Functions on Lists

We have already seen the List data type construction, and the two programs (operations) of append and reverse.

```
open List;;

(@);;
rev;;
```

We now look at a few important functions on lists.

Filter

Informal specification: Given a *predicate* `p: 'a -> bool`, and a list `s: 'a list`, return a list consisting of those elements *x* of `s` (in the same order as in `s`) for which `(p x)` is true. A (unary) predicate on the type `'a` is a function of type `'a -> bool`.

```
let rec filter p s = match s with
  [ ] -> [ ]
| x :: xs -> if (p x) then x :: (filter p xs)
                      else (filter p xs)
;;
```

The type of filter is `filter: ('a -> bool) -> ('a list -> 'a list)`

A slightly more elegant and readable, though *not* more efficient, implementation avoids repeating the subexpression `(filter p xs)`

```
let rec filter p s = match s with
  [ ] -> [ ]
| x :: xs -> let ys = filter p xs
             in
             if( p x) then x::ys else ys
;;
```

This version uses the **`let`** $x = e_1$ **`in`** e_2 construct of OCaml to give a name x to an expression e_1 (an act of *definition*), and use this defined name x in the expression e_2.

Testing `filter`:

```
filter (fun x -> ((x mod 2) = 0)) [ ];;
filter (fun x -> ((x mod 2) = 0)) [ 1; 2; 3; 4; 5; 6; 7];;
```

Here we have not give a name to the predicate which checks if an integer is even (or not), and instead have used an anonymous function form `(fun x -> ((x mod 2) = 0))`.
We can omit some parentheses and write it as: `(fun x -> x mod 2 = 0)`
This is the (unnamed) function on argument `x` that returns `true` if `x` is even and `false` otherwise.

Question: What is its time and space complexity of `filter`?

Exercise: Show that for any predicate `p: 'a -> bool`, and any list `s: 'a list`,

```
length (filter p s) ≤ length s
```

Exercise: Show that for the predicate `(fun x -> true)`, and any list `s: 'a list`,

```
filter (fun x -> true) s) = s
```

and for the predicate `(fun x -> false)`, and any list `s: 'a list`,

```
filter (fun x -> false) s) = [ ]
```

Exercise: Show that for any predicate `p: 'a -> bool`, and any lists `s1: 'a list` and `s2: 'a list`,

```
filter p (s1 @ s2) = (filter p s1) @ (filter p s2)
```

Exercise: Show that for any predicate `p: 'a -> bool`, and any list `s: 'a list`,

```
rev (filter p s) = filter p (rev s)
```

Exercise: (Idempotence of `filter p`) Show that for any predicate `p: 'a -> bool`, and any list `s: 'a list`,

```
filter p (filter p s) = filter p s
```

Exercise: (Commutation of predicates in `filter`) Show that for any predicates `p: 'a -> bool`, and `q: 'a -> bool`, and any list `s: 'a list`,

```
filter p (filter q s) = filter q (filter p s)
```

Map

Informal specification: Given a *function* `f: 'a -> 'b`, and a list `s: 'a list`, return a list consisting of elements `(f x)` (in the same order as the elements *x* appear in `s`), i.e., where the function is applied to each element of `s`.

The type of this function map is given by:

```
map:( 'a -> 'b) -> (('a list) -> ('b list) )

let rec map f s = match s with
    [] -> []
  | x :: xs -> (f x) :: (map f xs)
;;
```

What is the time and space complexity of `map`?

Testing `map` with a function that squares the input:

```
map (fun x -> x*x) [ ] ;;
map (fun x -> x*x) [1; 2; 3; 4; 5; 6 ] ;;
```

Exercise: Show that for any function `f: 'a -> 'b`, and any list `s: 'a list`,

```
length (map f s) = length s
```

Function composition as a higher-order function.

Let us now make a small digression into functions and their composition.

First we define a very useful (and useless!) function:

```
let id x = x;;
```

Testing `id` with different arguments

```
id 5;;
id true;;
id (fun x -> x*x);;
id id;;
```

Now let us define another *useful* function, that given a function `f: 'a -> 'b`, and another function `g: 'b -> 'c`, returns their function composition, which is a *function* of type `'a -> 'c`.

```
let comp f g x = g(f x);;
```

The type of `comp` is: `('a -> 'b ) -> (('b -> 'c) -> ('a -> 'c))`

Exercise: (Identities of comp) Show that for any function `f: 'a -> 'b`,

```
comp id f = f
```

and

```
comp f id = f
```

Exercise: (Associativity of comp) Show that for any functions `f: 'a -> 'b`, `g: 'b -> 'c`, and `h: 'c -> 'd`,

```
comp f (comp g h) = comp (comp f g) h
```

Returning to our discussion on `map`, we can show the following facts.

Exercise: Show that for any list `s: 'a list`,

```
map id s = s
```

Exercise: Show that for any function `f: 'a -> 'b`, and any lists `s1: 'a list` and `s2: 'a list`,

```
map f (s1 @ s2) = (map f s1) @ (map f s2)
```

Exercise: Show that for any function `f: 'a -> 'b`, and any list `s: 'a list`,

```
rev (map f s) = map f (rev s)
```

Exercise: Show that for any functions `f: 'a -> 'b, g: 'b -> 'c,` and any list `s: 'a list`,

```
map (comp f g) s = map g (map f s)
```

Since equality of two functions means that both return the same value on all inputs, this last equation can be presented as the following (by abstracting on the argument list):

For any functions `f: 'a -> 'b, g: 'b -> ‵c,`

```
map (comp f g) = comp (map f) (map g)
```

Note that while `map` has been defined here as a recursive function, which sequentially traverses its input list, it lends itself to perfect parallelisation: `(f x)` can be evaluated for each list element `x` *in parallel*. Thus this `map` is indeed the eponymous operation of the "*map-reduce*" paradigm.

Summing a list of integers

Given list of integers, add them. What is the time and space complexity? Write a recurrence relation.

```
let rec sum s = match s with
  [ ] -> 0
| x::xs-> x+(sum xs) ;;
```

The type of this program is: `sum: int list -> int`
Testing `sum:`

```
sum [ ];;
sum [ 1; 2; 3; 4; 5];;
```

Exercise: Show that for any lists `s1: int list` and `s2: int list`,

```
sum (s1 @ s2) = (sum s1) + (sum s2)
```

Question: Does this fact suggest a different recurrence relation for adding the elements of a list?

Exercise: Show that for any list `s: int list`,

```
sum (rev s) = sum s
```

Multiplying a list of integers

Given list of integers, multiply them. What is the time and space complexity? Write a recurrence relation.

```
let rec prod s = match s with
  [ ] -> 1
| x::xs-> x * (prod xs) ;;
```

The type of this program is: `prod: int list -> int`
Testing `prod:`

```
prod [ ];;
prod [ 1; 2; 3; 4; 5];;
```

Exercise: Show that for any lists `s1: int list` and `s2: int list`,

```
prod (s1 @ s2) = (prod s1) * (prod s2)
```

Question: Does this fact suggest a different recurrence relation for multiplying the elements of a list?

Exercise: Show that for any list `s: int list`,

```
prod (rev s) = prod s
```

Finding the least element in a list of integers

Given list of integers, find its least element. What is the time and space complexity? Write a recurrence relation.

```
let rec listmin s = match s with
  [ ] -> max_int
| x::xs-> min x (listmin xs) ;;
```

The type of this program is: `listmin: int list -> int`
Testing `listmin`:

```
listmin [ ];;
listmin [ 1; 2; 3; 4; 5];;
```

Does this program look similar to the earlier ones? Can you think of some facts to prove about `listmin`?

Finding the conjunction of a list of booleans

Given list of booleans, find their *conjunction*. What is the time and space complexity? Write a recurrence relation.

```
let rec forall s = match s with
  [ ] -> true
| x::xs ->  x && (forall xs) ;;
```

The type of this program is: `forall: bool list -> bool`
Testing `forall`:

```
forall [ ];;
forall [ true; true; true];;
forall [ true; false; true];;
```

Does this program look similar to the earlier ones? Can you think of some facts to prove about `forall`?

Exercise: Write a program to find the *disjunction* of a list of booleans. What facts can you prove about your program?

A generalised "FOLD" operation

It is possible to generalise from these different programs: in all of them, we have a binary associative operator (addition, multiplication, minimum, conjunction). Note that these examples also happen to be commutative, but that may be a distraction. To dispel that misconception, we can try to write a program that given a list of $n \times n$ matrices (over a field), multiplies them to return an $n \times n$ matrix.

We further note that in the base case (i.e., an empty list), we return the *identity* element of the binary operation. And in the inductive case, we perform the binary operation on the first element with the result of a recursive call on the remaining elements.

Thus the programs above provide a generalisation of an associative binary operator to a corresponding n-ary operator. Let us codify this in a program which we shall call `foldr`.

`foldr` takes a function `f: 'a -> 'a -> 'a,` its identity element `e: 'a,` and a list `s: 'a list,` and returns the result of applying `f` repeatedly, "folding" in one element of the list at a time using the binary operation `f.` In the base case (empty list) the identity element of `f,` namely `e,` is returned. The intended type is:

```
foldr: ('a -> 'a -> 'a ) -> 'a ->('a list) -> 'a
```

```
let rec foldr f e s = match s with
      [ ] -> e
    | x::xs-> f x (foldr f e xs) ;;
```

Question: What is the time and space complexity of `foldr`?

We claim we can code all the earlier programs using this generalised higher-order function. Test for example the following:

```
foldr (fun x-> fun y -> x+y) 0 [1;2;3;4;5];;
foldr (fun x -> fun y -> x*y) 1 [ 1; 2; 3; 4; 5];;
foldr min max_int [ 1; 2; 3; 4; 5];;
foldr (fun x -> fun y -> x && y) true [true; false; true];;
```

Exercise: Prove the following equalities of functions, using structural induction on an argument list.

```
sum = foldr (fun x -> fun y -> x+y) 0
prod = foldr (fun x -> fun y -> x*y) 1
listmin = foldr min max_int
forall = foldr (fun x -> fun y -> x && y) true
```

But wait! Did you notice the type of `foldr?` It is *not* the intended type. Instead we have got the following type

```
foldr: ('a -> 'b -> 'b ) -> 'b ->('a list) -> 'b
```

This is because the type inference method (quite correctly) did *not* make the assumption that the result type must be the same type as that of the elements of the list.

A more efficient tail-recursive fold

The "problem" with `foldr` is that it makes recursive calls until it reaches an empty list, and starts computing the result by applying the function `f` on *returning* from the recursive call. Thus it is "folding" the answer from the *right*:

$$f(x_1, f(x_2, \ldots f(x_n, e)\ldots))$$

A more efficient approach would be to perform the computation on the way *forward*, as we traverse the list. The trick is to carry an *accumulated* partial result, and apply the operation `f` to the accumulated result and the next element in the list, to generate a new accumulated result and so on. Thus we are computing the "reduce" operation by folding from the left. (It is correct to do so if the operation `f` is associative, and the left and right identity elements coincide.)

```
let rec foldl f e s = match s with
      [ ] -> e
    | x::xs-> foldl f (f e x) xs ;;
```

Note that the recursive call to `foldl` is a tail recursive call — it is not embedded under any other operation, unlike in `foldr`, where the recursive call is under `f x (___)`. And instead of retaining the identity element `e` in the (tail) recursive call (as was the case in `foldr`), we have replaced it by the new accumulated result `(f e x)`.

Let us test the program `foldl`:

```
foldl (fun x -> fun y -> x+y) 0 [1;2;3;4;5];;
foldl (fun x -> fun y -> x*y) 1 [ 1; 2; 3; 4; 5];;
foldl min max_int [ 1; 2; 3; 4; 5];;
foldl (fun x -> fun y -> x && y) true [true; false; true];;
```

But again, we have a slight surprise when seeing what type the OCaml interpreter inferred for `foldl`

```
foldl: ('a -> 'b -> 'a ) -> 'a ->('b list) -> 'a
```

If we swap the type variables `'a` and `'b` around, we would get

```
foldl: ('b -> 'a -> 'b ) -> 'b ->('a list) -> 'b
```

which is not identical to, but not too different from

```
foldr: ('a -> 'b -> 'b ) -> 'b ->('a list) -> 'b
```

Note that the OCaml `List` module contains functions called

```
fold_left;;
fold_right;;
```

Look at their types, which indicate a reordering of the arguments compared with the definitions we gave above.

Applications using lists

Linear search for an element `x` in a list `s`.

```
member:  'a -> 'a list -> bool

let rec member x s = match s with
        [ ] -> false
      | y :: ys -> if x=y then true
                           else member x ys ;;
```

What is the time complexity (worst-case) of `member`?

Insertion sort. Here we assume that we have a list of integers. This can be generalised to any type for which there is a total ordering relation (i.e., a partially ordered set that satisfies the dichotomy condition for any two elements).

We first define a function `insert` that inserts an element `elt` into its appropriate position in a given sorted list `srtdlst`:

```
let rec insert elt srtdlst = match srtdlst with
        [ ] -> [elt]   (* inserting into an empty list *)
      | head :: tail -> if elt <= head
```

```
                        (* insert at front, returns a sorted list *)
                                    then elt :: srtdlst
                        (* recursively: insert at appropriate place
                           in the tail, returns a sorted list *)
                                    else head :: (insert elt tail) ;;
```

The important mathematical fact about `insert` is that if it is given a sorted list (in non-decreasing order) as the second argument, it returns a sorted list. Without this assumption on the input list, we will not be able to guarantee anything about sortedness. What is the time complexity (worst-case) of `insert`?

We now develop an insertion sort program which uses `insert`. If you want to hide the program `insert` from being visible and available, define it locally using the **`let rec`** ___ **`in`** ___ construct introduced above in the sorting program `in_sort`:

```
let rec in_sort s = match s with
      [ ] -> [ ]   (* an empty list is sorted *)
     | h :: t -> insert h (in_sort t)
          (* recursively sort the tail,
             and insert the head at the appropriate place *)
;;
```

The correctness of this program arises from a mathematical invariant which is maintained: that it always returns a *sorted list*, assuming the correctness of `insert`.
Testing `in_sort`:

```
in_sort [ ];;
in_sort [ 65 ];;
in_sort [ 4; 3; -2; 5; 7; 4; 8; 2; 43; 2; 32; 1; 99];;
```