Data una lista di interi vogliamo la somma degli elementi che precedono il primo ∅.

mercoledì 13 dicembre 2017 11:15

Se non ci sono elementi = ∅ la funzione restituisce la somma di tutti gli elementi della lista.

- ricorsivamente
- senza ricorsione esplicita (con folder)

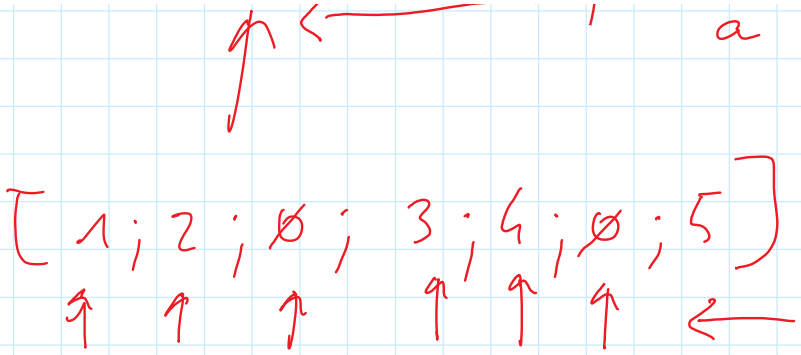
Ricorsivamente

```
let rec somma l = match l with
    [] -> ∅
  | x::xs -> if x=∅ then ∅
             else x + somma xs ;;
```

somma : int list -> int

[1; ∅; 3]

f 1 (f ∅ (f 3 a))

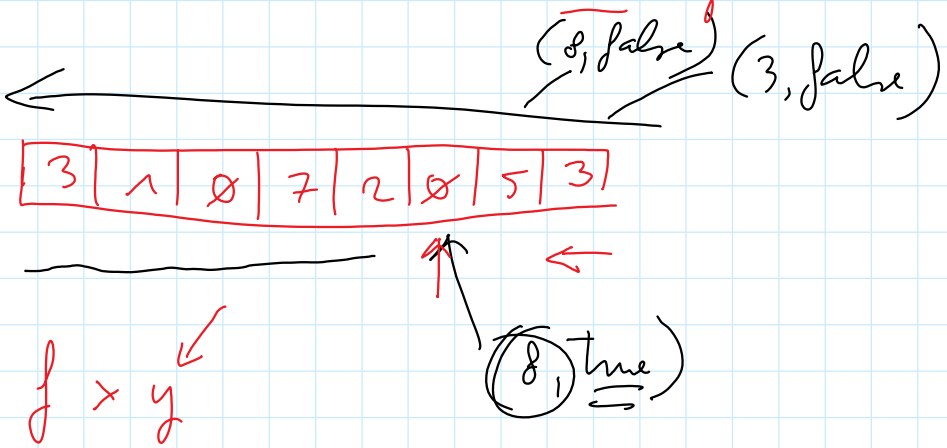


```
let somme l =
    let f x y = if x = 0 then ∅
                else x + y
    in foldr f ∅ l ;;
```

x: l'elemento di lista che stiamo esaminando

y: risultato sugli elementi di liste che seguono x

Sommare gli elementi che SEGUONO l'ultimo $\emptyset$.
Se non ci sono elementi $= \emptyset$ la funzione restituisce la somma di tutti gli elementi.



f x y

(s, b)

Somma degli elementi che seguono l'ultimo $\emptyset$

true se ho incontrato uno $\emptyset$
false altrimenti

```
let somme l =
    let f x (s, trovato) =
        if trovato then (s, trovato)
        else if x = 0 then (s, true)
        else (x + s, trovato)
```

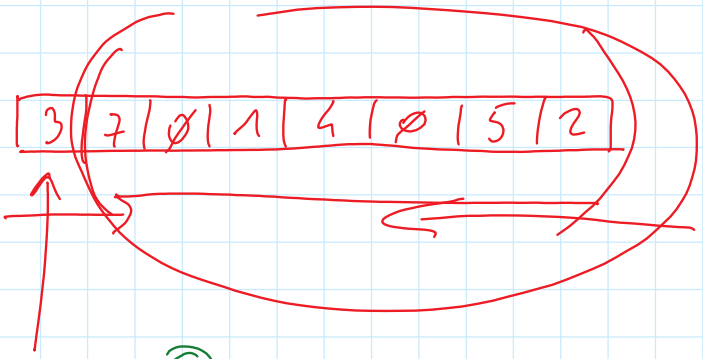
true

false

↑ false

```
in let (s,b) = foldr f (∅,false) l
   in s ;;
```

eliminare un elemento dalla coppia simultaneamente dalla foldr




```
let somma l =
  let rec f l = match l with
      [] -> (0, false)
    | x::xs -> let (s,b) = f xs
               in if b then (s,b)
                  else x=0 then (s, true)
                  else (x+s, b)
  in let (s,b) = f l
     in s ;;
```

true

false

← parametro di somma

mercoledì 13 dicembre 2017 11:49

Senza utilizzare ricorsione esplicita

minfirst : 'a list → 'a list

in modo che minfirst l restituisca una lista che contiene tutti gli elementi di l, in cui le occorrenze del minimo sono in testa. L'ordine degli elementi nel risultato è irrilevante

ES:

minfirst [2;3;4;1;3;1;6] = [1;1] @ L ←

dove L è una qualsiasi permutazione di

[2;3;4;3;6]

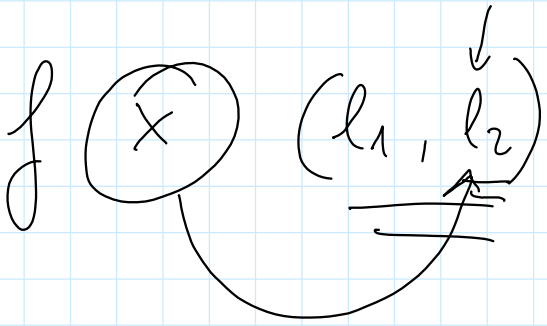
f x y ↑ ↑ liste con i minimi in testa

```
let minfirst l =
  let f x y = match y with
      [] -> [x]
    | z::zs -> if x<=z then x::y
               else y@[x]
  in fold_left f [] l;;
```

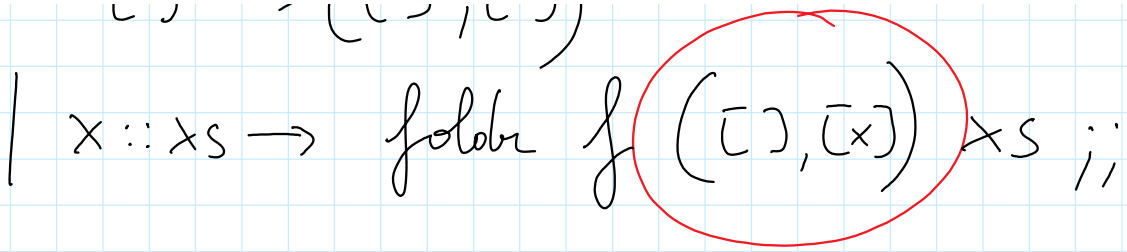
split : 'a list → 'a list * 'a list

in modo che split le restituisca la coppia (l1, l2) in cui l1 contiene tutti gli elementi di l maggiori del primo elemento di l e l2 contiene tutti quelli minori o uguali.

Se l è vuota la funzione restituisce ([],[])



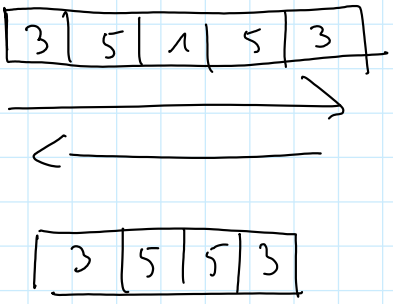

```
let split l =
    let f x (l1, l2) = match l2 with
        | y::ys -> if x <= y
                   then (l1, l2@[x])
                   else (x::l1, l2)
    in match l with
        [] -> ([], [])
      | x::xs -> foldr f ([], [x]) xs ;;
```



```
let split l = match l with
    [] -> ([], [])
  | z::zs ->
```

```
let f x (l1, l2) =
    if x <= z then (l1, x::l2)
              else (x::l1, l2)
in fold_left f ([],[]) l ;;
```

Controllare se una lista è palindroma




```
let rec ultimo l = match l with
    [x] -> x
  | x::y::ys -> ultimo (y::ys) ;;

let rec concult l = match l with
    [x] -> []
  | x::y::ys -> x :: concult (y::ys) ;;

let rec palindrome l = match l with
    [] -> true
  | [x] -> true
  | x::y::ys -> if x <> ultimo (y::ys)
                then false
                else palindrome (concult (y::ys)) ;;
```

```
let rec reverse l = .....
let rec uguali l1 l2 = match (l1, l2) with
```

```
| ([], []) -> true
| (x::xs, []) -> false
| ([], x::xs) -> false
| (x::xs, y::ys) -> if x <> y then false
                    else uguali xs ys ;;

let palindroma l =
    uguali l (reverse l) ;;
```

split : 'a list → 'a list list

in modo che split l restituisca la lista delle più lunghe sottoliste non decrescenti in l.

Es

split [1;2;2;1;5;4;6;3;2;3] =

[[1;2;2];[1;5];[4;6];[3];[2;3]]

f x y ← y è una lista di liste. La prima di queste è quella che sto costruendo

f 3 [[1;2];[1;5;6]] =

[[3];[1;2];[1;5;6]]

f ∅ [[1;2];[1;5;6]] =

[[∅;1;2];[1;5;6]]

let split l =

```
let f x y = match y with
    [] -> [[x]]
  | (z::zs)::zss -> if x<=z
                    then (x::z::zs)::zss
                    else [x]::(z::zs)::zss
in foldr f [] l ;;
```

@

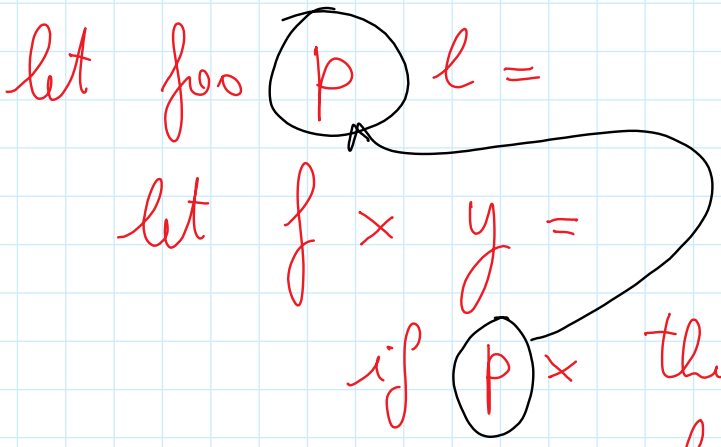
foo : 'a list → 'a * int * int

in modo che foo restituisce una terna (max, n, m) in cui max è il valore massimo in l, n è il suo numero di occorrenze e m è il numero degli elementi di l.

```
let foo l =
  let f x (max, n, m) =
    if x > max then (x, 1, m+1)
    else if x = max
      then (max, n+1, m+1)
      else (max, n, m+1)
  in match l with
    x::xs -> foldr f (x, 1, 1) xs ;;
```

foo : ('a → bool) → 'a list → 'a list

in modo che foo p l restituisca una lista in cui tutti gli elementi di l che soddisfano p precedono tutti quelli che non soddisfano p.



```
let foo p l =
  let f x y =
    if p x then x::y
           else y @ [x]
  in foldr f [] l ;;
```

```
let foo p l =
  let notp x = not p x
  in
    (filter p l) @ (filter notp l) ;;
```