Announcements:

- PS2 due 11:59PM
- Quiz comments & feedback

- A good OCaml sanity check with match: RHS is all same type, as is LHS. When used as the body of a function, RHS is return type.

```
(* Avoid code reuse via map *)


let rec map (f:int->int) (lst:int list) =
 match lst
 with
   [] -> []
 | h::t -> (f h)::(map f t)

let incr z = z + 1

map incr [3;1;1;0]

map (fun z -> z + 1) [3;1;1;0]



(* Can we do this via map? *)

let rec filter (f:int->bool) (lst:int list) =
 match lst
 with
   [] -> []
 | h::t -> if (f h) then h::(filter f t) else filter f t

filter (fun z -> z > 2) [3;1;1;0]
```

```
(* Another pattern we'd like to abstract and re-use *)


(* Product of elements in a list *)

let rec listprod (lst:int list) =
 match lst
 with
  [] -> 1
 | h::t -> h*listprod(t)



(* Sum of elements in a list *)

let rec listsum (lst:int list) =
 match lst
 with
  [] -> 0
 | h::t -> h+listprod(t)


(* Generalization. List must be last argument! *)

let rec listop (base: int) (f:int->int->int) (lst:int list) =
 match lst
 with
  [] -> base
 | h::t -> f h (listop base f t)


(* Try it out *)


let listsum2 = listop 0 (+)
let mul x y = x * y
let listprod2 = listop 1 mul
```

```
(* Important pattern we'd like to abstract and re-use *)

(* Do something to each element of a list. Then turn the whole list into a
scalar somehow. *)

let rec makefuns (lst: int list) =
 match lst
 with
  [] -> []
 | h::t -> (fun z -> z+h)::makefuns(t)

let mf = makefuns [3;1;1;0]

let makefuns2 = map (fun h -> fun z -> z + h)


(* Notes:
   - can generalize the base case 0 and operator +
 *)

let rec dofunsandadd (funs: (int->int) list) (z:int) =
 match funs
 with
  [] -> 1
 | h::t -> h(z) + (dofunsandadd t z)

let ans = dofunsandadd mf 9
```

```
(* Adding new data types. Do not confuse this with the built-in data types in
OCaml! *)


(* Recursive parameterized type *)

type intlist = Nil | Cons of (int * intlist)

(* Examples *)

let list1 = Nil                    (* the empty list: [] *)
let list2 = Cons (1, Nil)          (* the list containing just 1: [1] *)
let list3 = Cons (2, Cons(1,Nil)) (* the list [2; 1] *)
let list4 = Cons (2, list2)        (* also the list [2; 1] *)

(* the list [1; 2; 3; 4; 5] *)
let list5 = Cons (1, Cons (2, Cons (3, Cons (4, Cons (5, Nil)))))

(* the list [6; 7; 8; 9; 10] *)
let list6 = Cons (6, Cons (7, Cons (8, Cons (9, Cons (10, Nil)))))



(* Manipulate intlists *)

(* Returns the length of lst *)
let rec length(lst: intlist): int =
  match lst with
    Nil -> 0
  | Cons(h,t) -> length(t) + 1

(* test to see if the list is empty *)
let is_empty(xs:intlist):bool =
    match xs with
      Nil -> true
    | Cons(_,_) -> false
```

```
(*****************************************************************)

(* Working our way up to map *)

(* Here is a way to perform a function on each element
 * of a list.  We apply the function recursively.
 *)

let inc(x:int):int = x + 1
let square(x:int):int = x * x

(* Given [i1;i2;...;in], return [i1+1;i2+1;...;in+n] *)
let rec addone_to_all(list:intlist):intlist =
    match list with
      Nil -> Nil
    | Cons(hd,tl) -> Cons(inc(hd), addone_to_all(tl))

(* Given [i1;i2;...;in], return [i1*i1;i2*i2;...;in*in] *)
let rec square_all(list:intlist):intlist =
    match list with
      Nil -> Nil
    | Cons(hd,tl) -> Cons(square(hd), square_all(tl))



(* Here is a more general method. *)

(* Given a function f and [i1;...;in], return [f(i1);...;f(in)].
 * Notice how we factored out the common parts of addone_to_all
 * and square_all. *)

let rec do_function_to_all((f:int->int), (list:intlist)):intlist =
    match list with
      Nil -> Nil
    | Cons(hd,tl) -> Cons(f(hd), do_function_to_all(f,tl))

let addone_to_all(list:intlist):intlist =
    do_function_to_all(inc, list)

let square_all(list:intlist):intlist =
    do_function_to_all(square, list)

(* Even better: use anonymous functions. *)

let addone_to_all(list:intlist):intlist =
    do_function_to_all((fun(x) -> x+1), list)

let square_all(list:intlist):intlist =
    do_function_to_all((fun(x) -> x*x), list)
```

```
(*****************************************************************)


(* Working our way up to reduce *)

(* Say we want to compute the sum and product of integers
 * in a list. *)

(* Explicit versions *)
let rec sum(list:intlist):int =
    match list with
      Nil -> 0
    | Cons(hd,tl) -> hd + sum(tl)

let rec product(list:intlist):int =
    match list with
      Nil -> 1
    | Cons(hd,tl) -> hd * product(tl)

(* Better: use a general function collapse that takes an
 * operation and an identity element for that operation.
 *)

(* Given f, b, and [i1;i2;...;in], return f(i1,f(i2,...,f(in,b))).
 * Again, we factored out the common parts of sum and product. *)
let rec collapse((f:(int * int) -> int), (b:int), (list:intlist)):int =
    match list with
      Nil -> b
    | Cons(hd,tl) -> f(hd,collapse(f,b,tl))

(* Now we can define sum and product in terms of collapse *)
let sum(list:intlist):int =
    let add((i1:int),(i2:int)):int = i1 + i2
    in
        collapse(add,0,list)

let product(list:intlist):int =
    let mul((i1:int),(i2:int)):int = i1 * i2
    in
        collapse(mul,1,list)

(* Here, we use anonymous functions instead of defining add and mul.
 * After all, what's the point of giving those functions names if all
 * we're going to do is pass them to collapse? *)
let sum(list:intlist):int =
    collapse((fun (i1,i2) -> i1+i2),0,list)

let product(list:intlist):int =
    collapse((fun (i1,i2) -> i1*i2),1,list)
```

```
(*****************************************************************)

(* Tree example with mutually recursive types *)

type inttree = Empty | Node of node
and node = { value: int; left: inttree; right: inttree }

(*

  2
 / \        Node {value=2; left=Node {value=1; left=Empty; right=Empty};
1   3                    right=Node {value=3; left=Empty; right=Empty}}

*)

(* Return true if the tree contains x. *)
let rec search ((t: inttree), (x:int)): bool =
  match t with
      Empty -> false
    | Node {value=v; left=l; right=r} ->
        v = x || search (l, x) || search (r, x)

(*****************************************************************)

(* Define our own lists *)

type 'a list_ = Nil_ | Cons_ of ('a * 'a list_)

let rec mymap (f: 'a->'b) (x: 'a list_): 'b list_ =
  match x with
      Nil_ -> Nil_
    | Cons_(h,t) -> Cons_(f(h), mymap f t)
```