 7.
Proof. reflexivity. Qed.
Example test_plus3' : doit3times plus3 0 = 9.
Proof. reflexivity. Qed.
Example test_plus3'' : doit3times (plus 3) 0 = 9.
Proof. reflexivity. Qed.
```

Additional Exercises

```
Module Exercises.
```

Exercise: 2 stars (fold_length)

Many common functions on lists can be implemented in terms of `fold`. For example, here is an alternative definition of `length`:

```
Definition fold_length {X : Type} (l : list X) : nat :=
  fold (fun _ n ⇒ S n) l 0.

Example test_fold_length1 : fold_length [4;7;0] = 3.
+
```

use reflexivity when you have no idea what to do!!!!!

Prove the correctness of `fold_length`.

```
Theorem fold_length_correct : ∀ X (l : list X),
  fold_length l = length l.
Proof.
(* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (fold_map)

We can also define `map` in terms of `fold`. Finish `fold_map` below.

```
Definition fold_map {X Y:Type} (f : X → Y) (l : list X) : list Y
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
```

Write down a theorem `fold_map_correct` in Coq stating that `fold_map` is correct, and prove it.

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

☐

Exercise: 2 stars, advanced (currying)

In Coq, a function `f : A → B → C` really has the type `A → (B → C)`. That is, if you give `f` a value of type `A`, it will give you function `f' : B → C`. If you then give `f'` a value of type `B`, it will return a value of type `C`. This allows for partial application, as in `plus3`. Processing a list of arguments with functions that return functions is called *currying*, in honor of the logician Haskell Curry.

Conversely, we can reinterpret the type `A → B → C` as `(A * B) → C`. This is called *uncurrying*. With an uncurried binary function, both arguments must be given at once as a pair; there is no partial application.

We can define currying as follows:

```
Definition prod_curry {X Y Z : Type}
  (f : X * Y → Z) (x : X) (y : Y) : Z := f (x, y).
```

As an exercise, define its inverse, `prod_uncurry`. Then prove the theorems below to show that the two are inverses.

```
Definition prod_uncurry {X Y Z : Type}
  (f : X → Y → Z) (p : X * Y) : Z
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
```

As a (trivial) example of the usefulness of currying, we can use it to shorten one of the examples that we saw above:

```
Example test_map1': map (plus 3) [2;0;2] = [5;3;5].
Proof. reflexivity. Qed.
```

Thought exercise: before running the following commands, can you calculate the types of `prod_curry` and `prod_uncurry`?

```
Check @prod_curry.
Check @prod_uncurry.

Theorem uncurry_curry : ∀ (X Y Z : Type)
                        (f : X → Y → Z)
                        x y,
  prod_curry (prod_uncurry f) x y = f x y.
```

```
Proof.
  (* FILL IN HERE *) Admitted.

Theorem curry_uncurry : ∀ (X Y Z : Type)
                          (f : (X * Y) → Z) (p : X * Y),
  prod_uncurry (prod_curry f) p = f p.
Proof.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 2 stars, advanced (nth_error_informal)

Recall the definition of the `nth_error` function:

```
Fixpoint nth_error {X : Type} (l : list X) (n : nat) : option X :=
  match l with
  | [] ⇒ None
  | a :: l' ⇒ if beq_nat n O then Some a else nth_error l' (pred n)
  end.
```

Write an informal proof of the following theorem:

```
∀ X n l, length l = n → @nth_error X l n = None
```

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

□

Exercise: 4 stars, advanced (church_numerals)

This exercise explores an alternative way of defining natural numbers, using the so-called *Church numerals*, named after mathematician Alonzo Church. We can represent a natural number `n` as a function that takes a function `f` as a parameter and returns `f` iterated `n` times.

```
Module Church.
Definition nat := ∀ X : Type, (X → X) → X → X.
```

Let's see how to write some numbers with this notation. Iterating a function once should be the same as just applying it. Thus:

```
Definition one : nat :=
  fun (X : Type) (f : X → X) (x : X) ⇒ f x.
```

Similarly, `two` should apply `f` twice to its argument:

```
Definition two : nat :=
  fun (X : Type) (f : X → X) (x : X) ⇒ f (f x).
```

Defining `zero` is somewhat trickier: how can we "apply a function zero times"? The answer is actually simple: just return the argument untouched.

```
Definition zero : nat :=
  fun (X : Type) (f : X → X) (x : X) ⇒ x.
```

More generally, a number `n` can be written as `fun X f x ⇒ f (f ... (f x) ...)`, with `n` occurrences of `f`. Notice in particular how the `doit3times` function we've defined previously is actually just the Church representation of `3`.

```
Definition three : nat := @doit3times.
```

Complete the definitions of the following functions. Make sure that the corresponding unit tests pass by proving them with `reflexivity`.

Successor of a natural number:

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

Example succ_1 : succ zero = one.
Proof. (* FILL IN HERE *) Admitted.

Example succ_2 : succ one = two.
Proof. (* FILL IN HERE *) Admitted.

Example succ_3 : succ two = three.
Proof. (* FILL IN HERE *) Admitted.
```

Addition of two natural numbers:

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

Example plus_1 : plus zero one = one.
Proof. (* FILL IN HERE *) Admitted.

Example plus_2 : plus two three = plus three two.
Proof. (* FILL IN HERE *) Admitted.

Example plus_3 :
  plus (plus two two) three = plus one (plus three three).
Proof. (* FILL IN HERE *) Admitted.
```

Multiplication:

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

Example mult_1 : mult one one = one.
Proof. (* FILL IN HERE *) Admitted.

Example mult_2 : mult zero (plus three three) = zero.
Proof. (* FILL IN HERE *) Admitted.

Example mult_3 : mult two three = plus three three.
Proof. (* FILL IN HERE *) Admitted.
```

Exponentiation:

(*Hint*: Polymorphism plays a crucial role here. However, choosing the right type to iterate over can be tricky. If you hit a "Universe inconsistency" error, try iterating over a different type: `nat` itself is usually problematic.)

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

Example exp_1 : exp two two = plus two two.
Proof. (* FILL IN HERE *) Admitted.

Example exp_2 : exp three two = plus (mult two (mult two two))
one.
Proof. (* FILL IN HERE *) Admitted.

Example exp_3 : exp three zero = one.
Proof. (* FILL IN HERE *) Admitted.

End Church.
```

☐

```
End Exercises.
```