Topic 1. **fold.** In addition to `map`, another common abstracted pattern of control over lists is `fold`, that combines elements of a list, given some method of combining elements, and a basis of folding `b`:

```ocaml
let rec fold s b l =  
  match l with  
  | [] -> b  
  | (x::xs) -> s(x, fold s b xs)

fold : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

So for example:

```ocaml
let sumlist = fold (fun (x,y) -> x + y) 0
sumlist [2;3;8] ⇓ 13
```

```ocaml
let prodist = fold (fun (x,y) -> x * y) 1
prodlist [2;3;8] ⇓ 48
```

A more complicated example, illustrating the powers of higher order functions:

```ocaml
(* forall : ('a -> bool) -> 'a list -> bool  
in : a predicate p on values v : 'a  
out : a function f : 'a list -> bool such  
that f(l) is true iff p(x) for all x in l  
*)
```

```ocaml
let forall p = fold (fun (x,b) -> p(x) && b) true
forall is_odd [3;4;5] ⇓ false
```

**Topic 2. Type declarations and variants.** Users may define new datatypes via *type declarations*, including *variants* type declarations. A variant type is specified by cases:

```ocaml
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
```

Note that this type definition is *polymorphic*; values at tree nodes can be any type, although any given tree must have be consistent in its node value type.

In any variant definition, the “tags” are called *constructors*, e.g. `Leaf` and `Node` are constructors, the former nullary; constructors must be capitalized. As with any type, variant types describe a set of values:

```ocaml
Node(Node(Leaf,1,Leaf), 4, Node(Node(Leaf,2,Leaf),1,Leaf)) : int tree
Node(Node(Leaf,'a',Leaf), 'b', Node(Node(Leaf,'d',Leaf),'r',Leaf)) : char tree
```

Variants are deconstructed via pattern matching, with constructors serving as pattern components, as in the
following function that performs an inorder traversal of a tree:

```ocaml
let rec inorder t = match t with
  Leaf -> []
| Node(tl,x,tr) -> (inorder tl) @ [x] @ (inorder tr)
inorder : 'a tree -> 'a list
```

```
inorder (Node(Node(Leaf,1,Leaf), 2, Leaf)) ↓ [1;2]
```

Variant types need not be recursive:

```ocaml
type 'a option = None | Some of 'a
```

**Topic 3. Extending patterns of control.** Abstracted patterns of control need not be confined to lists; for example, the concepts of mapping and folding are easily lifted to trees:

```ocaml
let rec treemap f t =
  match t with
  Leaf -> Leaf
| Node(tl,x,tr) -> Node(treemap f tl, (f x), treemap f tr)

treemap : ('a -> 'b) -> 'a tree -> 'b tree
```

```ocaml
let rec treefold s b t =
  match t with
  Leaf -> b
| Node(tl,x,tr) -> s(treefold s b tl, x, treefold s b tr)

treefold : ('a * 'b * 'a -> 'a) -> 'a -> 'b tree -> 'a
```

So for example, we can define preorder, inorder, and postorder traversals of a tree in terms of `treefold`, as follows:

```ocaml
let preorder = treefold (fun (l,x,r) -> [x] @ l @ r) []
let inorder = treefold (fun (l,x,r) -> l @ [x] @ r) []
let postorder = treefold (fun (l,x,r) -> l @ r @ [x]) []
```