FUNZIONI DI ORDINE SUPERIORE AL PRIMO

Funzioni che hanno come argomento o come risultato altre funzioni.

```
# let sum x y = x+y;;
sum : int -> int -> int = <fun>
```

Curryed

```
# let g = sum 3 ;;
g : int -> int = <fun>
```

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

('a -> 'b): tipo di f; 'a: tipo di x; 'b: tipo ris

```
# let inca x = x + 1 ;;
inca : int -> int = <fun>
```

```
# let s (x, y) = x + y ;;
s : int * int -> int = <fun>
```

apply inca 3 ;;

int -> int

apply : ('a -> 'b) -> 'a -> 'b

'a = int

'b = int

int int.

- : int = 4

apply s 3 ;;

int * int -> int

errore di tipo

apply : ('a -> 'b) -> 'a -> 'b

'a = int * int

'b = int

int * int

int

apply s (3,4) ;;

- : int = 7

corretto!

lunedì 2 dicembre 2019 09:34

Funzioni di ordine superiore che coinvolgono liste CAML

Predicato

Una funzione che restituisca un valore di verità (bool)

predicato!!

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

definizione

operatore di uguaglianza

```
pari : int -> bool = <fun>
```

```
# let sto (n,m) = n+m = 10 ;;
sto : int * int -> bool = <fun>
```

```
# sto (3,7) ;;
- : bool = true
```

```
# sto (5,6) ;;
- : bool = false
```

Funzione che verifica che un predicato sia vero su tutti gli elementi di una lista.

forall pari [2;8;10] = true

forall pari [2;3;8;10] = false

forall sto [(5,5);(7,3)] = true

~~forall pari [(5,5);(7,3)]~~ → errore di tipo

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

unbound

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

tipo p — tipo l — tipo xs

Quantificazione esistenziale

exists p l

vale true se p è vero su almeno un elemento della lista l.

exists p [3;4;5;10] = true

```
# let rec exists p l =
        match l with
        [] -> false
      | x::xs when not p x -> exists p xs
      | x::xs when p x -> true ;;
exists : ('a -> bool) -> 'a list -> bool = <fun>
```

map f l

funzione lista

il risultato è la lista in cui gli elementi sono ottenuti applicando la funzione f a tutti gli elementi di l.

map inc [3;4;5] = [4;5;6]

```
# 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

```
# map inc [3;4];;
```

int -> int

'a = int

'b = int

map : ('a -> 'b) -> 'a list -> 'b list

'b = int

- : int list = [4; 5]

map pari [3; 4];;

int → bool

'a = int

'b = bool

- : bool list = [false; true]

```
let apply (f,x) = f x ;;
apply : ('a -> 'b) * 'a -> 'b = <fun>
```

tipo di f · tipo x · tipo ris

```
# map apply [(inc,3);(inc,4)] ;;
```

map : $('a \to 'b) \to 'a\ list \to 'b\ list$

apply : $('a \to 'b) * 'a \to 'b$

$'a = ('a \to 'b) * 'a$

$'b = 'b$

```
# map apply ;;
- : (('a -> 'b) * 'a) list -> 'b list
```

lista di coppie in cui il primo elemento è una funzione ('a -> 'b) e il secondo elemento è un valore

di tipo 'a

```
map apply [(inc, 3); (inc, 4)];;
- : int list = [4; 5]
```

filter p l

p: funzione che restituisce un valore di verità (bool)

l: lista

il risultato di filter p l è la lista che contiene gli elementi di l in cui il predicato p è vero!

filter pari [3;4;5;6;7;8] = [4;6;8]

```
#let rec filter p l =
        match l with
          [] -> []
        | x::xs when p x -> x::filter p xs
        | x::xs when not p x -> filter p xs;;
```

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

('a -> bool): tipo p; 'a list: tipo l; 'a list: tipo xs

```
# filter sto [(3,5);(5,5);(6,4);(7,2)];;
```

// ...

~: int * int list = [(5,5), (6,4)]

foldr riduce una lista, mediante una funzione f, partendo da destra

foldr f a $[x_1; x_2; \ldots; x_n]$ =

f x_1 (f x_2 (... (f x_n a) ...))

elementi di lista

risultato nelle parti rimanenti della lista

```
f x y = 1 + y
```

foldr f ∅ [3;4;5] =

f 3 (f 4 (f 5 ∅))

1+0

(f 4 1)

$1+1=2$

f 3 2 = $1+2=3$

f x y = 1 + y

elemento di liste

risultato su tutti gli elementi che seguono

foldr f ø [3;4;5]

---

f x y = 1 + y

foldr f ø [3;4;5] è la lunghezza delle liste

l'elemento a è il risultato sulle liste vuote

f x n a

```
g x y = x + y
```

Qual'è il risultato di

```
foldr g 0 [3;4;5]
```

?

```
foldr g 0 [3;4;5] =
g 3 (g 4 (g 5 0)) = 12
```

Nella foldr la funzione che viene usata ha due argomenti:

```
f x y
```

dove

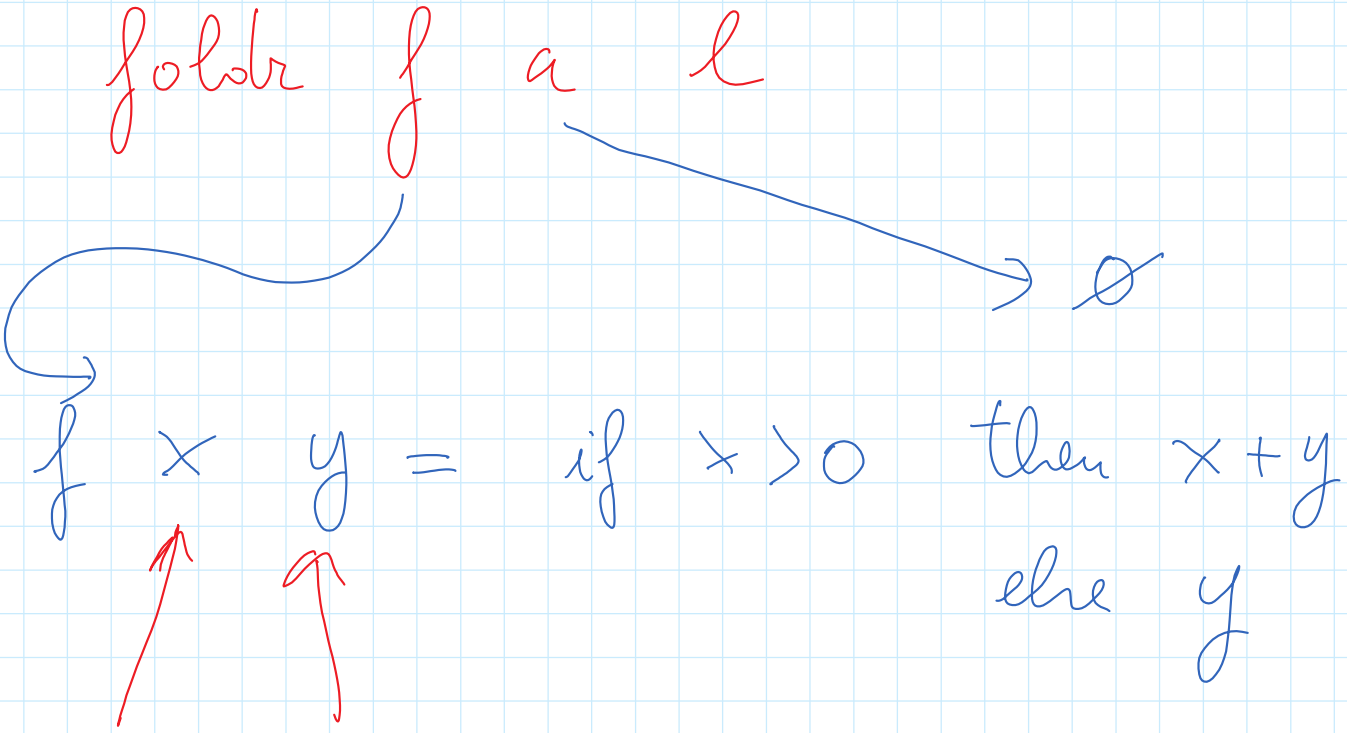
x è l'elemento di liste che stiamo considerando

y è il risultato su tutti gli elementi di liste che seguono x

↳ che vogliamo ottenere

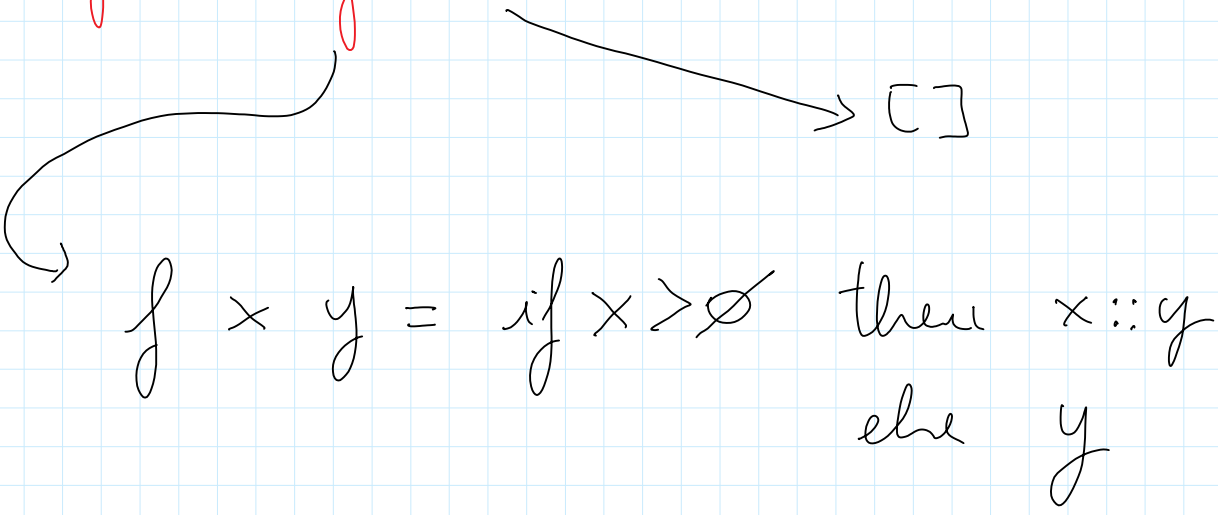
↳ che vogliamo ottenere

Vogliamo la somma di tutti i valori > 0 di una lista



la liste dei soli element > ∅ di l

foldr f a l



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

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

tipo f — tipo a — tipo l — tipo xs

lunedì 2 dicembre 2019 10:54

Definire la funzione filter usando foldr (senza ricorsione)

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

x è l'elemento di lista che stiamo guardando

y è il risultato sulla lista che segue x

map usando foldr

```
let map f l =
    let g x y = f x :: y
    in foldr g [] l ;;
```

---

map inca [3;4]

=

foldr g [] [3;4] g = inca

= { def. foldr }

g 3 (foldr g [] [4])

= { def foldr }

g 3 (g 4 (foldr g [] []))

= { def foldr }

g 3 (g 4 [])

$= \{dy\ g\}$

$g\ 3\ [5)$

$= \{dy\ g\}$

$[4;5]$