FUNZIONI RICORSIVE SU LISTE

- ESEMPIO (SOMMA DEGLI ELEMENTI)

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

sum : int list → int = <fun>

(int list: Tipo l; int: Tipo xs)

- 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

chiamo 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 l =

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

FUNZIONI AUSILIARIE (LOCALI)

② # let rec minmax e =
match e with
x::[] → (x,x)
| x::xs → let w = minmax xs in
if x > fst w Then (x, snd w)
else if x < snd w Then (fst w, x)
else w ;;

w = (6,2) x = 8
(8,2)

MODO ALTERNATIVO DI SCRIVERE IL 2° CASO DEL PATTERN MATCHING

| x::xs → let (m1,m2) = 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) ;;
```

[3;2;5] [4;6]

[3;2] [5;4;6]

[3] [2;5;4;6]

[] [3;2;5;4;6]

=

[3;2;5] [4;6]

3::[2;5;4;6]

Ricordate ca reverse? Poteva scriverla così:

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

// SOLUZIONE CORRETTA MA INEFFICIENTE

```
#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>
```

```
#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)

Take 2 [6;4;5;2] = [6;4]

Take 0 l = []
Take n [] = []

```
# let rec Take n l =
    match (n,l) with
      (0,_) -> []
    | (_,[]) -> []
    | (n, x::xs) -> x :: (Take (n-1) xs) ;;
```

posso usare _ al posto delle variabili nei pattern quando non sono interessato al valore da fa match con quella parte di pattern

- fa match con qualunque cosa

```
Take -2 [7;5;4]
        ↓
7 :: (Take -3 [5;4])
        ↓
7:: ( 5:: Take -4 [4])
        ↓
7:: ( 5:: ( 4:: 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) -> x::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>
```

Tipo f — Tipo x — 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 ?

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

FORALL

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

```
# let rec forall p l =
    match l with
      [] -> True
    | x::xs -> if not (p x) Then false
               else forall p xs ;;
```

```
forall : ('a -> bool) -> 'a list -> bool = <fun>
```

('a -> bool): p — 'a list: l — bool: risultato

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

corrisponde a TUTTIpari

```
# let dispari x = x mod 2 != 0
# forall dispari l ;;
# let positivo x = x > 0 ;;
# forall positivo l ;;
```