This topic covers some commonly used functions for lists and sequences, including the higher-order functions filter, concat and fold.
General list functions: filter

- map is a useful and general function.
  - It captures a general situation: doing the same thing to each element of a list/seq to create a new list/seq.
  - What other general list/seq functions are there? Which is “most general”?

- One common pattern is to remove some elements from a list/seq – this is captured by filter:

  ```
  let rec filter : ('a -> bool) -> 'a list -> 'a list =
  fun isKept -> function
  | [] -> [] // 2nd arg is pattern matched
  | (x::xs) -> if isKept x then x::filter isKept xs
  else filter isKept xs
  
  > filter ((<=) 0) [1, -5, -20, 30, -35, 40];;
  val it : int list = [1; 30; 40]
  ```

- A filter function like this is in the List module of the F# libraries, and similar functions are in Seq, Set, Map, Array, ...

- List comprehensions with “if” essentially do the same thing.
  - But, sometimes filter is clearer, particularly with partial applications.
Using map and filter, we can almost do everything that list comprehensions can.
- map works like “[for ... in ... -> ...]”
- filter works like “if”.

But we can’t emulate multiple generators. Compare these:

```fsharp
[ for i in [1..5] do for j in [1..6] -> (i,j)]
val it : (int * int) list = [(1, 1); (1, 2); ...; (2,1); (2,2); ... ]

map (fun i -> map (fun j -> (i,j)) [1..6]) [1..5]
val it : (int * int) list list = [ [(1,1); (1,2); ...]; [(2,1); (2,2)...]; ... ]
```

- The first produces a list, the second a list of lists.

We can fix this with the following fun (also called flatten)

```fsharp
let rec concat : 'a list list -> 'a list = function
| [] -> []
| (xs :: xss) -> xs @ (concat xss) // Parens not needed.
```

```fsharp
concat [[1; 2]; [3]; [4; 5; 6]]
val it : int list = [1; 2; 3; 4; 5; 6]
```

Again, the F# libraries include `concat` for collection types, and comprehensions are often clearer.
fold functions

- filter, concat, map, sum, and many other list functions recursively pass through a list, adding each element to the result so far.

- We can define two functions that capture this very general pattern:

  ```
  let rec fold : ('r -> 'a -> 'r) -> 'r-> 'a list -> 'r =
  fun addElem soFar -> function
    // 3 args: addElem, soFar & pattern
    | [] -> soFar
    | x::xs -> fold addElem (addElem soFar x) xs  // tail recursion
  
  let rec foldBack : ('a -> 'r -> 'r) -> 'a list -> 'r -> 'r =
  fun addElem soFar -> function
    | [] -> soFar
    | x::xs -> addElem x (foldBack addElem xs soFar)
  
  // Examples
  let sum = fold_left (+) 0          // Start with 0, add the elems via +
  let concat xs = fold_left (@) [] xs // Start with [], add elems via @
  let map f xs = fold_right (fun y ys-> f y::ys) xs []  // Add via fun to []
  
  fold starts at the left of the list, foldBack at the right.
  Choose fold if both will work correctly - it is tail recursive.
  ◦ For seq’s there is only fold.
Fold and reduce functions

- Left vs right folds by expanding their definitions:
  \[ \text{fold} (+) \text{ start} \ [x_1; \ x_2; \ x_3; \ldots] = \]
  \[ (((\text{start} + x_1) + x_2) + x_3) + \ldots \]

  \[ \text{foldBack} (+) \ [x_1; \ x_2; \ x_3; \ldots] \text{ start} = \]
  \[ x_1 + (x_2 + (x_3 + (\ldots \ + \text{start}))) \]

- When the argument and element types are the same, and the list isn’t empty, the following fun avoids the start value:
  \[
  \text{let} \ \text{reduce} : \ ('a \rightarrow 'a \rightarrow 'a) \rightarrow 'a \ list \rightarrow 'a = \\
  \text{fun} \ \text{addElem} \ (x::xs) \rightarrow \text{fold} \ \text{addElem} \ x \ xs
  \]

  \[
  \text{If the pattern doesn’t match, there will be an exception.}
  \]

  \[
  \text{Similarly, there is reduceBack for lists, and reduce for seq’s.}
  \]
Other useful functions

- Check the Seq library docs for details of these, and more. Most have versions in List and Array that avoid conversions to Seq.
  - `singleton : 'a -> seq<'a>`  // seq with one element
  - `append : seq<'a> -> seq<'a> -> seq<'a>`  // alas, not @ or +
  - `isEmpty : seq<'a> -> bool`
  - `hd : seq<'a> -> 'a`  // first element
  - `nth : int -> seq<'a> -> 'a`  // nth element
  - `take : int -> seq<'a> -> seq<'a>`  // first n elements
  - `takeWhile : ('a -> bool) -> seq<'a> -> seq<'a>`  // up to 1st false.
  - `skip : int -> seq<'a> -> seq<'a>`  // skip n elements
  - `sort : seq<'a> -> seq<'a>`  // sort via built in <, =
  - `sortBy : ('a -> 'key) -> seq<'a> -> seq<'a>`  // order via fun
  - `min : seq<'a> -> 'a`  // least element
  - `zip : seq<'a> -> seq<'b> -> seq<'a * 'b>`  // pairs seqs.
  - `zip3 : seq<'a> -> seq<'b> -> seq<'c> -> seq<'a * 'b * 'c>`
  - `distinct : seq<'a> -> seq<'a>`  // removes duplicates
  - `initFinite : int -> (int -> 'a) -> seq<'a>`  // for i in 0...n -> f i
  - `initInfinite : (int -> 'a) -> seq<'a>`  // for i in 0... -> f
The following function takes a directory and returns a sequence of all the files it contains, including all sub-dirs.
The delay avoids reading directories until the next element is needed.

```plaintext
let rec allFiles dir =
    Seq.delay (fun () ->
        let files = Directory.GetFiles(dir)
        let subdirs = Directory.GetDirectories(dir)
        Seq.append
            files
            (subdirs |> Seq.map allFiles |> Seq.concat)
```