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

Tuple version:
let f (x1,...,xn) = e

Curried version:
let f x1 ... xn = e

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)

map

(* return the list containing f(e) for each element e of l *)
let rec map f l =

(* return the list containing f(e) for each element e of l *)
let rec map f l =
  match l with
  | []  -> []
  | h::t -> (f h)::(map f t);;
map

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

composing functions

\[(f \circ g) (x) = f(g(x))\]

(* return a function that given an argument x applies f2 to x and then applies f1 to the result*)

```ocaml
let compose f1 f2 = fun x -> (f1 (f2 x));;
```

(*) another way of writing it *

```ocaml
let compose f1 f2 x = f1 (f2 x);;
```

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);;
```

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 list -> 'a
```
Exercise 1 Solution

```ocaml
val 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:

```let map f l =
  List.fold_left (fun acc x -> acc@[f x]) [] l```
Different way of thinking

“Free your mind”
-Morpheus

- Different way of thinking about computation
- Manipulate the manipulators