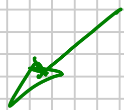
FUNZIONI RICORSIVE su LISTE

Ricerca del valore massimo in una lista

```
let rec max l = match l with
      [x] -> x
    | x :: y :: xs -> if x > max (y :: xs) then x else max (y :: xs);;
```



```
let rec max l = match l with
      [x] -> x
    | x :: y :: xs -> let m = max (y :: xs) in
                      if x > m then x else m ;;
```

max : int list -> int = <fun>

max [4; 3]

= { 2° pattern, x = 4, y = 3, xs = [] }

let m = max (3 :: []) in
  if 4 > m then 4 else m

= { def. di max, 1° pattern, x = 3 }

let m = 3 in
  if 4 > 3 then 4 else 3

= 4

Se avete una espressione E in cui viene calcolata più volte una sottoespressione E1, potete riscrivere E con

let v = E1 in

$E\left({}^{v}/_{E1} \right)$ ← E in cui v rimpiazza E1

Esempio

E: if $x > f(y)$ then x else $f(y)$

let $v = f(y)$ in if $x > v$ then x else v ⟵ $E\left(v/f(y)\right)$

E: if $g(z) > f(x,y)$ then $g(z) + 1$ else $f(x,y) + g(z)$

let $v1 = g(z)$ in
  let $v2 = f(x,y)$ in
    $E\left(v2/f(x,y),\ v1/g(z)\right)$ if $v1 > v2$ then $v1 + 1$ else $v1 + v2$

let $v1 = g(z)$ and $v2 = f(x,y)$
in if $v1 > v2$ then $v1+1$ else $v1+v2$;;

```
let rec max l = match l with
    [x] -> x
  | x::y::xs -> let m = max (y::xs) in
                if x>m then x else m;;
```

```
let rec max l =
  if tl l=[] then hd l
  else let m = max (tl l)
    in
    if (hd l) > m
      then (hd l)
      else m ;;
```

selettori
funzioni che
selezionano parti
di un valore
strutturato

Scrivere una funzione che, data una lista, calcola la coppia (m_1, m_2) dove m_1 è il valore massimo nella lista e m_2 è il valore minimo nella lista

①
```
let minmax l =

    let rec max l = match l with [x] -> x | x::y::xs -> let m = max (y::xs) in
                                                        if x>m then x else m
    in
    let rec min l = match l with [x] -> x | x::y::xs -> let m = min (y::xs) in
                                                        if x<m then x else m
    in
      (max l, min l) ;;
```

← qui posso "usare" max e min

minmax : int list → int * int = <fun>

(a, b) è una coppia

```
let z =
    let x = 10 in let y = 11
    in
        x + y ;;
```

```
let rec minmax l = match l with
  [x] -> (x, x)
| x::y::ys -> let (m1, m2) = minmax (y::ys)
  in
     if x < m2 then (m1, x)
     else if x > m1 then (x, m2) else (m1, m2);;
```

pattern matching:
m1 "sta per" il 1° elemento della coppia
m2 "sta per" il 2° elemento della coppia

minmax : int list -> int * int = <fun>

Senza usare il let (m1, m2) = ... locale:

```
| x::y::ys -> if x < snd (minmax (y::ys))
  then (fst (minmax (y::ys)), x)
```

```
let snd (a,b) = b
let fst (a,b) = a
```

INVERTIRE una LISTA

reverse : 'a list → 'a list   che inverte l'ordine degli elementi di una lista

reverse [1;2;3] = [3;2;1]

reverse $[x_1; \dots; x_n]$ =

(reverse $[x_2; \dots; x_n]$) @ $[x_1]$

```
let rec reverse l = match l with
    []     ->  []
  | x::xs  ->  (reverse xs) @ [x] ;;
```

~~(reverse xs) @ x~~   no! errore di tipo   @ : 'a list → 'a list → 'a list

~~(reverse xs) :: x~~   no! errore di tipo   :: : 'a * 'a list → 'a list

reverse [1;2;3]

= {2° pattern, x = 1, xs = [2;3]}

(reverse [2;3]) @ [1]

= {2° pattern, x = 2, xs = [3]}

(reverse [3] @ [2]) @ [1]

= {2° pattern, x = 3, xs = []}

• (((reverse []) @ [3]) @ [2]) @ [1]

= {1° pattern}

(([] @ [3]) @ [2]) @ [1]

= ([3] @ [2]) @ [1]

= [3; 2; 1]

Reverse con ACCUMULATORE

```
let rec reva l a = match l with
    [] -> a
  | x::xs -> reva xs (x::a) ;;
```

reva : 'a list → 'a list → 'a list

('a list = l, 'a list = a, 'a list = ris.)

reva [1] [2;3]

= {2° patt., x = 1, xs = []}

reva [] [1;2;3]

= {1° patt.}

[1;2;3]

Cosa succede se chiamiamo reva con la 2ª lista vuota?

reva [1;2;3] [ ]
= {2° patt.}
reva [2;3] [1]
= {2° pattern}
reva [3] [2;1]
= {2° patt.}
reva [ ] [3;2;1]
= {1° pattern}
[3;2;1]

→

```
let reverse l =
    let rec reva l a = . . . .
    in
      reva l [ ] ;;
```

reverse : 'a list → 'a list = <fun>

l è usato come 1° argomento dell'applicazione di reva

reva : 'a list → 'a list → 'a list

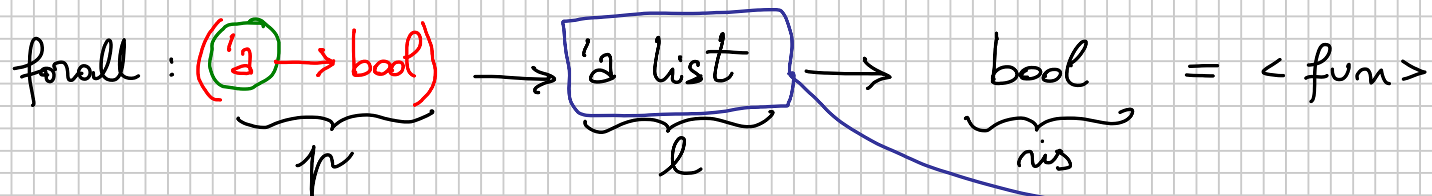
FUNZIONI di ORDINE SUPERIORE (al primo)

Sono funzioni che hanno come argomento (tra gli altri) altre funzioni, o che restituiscono come risultato funzioni.

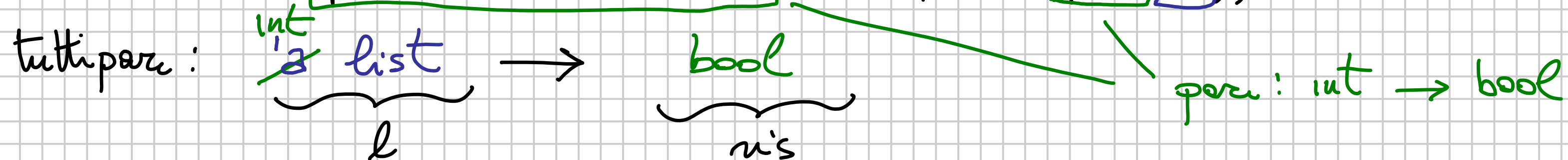
FORALL : una funzione che restituisce <u>true</u> se tutti gli elementi di una lista soddisfano una proprietà, <u>false</u> altrimenti

```
let rec tuttipari l = match l with        (tuttidispari)
    [] -> true
  | x::xs when x mod 2 <> 0 -> false      (=)
  | x::xs when x mod 2 = 0 -> tuttipari xs;;   (<>) (tuttidispari)

tuttipari : int list -> bool = <fun>

tuttidispari : int list -> bool = <fun>
```

```
let rec forall p l = match l with
    []                      -> true
  | x::xs when not (p x)    -> false
  | x::xs when p x          -> forall p xs ;;
```

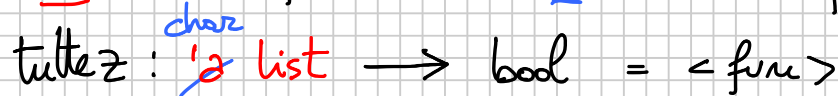



```
# let tuttipari l = let pari x = x mod 2 = 0 in forall pari l ;;
```




```
# let tuttez l = let foo x = x='z' or x='Z' in forall foo l ;;
```

foo : char -> bool



Esercizio : definire

exists : ('a → bool) → 'a list → bool   che realizza ∃

o ———————— o

Applicazione di uno stesso "operatore" (di una stessa funzione) a tutti gli elementi di una lista. Es.

```
let rec incr l = match l with
    [] -> []
  | x::xs -> (x+1) :: (incr xs);;

incr : int list -> int list = <fun>

# incr [10;20;-1] =
      [11;21;0]
```

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

# let incr l = let f x = x+1 in map f l;;
# let double l = let f x = x+x in map f l;;
# let incr10 l = let f x = x+10 in map f l;;
```

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

map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

tipo f — tipo l — tipo ris.

```
# let double l = let f x = x+x in map f l;;
    double : int list -> int list = <fun>
```

f: int -> int

```
# let foo l = let f x = if x>0 then 'a' else 'b' in map f l;;
    foo : int list -> char list = <fun>
```

f: int -> char

'a 'b