Concatenazione tra liste

@ operatore predefinito infisso

[3;4] @ [3;4;5] = [3;4;3;4;5]

append.

prefix operato predefinito prefisso
che fa diventare un operatore
una funzione

@;;
errore

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

prefix @ [3;4] [3;4;5];;
- : int list = [3;4;3;4;5]

prefix +;;
- : int -> int -> int = <fun>

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

2) $x :: (l1 @ l2) = (x :: l1) @ l2$

@ vogliamo definirle come funzione CAML

```
#let rec append l1 l2 =
        match l1 with
        [] -> l2
      | x::xs -> x :: append xs l2;;
append : 'a list -> 'a list -> 'a list = <fun>
```

tipo di l1 — tipo di l2 — tipo ris.

append [3;4] [3;4;5] ;;

= { def. append, l1 = [3;4], l2 = [3;4;5] }

3 :: append [4] [3;4;5]

= { def. append, l1 = [4], l2 = [3;4;5] }

3 :: 4 :: append [] [3;4;5]

= { def. append, l1 = [], l2 = [3;4;5] }

3 :: 4 :: [3;4;5]

= { notazione }

[3;4;3;4;5]

$(\forall l_1, l_2 \in$ 'a list.
append l_1 l_2 $= l_1 @ l_2)$?

```
# let rec append (l1, l2) =
        match l1 with
    [] -> l2
  | x::xs ->  x:: append (xs, l2);;
append : 'a list * 'a list -> 'a list = <fun>
```

$(\forall l_1, l_2 \in$ 'a list. append $(l_1, l_2) = l_1 @ l_2)$

intuitivamente

$(xs, l_2) \prec (l_1, l_2)$ dove $l_1 = x :: xs$

formalmente

$(\forall l_s, l_2, l_s', l_2'.$
$(l_1, l_2) \prec (l_1', l_2') \equiv$

$l1 = l1' \wedge l2 = l2'$)

ben fondata? sì

($\forall$ l1, l2, l1', l2' $\in$ 'a list.

(l1, l2) $\prec$ (l1', l2') $\equiv$

($\exists$ x $\in$ 'a. $\exists$ xs $\in$ 'a list.

l1' = x :: xs $\wedge$ l1 = xs $\wedge$ l2 = l2'))

$$(\forall l_1, l_2, l_1', l_2'.\ (l_1, l_2) \prec (l_1', l_2') \equiv l_1 = tl\ l_1' \wedge l_2 = l_2')$$

Quali sono i minimali?

tutte le coppie $([\,], l_2)$

↑ quantiamo!!

Induzione ben fondata su

$('a\ list * 'a\ list, \prec)$

per dimostrare

$$(\forall l_1, l_2 \in 'a\ list.\ append\ (l_1, l_2) = l_1 @ l_2)$$

Caso base

$$append\ ([\,], l_2) = [\,] @ l_2$$

Caso induttivo

$$append\ (l_1, l_2) = l_1 @ l_2$$

$$\Rightarrow$$

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

Caso base

append ([], l2) = [] @ l2

append ([], l2)
= { def append }
l2
= { proprietà 1) di @, l = [] @ l = l @ [] }
[] @ l2

Caso induttivo

append (l1, l2) = l1 @ l2 ⇒ append (x :: l1, l2) = (x :: l1) @ l2

ipotesi induttiva

append (x :: l1, l2)
= { def. append, x :: l1 ≠ [] }
x :: append (l1, l2)
= { ip. induttiva: append (l1, l2) = l1 @ l2 }
x :: (l1 @ l2)
= { proprietà 2) di @: x :: (l1 @ l2) = (x :: l1) @ l2 }
(x :: l1) @ l2

hd

generalizzare hd a una funzione che restituisce i primi n elementi di una lista (se ci sono)

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

take 2 [3] = [3]

```
# let rec take n l =
    match (n, l) with
```

(0, _) → []

Sono matematicamente equivalenti?

| (n, []) when n>0 → []

qualunque valore che non posso utilizzare nel risultato (perché è senza nome)

```
| (n, x::xs) when n>0
    -> x :: take (n-1) xs ;;
```

take : int → 'a list → 'a list = <fun>

tipo n tipo l tipo rs

generalizzazione della tl

drop n l cancella dalle liste l i primi n elementi (se ci sono)

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

drop 2 [3] = []

```
# let rec drop n l =
        match (n,l) with
  (0,l) -> l
| (n,[]) when n>0 -> []
| (n,x::xs) when n>0 -> drop (n-1) xs;;
drop: int -> 'a list -> 'a list = <fun>
```

Funzione che restituisce l'n-simo elemento di una lista.

Se non esiste l'n-simo elemento la funzione è indefinita.

n_simo 2 [3;4;5] = 4

n_simo 3 [2;3] indefinita!!

```
# let rec n_simo n l =
    match (n,l) with
    (0,l) ->
    (n,[]) when n>0 ->
    (n,x::xs) when n>0 ->
        if n=1 then x
        else n_simo (n-1) xs;;
n_simo : int -> 'a list -> 'a = <fun>
# (1,x::xs) -> x
```

```
let rec n_simo (n, x::xs) = x
  | (n, x::xs) when n > 1 ->
      n_simo (n-1) xs ;;
```

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

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

$$\left(\forall n \in \mathbb{N}.\ \forall l \in \text{'a list}.\ \texttt{take}\,(n,l) \mathbin{@} \texttt{drop}\,(n,l) = l\right)$$

Il dominio è il dominio della coppia in $\mathbb{N} \times$ 'a list

intuitivamente

$$(n-1, xs) < (n, x::xs)$$

Formalmente

$$\Big(\forall n, n' \in \mathbb{N}.\ \forall \ell, \ell' \in 'a\ list.$$

$$(n, \ell) \prec (n', \ell') \equiv$$

$$n = n' - 1 \ \wedge\ \ell = tl\ \ell'\Big)$$

es

$$(4, [3;4;5])$$

$$\downarrow a$$

$$(3, [4;5])$$

$$\downarrow a$$

$$(2, [5])$$

$$\downarrow a$$

$$(1, [])$$

$$\not\downarrow$$

I minimali:

$$(\varnothing, \ell) \qquad (n, [])$$

$(\mathbb{N} \times 'a\ list, \prec)$ è ben fondato!

$(\forall n \in \mathbb{N}.\ \forall l \in 'a\ list.\ take(n,l) @ drop(n,l) = l)$

Induzione ben fondata con $\prec$

Casi base $(\varnothing, l)$ $(n, [])$

Caso base 1: $take(\varnothing, l) @ drop(\varnothing, l) = l$

$take(\varnothing, l) @ drop(\varnothing, l)$

$= \{$ def. take, def drop $\}$

$[] @ l$

$= \{$ proprietà 1) di $@\}$

l

Caso base 2: $take(n, []) @ drop(n, []) = []$

$take(n, []) @ drop(n, [])$

$= \{$ def. take, drop $\}$

$[] @ []$

$= \{$ proprietà 1) di $@\}$

$[]$

Caso induttivo ipotesi induttiva

$take(n,l) @ drop(n,l) = l$

$\Rightarrow$

$take(n+1, x::l) @ drop(n+1, x::l) = x::l$

$take(n+1, x::l) @ drop(n+1, x::l)$

$= \{$ def. take, drop; $n+1 > 0 \}$

$(x :: take(n,l)) @ drop(n,l)$

$= \{$ proprietà 2) di @ : $(x::l1) @ l2 = x::(l1 @ l2) \}$

$x :: (take(n,l) @ drop(n,l))$

$= \{$ ipotesi induttiva $\}$

$x :: l$

Cercare il valore massimo di una lista.
Funzione non definita su []

```
# let rec max l =
    match l with
    [] ->
    [x] -> x
  | x::y::ys ->
      if x > max (y::ys)
      then x
      else max (y::ys) ;;
max : 'a list -> 'a = <fun>
```

Dichiarazioni locali

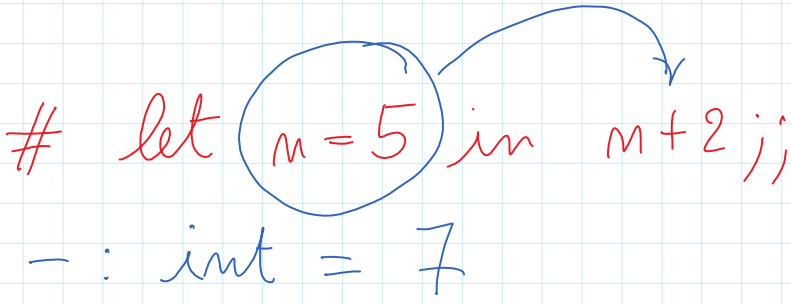
let ... in ...

↑ dichiarazioni

↖ espressione

nome = valore

Le associazioni sono locali alla valutazione della espressione




```
# n ;;
errore
```

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