Next

- More on recursion
- Higher-order functions
  - taking and returning functions
- Along the way, will see map and fold

max function

```ml
let max x y = if x < y then y else x;;
(* return max element of list l *)
let list_max l =
  let rec helper curr l =
    match l with
    | [] -> curr
    | h::t -> helper (max h curr) t
  in
  helper 0 l;;
```

concat function

```ml
let concat l =
  let rec helper curr l =
    match l with
    | [] -> curr
    | h::t -> helper (curr ^ h) t
  in
  helper "" l;;
```

What's the pattern?

```ml
let list_max l =
  let rec helper curr l =
    match l with
    | [] -> curr
    | h::t -> helper (max h curr) t
  in
  helper 0 l;;

let concat l =
  let rec helper curr l =
    match l with
    | [] -> curr
    | h::t -> helper (curr ^ h) t
  in
  helper "" l;;
```
fold, the general helper func!

(* to help us see the pattern: *)
let rec helper curr l =  
  match l with  
  | [] -> curr  
  | h::t -> helper (max h curr) t  
in helper 0 l;;

(* fold, the coolest function there is! *)
let rec fold f curr l =  
  match l with  
  | [] -> curr  
  | h::t -> fold f (f curr h) t;;

Examples of fold

let list_max = fold max 0;;

let concat = fold (^) "";;

let multiplier = fold (*) 1;;

Examples of fold

let fact n =  
multiplier (interval 1 n);;

Notice how all the recursion is buried inside two functions: interval and fold!
Examples of fold

let cons x y = y::x;;
let f = fold cons [[]];;
(* same as: let f l = fold cons [ ] l *)

Examples of fold

let cons x y = y::x;;
let f = fold cons [[]];;
(* same as: let f l = fold cons [ ] l *)

More recursion: interval

(* return a list that contains
the integers i through j inclusive *)
let rec interval i j =

interval

(* return a list that contains
the integers i through j inclusive *)
let rec interval i j =
if  i > j
then []
else i::(interval (i+1) j);;

interval function with init fn

(* return a list that contains the
elements f(i), f(i+1), ... f(j) *)
let rec interval_init i j f =

interval function with init fn

(* return a list that contains the
elements f(i), f(i+1), ... f(j) *)
let rec interval_init i j f =
if  i > j
then []
else (f i)::(interval_init (i+1) j f);;
interval function again

(* our regular interval function in terms of the one with the init function *)
let rec interval i j =

interval function again

(* our regular interval function in terms of the one with the init function *)
let rec interval i j =
  interval_init i j (fun x -> x);

Interval function yet again!

(* let's change the order of parameters... *)
let rec interval_init f i j =
  if  i > j
  then []
  else (f i)::(interval_init f (i+1) j);

(* now can use currying to get interval function! *)
let interval = interval_init (fun x -> x);

Function Currying

In general, these two are equivalent:

```
let f = fun x1 -> ... -> fun xn -> e

let f x1 ... xn = e
```

Multiple argument functions by returning a function that takes the next argument
- Named after a person (Haskell Curry)

Function Currying vs tuples

<table>
<thead>
<tr>
<th>Tuple version:</th>
<th>fn definition</th>
<th>fn call</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>let f (x1, ..., xn) = e</td>
<td>f (x1, ..., xn)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Curried version:</th>
<th>fn definition</th>
<th>fn call</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>let f x1 ... xn = e</td>
<td>f x1 ... xn</td>
</tr>
</tbody>
</table>

Consider the following:

```
let lt x y = x < y;
```

Could have done:  

```
let lt (x,y) = x<y;
```
- But then no “testers” possible

In general: Currying allows you to set just the first n params (where n smaller than the total number of params)
let rec map f l = match l with [] -> [] | h::t -> (f h)::(map f t);

let incr x = x+1;;
let map_incr = map incr;;
map_incr (interval (-10) 10);;

let compose f1 f2 = fun x -> (f1 (f2 x));;
(* another way of writing it *)
let compose f1 f2 x = f1 (f2 x);;

let map_incr_2 = compose map_incr map_incr;;
map_incr_2 (interval (-10) 10);;

let map_incr_3 = compose map_incr map_incr_2;;
map_incr_3 (interval (-10) 10);;

let map_incr_3_pos = compose pos_filter map_incr_3;;
map_incr_3_pos (interval (-10) 10);;
(compose map_incr_3_pos_filter) (interval (-10) 10);;
Higher-order functions!

```ocaml
let map_incr_2 = compose map_incr map_incr;;
map_incr_2 (interval (-10) 10);

let map_incr_3 = compose map_incr map_incr_2;;
map_incr_3 (interval (-10) 10);

let map_incr_3_pos = compose pos_filter map_incr_3;;
map_incr_3_pos (interval (-10) 10);

(compose map_incr_3 pos_filter) (interval (-10) 10);
```

Instead of manipulating lists, we are manipulating the list manipulators!

Exercise 1

```ocaml
let rec filter f l =
  match l with
  | [] -> []
  | h::t -> let t' = filter f t in
            if f h then h::t' else t'

let neg f x = not (f x)
let partition f l = (filter f l, filter (neg f) l)
```

This implementation is not ideal, since it unnecessarily processes the list twice. Rewrite partition so that it is a single call to fold_left, so the input list is processed only once. Recall:

```ocaml
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
let fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
let partition f l =
  let fold_fn (pass,passnot) x =
    if f x then (pass@[x], passnot)
    else (pass, passnot@[x])
  in
  List.fold_left fold_fn ([],[]) l;;
```

Exercise 2

```ocaml
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
val map : ('a -> 'b) -> 'a list -> 'b list
```

Implement map using fold:

```ocaml
let map f l =
  List.fold_left (fun acc x -> acc@[f x]) [] l
```
Benefits of higher-order functions

Identify common computation “patterns”
- **Iterate** a function over a set, list, tree ...
- **Accumulate** some value over a collection
Pull out (factor) “common” code:
- Computation Patterns
- Re-use in many different situations

Funcs taking/returning funcs

Higher-order funcs enable modular code
- Each part only needs local information

Different way of thinking

“Free your mind”
- Morpheus

- Different way of thinking about computation
- Manipulate the manipulators