- Funzione che restituisca il valore dell'ultimo elemento di una lista

- Funzione che cancelli l'ultimo elemento di una lista.

- Funzione che dà la lunghezza di una lista

- Funzione che dà la somma di tutti gli elementi di una lista di interi

- Funzione che cancella tutti gli elementi negativi da una lista di interi

Lunghezza di una lista

```
# let rec len l =
    if l = [] then 0
    else len (tl l) + 1 ;;
```

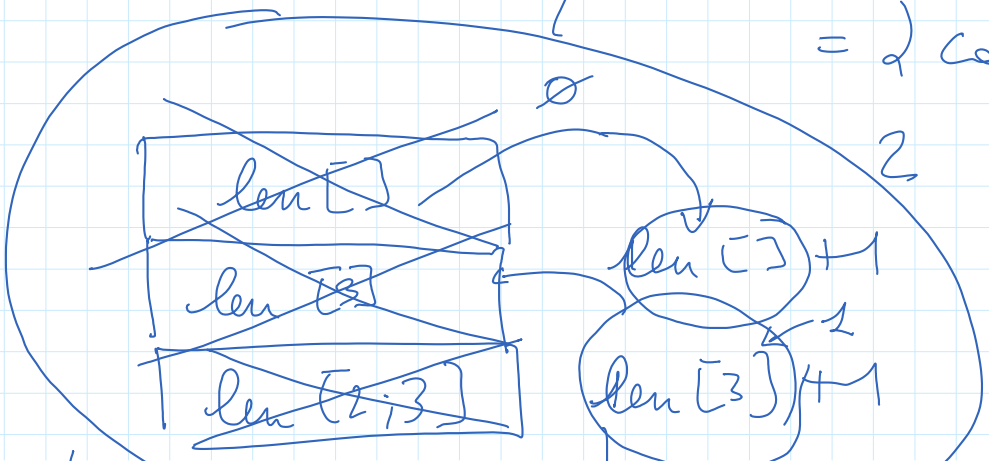
```
len : 'a list -> int = <fun>
```

'a list: tipo di l — int: tipo del risultato

```
# len [2;3];;
- : int = 2

# len ['a';'b';'c'];;
- : int = 3
```

len [2;3];;
= { def len, l = [2;3] }
len [3] + 1
= { def len, l = [3] }
len [] + 1 + 1
= { def len, l = [] }
0 + 1 + 1
= { calcolo }
2




→ 2

Somma degli elementi di una lista

```
# let rec sum l =
    if l = [] then 0
    else sum (tl l) + hd l ;;
```

+. somma float

```
sum : int list -> int = <fun>
```

tipo di l — tipo ris.

```
sum [2;3] ;;
= {def sum, l = [2;3]}
  sum (tl l) + hd l
= {}
  sum [3] + 2
= {def sum, l = [3]}
  sum (tl [3]) + hd [3] + 2
= {}
  sum [] + 3 + 2
= {def sum, l = []}
  0 + 3 + 2
```

$= \{calcola\}$

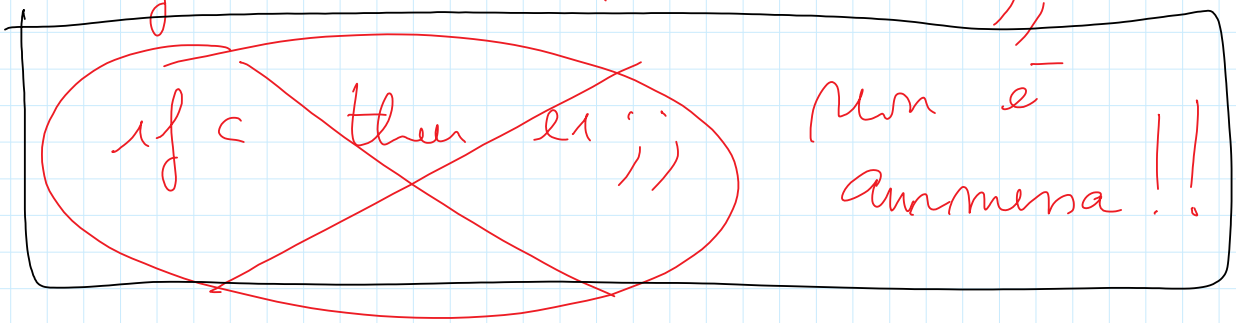
5

Cancellare tutti gli elementi negativi.

```
# let rec delneg l =
    if l=[] then []
      else if hd l < 0 then delneg (tl l)
        else hd l :: delneg (tl l);;
delneg : int list -> int list = <fun>
```

tipo di l — tipo NS

In CAML non posso avere una espressione if senza il ramo else.



delneg [3;-4;5] ;;

= { def delneg; l = [3;-4;5] }

hd [3;-4;5] :: delneg (tl [3;-4;5])

= { applico hd e tl }

3 :: delneg ([-4;5])

= { def. delneg, l = [-4;5] }

3 :: delneg [5]

= { def delneg, l = [5] }

3 :: (hd [5] :: delneg (tl [5]))

= { appl. hd e tl }

3 :: 5 :: delneg []

= { def. delneg, l = [] }

3 :: 5 :: []

= { definizione }

$= \{ \text{motivazione} \} \Big) =$

$[3;5]$

PATTERN MATCHING

pattern: una espressione che contiene variabili (nomi).

Le variabili possono essere istanziate a valori con tipo corretto!

Es: `x::[]` pattern

Un pattern non può contenere una variabile più di una volta

`x::x::[]` non è ammesso!!

Instanziazione delle variabili di un pattern

Es: `x::[]`

può essere instanziato a

`5::[]` instanziando x a 5

$(3,4) :: [\,]$ " x a $(3,4)$

Che cosa vuol dire fare pattern matching.

Vuol dire instanziare le variabili di un pattern per uguagliarlo a una espressione!

Es posso uguagliare il pattern `x::[]` alle liste `[5]`?

Sì, perché `[5] = 5::[]` se instanzio `x` a 5 ottengo `5::[] = [5]`

i pattern `x::[]` e `[x]` sono uguali

posso uguagliare il pattern

$[x)$ $(x::[])$

alle liste

$[4;5)$?

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

quindi basta istanziare x a $4::5$

↳ Valore CAML

errore di tipo!!

Il pattern $[x]$ $(x::[])$ può essere uguagliato solamente a liste lunghe 1.

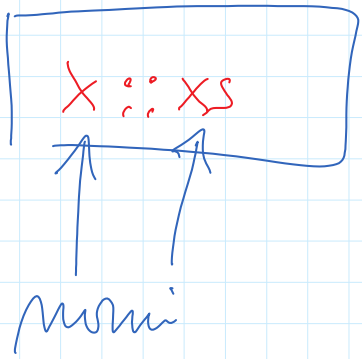
Uguagliare

$[x)$ a $[(3,5)]$ $x = (3,5)$

$[x)$ a $[[1;2]]$ $x = [1;2]$

9

pattern



xs ricorda che xs si uguaglia a una lista

x::xs si può uguagliare a tutte le liste escluso la lista vuota

I pattern si usano nelle espressioni match

match espressione with
  pattern1 -> esp.1
| pattern2 -> esp.2
  ⋮
| patternn -> esp.n

Si cerca di uguagliare ai pattern in sequenza.

quando si riesce a uguagliare ad un pattern, il risultato dell'intera espressione è il valore dell'esp. dopo la freccia.

ES:

```
# match [3;4] with
    [] -> 0
  | x::xs -> 1 ;;
```

$x = 3$
$xs = [4] = 4::[]$

```
- : int = 1
```

```
# match [3;4] with
    [] -> 0
  | x::xs -> x ;;
```

$x = 3$
$xs = [4]$

```
- : int = 3
```

```
# match [3;4] with
    [] -> 0
  | x::xs -> xs ;;
```

devono avere lo stesso tipo

$x = 3$
$xs = [4]$

errore di tipo

sono tipici

$x::[] = [x]$ si unifica a liste

`x :: [] = [x]` si unifica a liste lunghe 1

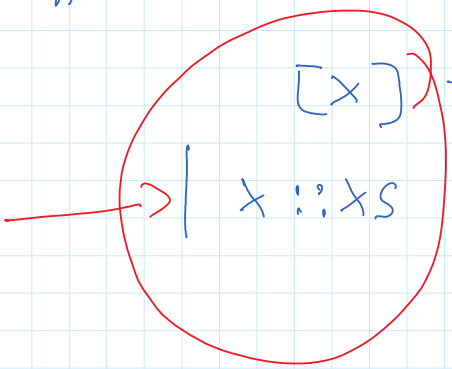
`x :: y :: [] = x :: [y] = [x, y]` si unificano solamente a liste lunghe 2

`x :: xs` si unifica a liste non vuote

`x :: (y :: ys)` si unifica a liste di almeno 2 elementi

last

#let rec last l = match l with



last (tl l)

non sono mutuamente esclusivi

[x] e x::xs non sono mutuamente esclusivi perché entrambi possono essere uguagliati alla lista [5] = 5::[]

giovedì 21 novembre 2019 17:21

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

dellast

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

---

len

```
# let rec len l = match l with
    [] -> 0
  | x::xs -> len xs + 1;;
```

1 + len xs uguale

delneg

```
# let rec delneg l = match l with
    [] -> []
  | x::xs -> if x<0 then delneg xs
             else x :: delneg xs ;;
```

Quando when

| pattern when condizione -> espress.

pattern uguaglia il valore
e la condizione è Vera!

```
let rec delneg l = match l with
    [] -> []
  | x::xs when x<0 -> delneg xs
  | x::xs when x>=0 -> x::delneg xs ;;
```

Operatore @ predefinito che è la concatenazione tra liste.
Operatore infisso

```
# [3;4] @ [5;-2;7];;
- : int list = [3;4;5;-2;7]
```

Concatenazione (append)

```
# @ ;;
errore di sintassi
```

prefix (predefinito) fa diventare un operatore infisso una funzione.

```
# prefix @ ;;
- : 'a list -> 'a list -> 'a list = <fun>
```

Curryed

```
# (prefix @) [3;4] [5;6];;
- : int list = [3;4;5;6]
```

```
- : int list = [3; 4; 5; 6]

# let g = prefix @ [3; 4] ;;
g : int list -> int list = <fun>

# g [4; 5] ;;
- : int list = [3; 4; 4; 5]
```

giovedì 21 novembre 2019 17:41

proprietà di @

1) $l \,@\, [\,] = [\,] \,@\, l = l$

2) $x :: (l_1 \,@\, l_2) = (x :: l_1) \,@\, l_2$