```
(* liste e programmazione ricorsiva e programmazione a combinatori)
# let empty = [] in empty;;
- : 'a list = []
# let u = 3 in [u];;
- : int list = [3]
# let u = 3 and w = [4;5] in u::w;;
- : int list = [3; 4; 5]


(* operatori intervallo e programmazione ricorsiva *)
# let rec nToK = fun n -> fun k -> if (k>0) then n::nToK(n+1)(k-1) else [];;
val nToK : int -> int -> int list = <fun>
# nToK 1 7;;
- : int list = [1; 2; 3; 4; 5; 6; 7]
# let rec nTom = fun n -> fun m -> if (n>m) then [] else n::nTom(n+1)m;;
val nTom : int -> int -> int list = <fun>
# nTom 3 12;;
- : int list = [3; 4; 5; 6; 7; 8; 9; 10; 11; 12]
# let rec cnTom = fun (n,m) -> if (n>m) then [] else n::cnTom((n+1),m);;
val cnTom : int * int -> int list = <fun>
# cnTom(3,15);;
- : int list = [3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15]


(* matching e programmazione ricorsiva *)
# exception Null;;
exception Null
# let hd = fun l -> match l with [] -> raise Null | x::xs -> x;;
val hd : 'a list -> 'a = <fun>
# hd[3;5;8];;
- : int = 3
# hd[];;
Exception: Null.
# let rec fact n = if n=0 then 1 else n * fact(n-1);;
val fact : int -> int = <fun>
# fact 3;;
- : int = 6
# exception IllegalArgument of string;;
exception IllegalArgument of string
# IllegalArgument("luigi");;
- : exn = IllegalArgument "luigi"
# let rec fact n = match n with
     0 -> 1
    |k when k>0 -> n * fact(n-1)
    |otherwise -> raise (IllegalArgument "fact");;
val fact : int -> int = <fun>
# fact (-5);;
Exception: IllegalArgument "fact".
# let rec append ul vl = match ul with [] -> vl | u::uls -> u::(append uls vl);;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [2] [3];;
- : int list = [2; 3]


(* programmazione higher order *)
# let rec map = fun g -> fun l -> match l with [] -> [] | x::xs -> (g x)::(map g xs);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map (fun x -> x+1) [10;20;30];;
- : int list = [11; 21; 31]
# let listSumk = fun l -> fun k -> map (fun x -> x+k) l;;
val listSumk : int list -> int -> int list = <fun>
```

```
# listSumk [3;5;7;9] 12;;
- : int list = [15; 17; 19; 21]
# let rec foldL g n l = match l with [] -> n | x::xs -> foldL g (g n x) xs;;
val foldL : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# foldL (-) 100 [3;5;7];;
- : int = 85
# let rec foldR g n l = match l with [] -> n | x::xs -> g x (foldR g n xs);;
val foldR : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
# foldR (-) 100 [3;5;7];;
- : int = -95


(* programmazione tail recursive *)
#let fact n =
    let rec factT n r = match n with
             0 -> r
            |k when k>0 -> factT(n-1)(n*r)
            |otherwise -> raise (IllegalArgument "fact")
    in factT n 1;;
    val fact : int -> int = <fun>
    # fact 3;;
- : int = 6
#let append ul vl =
    let rec appendT ul vl r = match ul with
                                   [] -> r(vl)
                                  |(u::uls) -> appendT uls vl (fun ls -> r(u::ls))
    in appendT ul vl (fun x -> x);;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [2;3;4][5;6];;
- : int list = [2; 3; 4; 5; 6]
# let foldR g n l =
  let rec foldRT g n l r = match l with
        [] -> r(n)
       |x::xs -> foldRT g n xs (fun r1 -> r(g x r1))
  in foldRT g n l (fun x -> x);;
# foldR (-) 100 [3;5;7];;
- : int = -95


(* programmazione a combinatori)
# let summation n = foldL (+) 1 (nTom 1 n);;
val summation : int -> int = <fun>
# summation 5 ;;
- : int = 15
# let prod x y = (x*y);;
val prod : int -> int -> int = <fun>
# let fact n = foldL (prod) 1 (nTom 1 n);;
val fact : int -> int = <fun>
# fact 3;;
- : int = 6
# let size = foldL (fun x -> (+) 1) 0;;
val size : int list -> int = <fun>
# size [1;2;3;5];;
- : int = 6
# let size = foldR (fun x -> (+) 1) 0;;
val size : '_a list -> int = <fun>
# size [1;2;3;5];;
- : int = 4
# let isIn x ls = foldR (||) false (map (fun u -> u=x) ls);;
```