lunedì 27 novembre 2017 09:19

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

precedenza INDOTTA della definizione ricorsiva

$(n-1)\ xs \prec n\ x::xs$

take: int → 'a list → 'a list = <fun>

Misurando $(n-1)\ xs \prec n\ x::xs$

$(0, l)$ $(n, [])$

Casi base

$P(n-1, xs) \Rightarrow P(n, x::xs)$

Caso induttivo

lunedì 27 novembre 2017 09:28

```
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>

precedenza minore è la stessa della def. di take

$\forall n \in \mathbb{N}$, $l \in$ 'a list.

$$(\text{take } n\ l) @ (\text{drop } n\ l) = l$$

Casi base

1) coppia $\varnothing$ l (infinita coppia)
2) coppia n $[]$ ↙

1) $(\text{take } \varnothing\ l) @ (\text{drop } \varnothing\ l) = l$

$(\text{take } \varnothing\ l) @ (\text{drop } \varnothing\ l)$
$= \{$ def. take, 1° pattern $\}$
$[] @ (\text{drop } \varnothing\ l)$
$= \{$ def. drop, 1° pattern $\}$
$[] @ l$
$= \{$ proprietà $@$: $[] @ l = l @ [] = l \}$
l

2) $(\text{take } n\ [])@(\text{drop } n\ []) = []$ $\quad n > \varnothing$

$(\text{take } n\ [])@(\text{drop } n\ [])$

$$
\begin{aligned}
&= \{ \text{def. take, } 2^{\circ}\ p. \} \\
&\quad [\,] \mathbin{@} (\text{drop } n\ [\,]) \\
&= \{ \text{def. drop, } 2^{\circ}\ p \} \\
&\quad [\,] \mathbin{@} [\,] \\
&= \{ \text{nop. } @ \} \\
&\quad [\,]
\end{aligned}
$$

Caso induttivo

$(take\ (n-1)\ xs) \mathbin{@} (drop\ (n-1)\ xs) = xs$ ip. induttiva

$\Rightarrow$

$(take\ n\ x::xs) \mathbin{@} (drop\ n\ x::xs) = x::xs$

$n > 0$

$(take\ n\ x::xs) \mathbin{@} (drop\ n\ x::xs)$

$=$ { def. take, 3° pattern }

$(take\ (n-1)\ xs) \mathbin{@} (drop\ n\ x::xs)$

$=$ { def. drop, 3° p. }

$(x :: take\ (n-1)\ xs) \mathbin{@} (drop\ (n-1)\ xs)$

$=$ { proprietà @ : $(x::xs) \mathbin{@} l = x :: (xs \mathbin{@} l)$ }

$x :: ((take\ (n-1)\ xs) \mathbin{@} (drop\ (n-1)\ xs))$

$=$ { ip. induttiva }

$x :: xs$

Valore massimo di una lista di interi

lunedì 27 novembre 2017 09:47

max [-2; 3; 5; 7; -3] = 7

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

non è definita su lista vuota

è definita per tipi diversi

x::[]

uguaglia a liste di almeno due elementi

max : 'a list → 'a = <fun>

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

---

dichiarazioni locali

```
let (legami) in expr ;;
```

(legami: nome, valore)

questi legami sono conosciuti solamente durante la valutazione di questa espressione

es:

```
let x = 5 in x + 1 ;;
- : int = 6
```

dichiarazione locale

```
let x = 5 ;;
x : int = 5
```

```
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 ;;
```

Definizioni ricorsive con "accumulatore".
L'accumulatore è un "argomento" in più alle funzioni che "accumula" il risultato parziale nelle chiamate ricorsive.

Es. max con "accumulatore"

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

In questo caso l'accumulatore mi tiene

il valore massimo incontrato fino a quel momento.

TAIL RECURSION

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

max : 'a list → 'a → 'a = <fun>

max [3;4;1] ∅
= { def. max, 2° p }
x = 3 a = ∅

max [4;1] 3
= { def max, 2° p }
x = 4 a = 3

max [1] 4
= { def max, 1° p }
4

max [-2;-10;-3] ∅
=
⋮
∅

max [-2;-10;-1] -2
⋮
= -1

Anche se definite con accumulatore, io
vorrei poter chiamare la funzione max
dovendogli come unico argomento la
lista di cui trovare il massimo.

```
max [-2; -10; -1]
```

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

lunedì 27 novembre 2017 10:38

Vogliamo definire una funzione che restituisca il massimo e il minimo di una lista (non vuota)

(max, min) (int * int)

```
let rec maxmin l = match l with
    [x] -> (x, x)
  | x::y::ys -> let (m, n) = maxmin (y::ys)
                in if x > m then (x, n)
                   else if x < n then (m, x)
                   else (m, n);;
maxmin : 'a list -> 'a * 'a = <fun>
let (m,n) = (3,4);;
m : int = 3
n : int = 4
```

Invertire l'ordine degli elementi di una lista

lunedì 27 novembre 2017 10:48

reverse

reverse [3;4;5] = [5;4;3]

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

lista — elemento

NO

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

[5;6] :: 3;;
errore di tipo
```

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

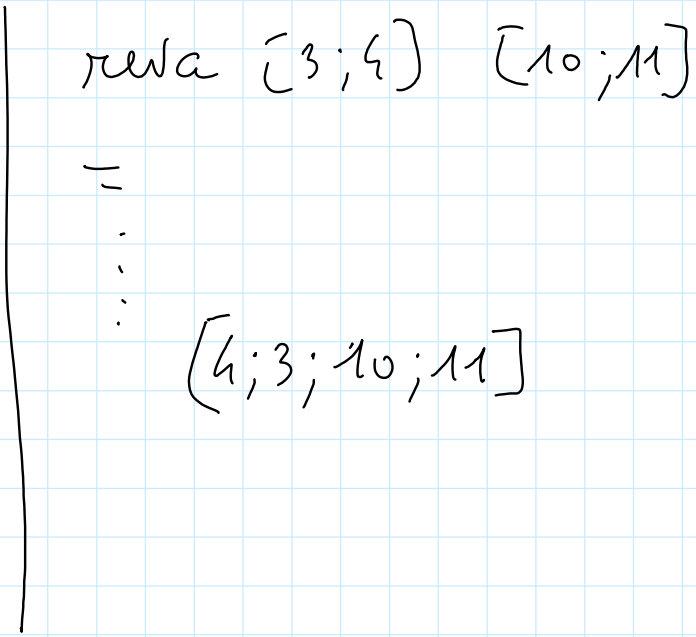
lunedì 27 novembre 2017 10:49

```
reverse : 'a list -> 'a list = <fun>

reverse [(3,4) ; (4;5)] ;;
- : (int * int) list = [(4,5); (3;4)]
```

Reverse con accumulatore rev a

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


rwa (3;4) [10;11]
=
(4;3;10;11]

lunedì 27 novembre 2017 10:57

```
let reverse l =
    let rec rev a l =
        ...
    in rev l [] ;;
```

reverse : 'a list → 'a list = <fun>

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

dato che questa è una operazione costosa non conviene scrivere le take con accumulatore.