now down to earth

- Many recursive declarations follows the same schema.

For example:

```
let rec f = function
  | []    ->   ...
  | x::xs ->   ... f(xs) ...
```

Succinct declarations achievable using higher-order functions

Contents

- Higher-order list functions (in the library)
  - map
  - exists, forall, filter, tryFind
  - foldBack, fold

Avoid (almost) identical code fragments by
parameterizing functions with functions

A simple declaration of a list function

A typical declaration following the structure of lists:

```
let rec posList = function
    | []    -> []
    | x::xs -> (x > 0)::posList xs;;
val posList : int list -> bool list

posList [4;  -5; 6];;
val it : bool list = [true; false; true]
```

Applies the function `fun x -> x > 0` to each element in a list

Another declaration with the same structure

```
let rec addElems = function
   | []        ->  []
   | (x,y)::zs -> (x+y)::addElems zs;;
val addElems : (int * int) list -> int list

addElems [(1,2) ;(3,4)];;
val it : int list = [3; 7]
```

Applies the addition function + to each pair of integers in a list

The function: `map`

Applies a function to each element in a list

$$\texttt{map}\ f\ [v_1; v_2; \ldots; v_n] \ = \ [f(v_1); f(v_2); \ldots; f(v_n)]$$

Declaration

Library function

```
let rec map f = function
   | []    -> []
   | x::xs -> f x :: map f xs;;
val map : ('a -> 'b) -> 'a list -> 'b list
```

Succinct declarations can be achieved using map, e.g.

```
let posList = map (fun x -> x > 0);;
val posList : int list -> bool list

let addElems = map (fun (x,y) -> x+y);;
val addElems : (int * int) list -> int list
```

Higher-order list functions: `filter`

Set comprehension: $\{x \in xs : p(x)\}$

`filter` p xs is the list of those elements x of xs where $p(x) = \texttt{true}$.

Declaration

Library function

```
let rec filter p = function
  | []     -> []
  | x::xs -> if p x then x :: filter p xs
                else filter p xs;;
val filter : ('a -> bool) -> 'a list -> 'a list
```

Example

```
filter System.Char.IsLetter ['1'; 'p'; 'F'; '-'];;
val it : char list = ['p'; 'F']
```

where `System.Char.IsLetter` c is true iff
$c \in \{'\mathrm{A}', \ldots, '\mathrm{Z}'\} \cup \{'\mathrm{a}', \ldots, '\mathrm{z}'\}$

Exercise

Declare a function

$$\texttt{inter}\ xs\ ys$$

which contains the common elements of the lists xs and ys — i.e. their intersection.

Remember:
$\texttt{filter}\ p\ xs$ is the list of those elements x of xs where $p(x) = \texttt{true}$.

Higher-order list functions: `exists`

Predicate: For some x in xs : $p(x)$.

$$\texttt{exists}\ p\ xs = \begin{cases} \text{true} & \text{if } p(x) = \texttt{true} \text{ for some } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Declaration

Library function

```
let rec exists p = function
  | []    -> false
  | x::xs -> p x || exists p xs;;
val exists : ('a -> bool) -> 'a list -> bool
```

Example

```
exists (fun x -> x>=2) [1; 3; 1; 4];;
val it : bool = true
```

Higher-order list functions: `forall`

Predicate: For every x in xs : $p(x)$.

$$\texttt{forall}\; p\; xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true}, \text{ for all elements } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Declaration

Library function

```
let rec forall p = function
  | []    -> true
  | x::xs -> p x && forall p xs;;
val forall : ('a -> bool) -> 'a list -> bool
```

Example

```
forall (fun x -> x>=2) [1; 3; 1; 4];;
val it : bool = false
```

Exercises

Declare a function

$$\texttt{disjoint}\; xs\; ys$$

which is true when there are no common elements in the lists *xs* and *ys*, and false otherwise.

Declare a function

$$\texttt{subset}\; xs\; ys$$

which is true when every element in the lists *xs* is in *ys*, and false otherwise.

Remember

$$\texttt{forall}\; p\; xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true, for all elements } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Higher-order list functions: `fold` (1)

Suppose that $\oplus$ is an infix function.

Then the `fold` function has the definitions:

$$\texttt{fold}\ (\oplus)\ e_a\ [b_0;\ b_1;\ \ldots\ ;\ b_{n-2};\ b_{n-1}\ ]$$
$$=\ ((\ldots((e_a \oplus b_0) \oplus b_1)\ldots) \oplus b_{n-2}) \oplus b_{n-1}$$

i.e. it applies $\oplus$ from left to right.

Examples:

$$\texttt{List.fold (-) 0 [1; 2; 3]} = ((0-1)-2)-3 = -6$$
$$\texttt{List.foldBack (-) [1; 2; 3] 0} = 1-(2-(3-0)) = 2$$

Higher-order list functions: fold (2)

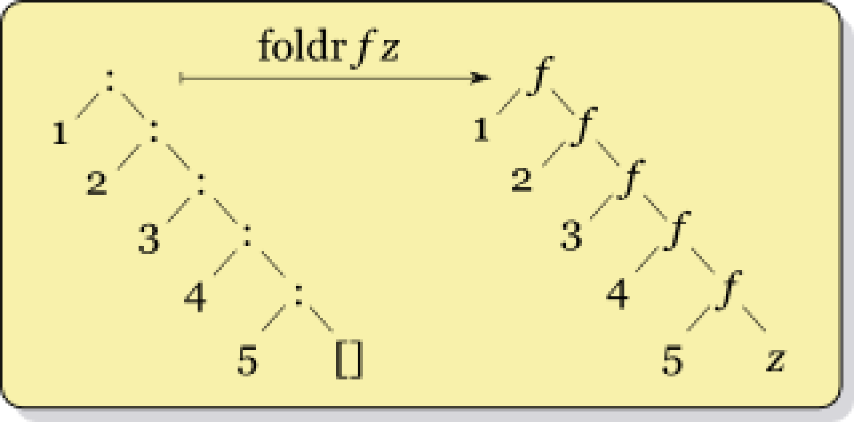
```
let rec fold f e = function
  | x::xs -> fold f (f e x) xs
  | []    -> e  ;;
val fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

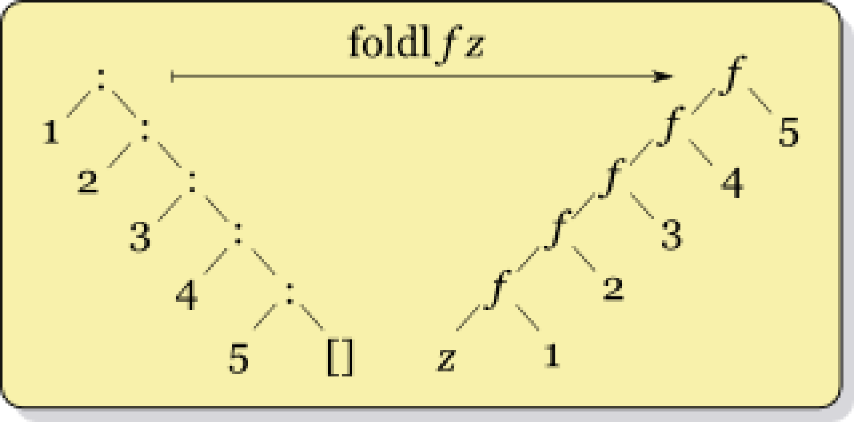
Using cons in connection with fold gives the reverse function:

```
let rev xs  = fold (fun rs x -> x::rs) [] xs;;
```

This function has a linear execution time:

```
   rev [1;2;3]
⇝  fold  (fun ...)  []          [1;2;3]
⇝  fold  (fun ...)  (1::[])     [2;3]
⇝  fold  (fun ...)  [1]         [2;3]
⇝  fold  (fun ...)  (2::[1])    [3]
⇝  fold  (fun ...)  [2;1]       [3]
⇝  fold  (fun ...)  (3::[2;1])  []
⇝  fold  (fun ...)  [3;2;1]     []
⇝  [3;2;1]
```


foldr f z
:
1
:
2
:
3
:
4
:
5
[]
f
1
f
2
f
3
f
4
f
5
z


foldl f z
:
1
:
2
:
3
:
4
:
5
[]
f
f
5
f
4
f
3
f
2
z
1