

FUNZIONI RICORSIVE SU LISTE

- ESEMPIO (SOMMA DEGLI ELEMENTI)

```
# let rec sum l =
  match l with
  | [] -> 0
  | x::xs -> x + sum xs ;;
```

$\text{sum} : \underbrace{\text{int list}}_{\text{Tipo } l} \rightarrow \underbrace{\text{int}}_{\text{Tipo } \text{ris}} = \langle \text{fun} \rangle$

- ESEMPIO (VALORE MASSIMO ALL'INTERNO DI UNA LISTA)

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

|| definita su liste non vuote

chiama due volte max

altra soluzione (con dichiarazione locale)

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

//> let... in
 mi consente di
 fare operazioni
 in sequenza

- ESEMPIO (scrivere una funzione che calcola la coppia $(m1, m2)$ dove $m1$ è il massimo di una lista e $m2$ è il minimo)

① # let minmax e =

let rec max e =

match e with

$x::[] \rightarrow x$

| $x::xs \rightarrow$ let $m = \text{max } xs$ in

if $x > m$ then x else m

in

let rec min e =

match e with

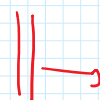
$x::[] \rightarrow x$

| $x::xs \rightarrow$ let $m = \text{min } xs$ in

if $x < m$ then x else m

in

$(\text{max } e, \text{min } e)$;;



FUNZIONI
AUSILIARIE
(LOCALI)



② # let rec minmax e =

match e with

$x :: [] \rightarrow (x, x)$

| $x :: xs \rightarrow$ let $w = \text{minmax } xs$ in

if $x > \text{fst } w$ then $(x, \text{snd } w)$

else if $x < \text{snd } w$ then $(\text{fst } w, x)$

else w ;;

$w = (\dot{6}, 2)$ $x = \dot{8}$
 $(8, 2)$

modo ALTERNATIVO di SCRIVERE IL 2° CASO DEL PATTERN MATCHING

| $x :: xs \rightarrow$ let $(m1, m2) = \text{minmax } xs$ in

if $x > m1$ then $(x, m2)$

else if $x < m2$ then $(m1, x)$

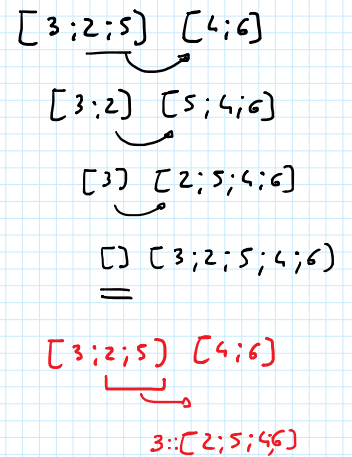
else $(m1, m2)$;;

- ESEMPIO (append @)

$$[3;2;5] @ [4;6] = [3;2;5;4;6]$$

scriviamo una funzione ricorsiva che corrisponde a @ (concatena liste)

```
#let rec append l1 l2 =
  match l1 with
  [] -> l2
  | x::xs -> x :: (append xs l2) ;;
```



Ricordate la reverse? Potevo scriverla così:

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

```
#let rec reverse_append l =
  match l with
  [] -> []
  | x::xs -> (reverse_append xs) @ [x];;
reverse_append : 'a list -> 'a list = <fun>
```

```
#let reverse l =
  let rec reverse_accum l1 l2 =
    match l1 with
    [] -> l2
    | x::xs -> reverse_accum xs (x::l2)
  in
  reverse_accum l [];;
reverse : 'a list -> 'a list = <fun>
```

// SOLUZIONE CORRETTA
MA INEFFICIENTE

```
#let rec crealista n =
  if n=0 then [] else n::crealista (n-1);;
crealista : int -> int list = <fun>
```

```
# reverse (crealista 10000);;
```

```
# reverse_append (crealista 10000);;
```

- ESEMPIO (Take - calcola la lista ottenuta prendendo i primi n elementi della lista passata come argomento)

$$\text{Take } 2 \ [6;4;5;2] = [6;4]$$

$$\text{Take } 0 \ [] = []$$

$$\text{Take } n \ [] = []$$

let rec Take n l =

match (n, l) with

| $(0, _)$ → $[]$

| $(_, [])$ → $[]$

| $(n, x::xs) \rightarrow x::(\text{Take } (n-1) \ xs) ;;$

posso usare $_$ al posto delle variabili nel pattern quando non sono interessato al valore che fa match con quella parte di pattern

- fa match con qualunque cosa

$$\text{Take } -2 \ [7;5;4]$$

↓

$$7::(\text{Take } -3 \ [5;4])$$

↓

$$7::(5::\text{Take } -4 \ [4])$$

↓

$$7::(5::(4::\text{Take } -5 \ []))$$

↓

$$7::(5::(4::[])) = [7;5;4]$$

supponiamo di volere la lista vuota come risultato se $n < 0$

let rec Take n l =

match (n, l) with

| $(n, _)$ when $n <= 0$ → $[]$

| $(-, [])$ → $[]$

| $(n, x::xs) \rightarrow x::\text{Take } (n-1) \ xs ;;$

consente di definire condizioni sui valori ottenuti dal pattern matching

X ESERCIZIO :

- definire una funzione `ord : int list → bool` che restituisce `True` se la lista passata come argomento è ordinata in modo crescente
- definire una funzione `drop` che restituisce tutti gli elementi di una lista esclusi i primi `n`
- definire una funzione `nth` che restituisce l'elemento della lista in posizione `n`

FUNZIONI DI ORDINE SUPERIORE (AL PRIMO)

Sono funzioni che prevedono un argomento o danno un risultato che sono altre funzioni

→ abbiamo già visto

```
# let apply f x = f x
apply : ('a → 'b) → 'a → 'b = <fun>
```

$\underbrace{\hspace{1.5cm}}_{\text{Tipo } f} \quad \underbrace{\hspace{1.5cm}}_{\text{Tipo } x} \quad \underbrace{\hspace{1.5cm}}_{\text{Tipo del risultato}}$

Vediamo cosa ci consentono di fare

→ scriviamo una funzione che verifica se gli elementi di una lista sono tutti pari

```
# let rec TuttiPari l =
  match l with
  | [] → True
  | x::xs → if (x mod 2 != 0) then false
            else TuttiPari xs;;
```

→ e Tutti dispari ?

→ e Tutti positivi ?

ottenerei tutte funzioni simili che "guardano" in modo diverso il singolo elemento

FORALL

predicato = funzione che valuta un elemento e dà un risultato booleano

```
# let rec forall p e =
```

```
  match e with
```

```
    [] → True
```

```
  | x::xs → if not (p x) then false
             else forall p xs ;;
```

forall: ($\underbrace{a \rightarrow \text{bool}}_p$) $\rightarrow$ ($\underbrace{a \text{ list}}_e$) $\rightarrow$ ($\underbrace{\text{bool}}_{\text{risultato}}$) = (fun)

```
# let pari x = x mod 2 = 0 ;;
```

```
# forall pari e ;;
```

corrisponde a 'Tutti pari'

```
# let dispari x = x mod 2 != 0
```

```
# forall dispari e ;;
```

```
# let positivo x = x > 0 ;;
```

```
# forall positivo e ;;
```