Esercizi CAML

Accumulatore nelle definizioni ricorsive

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

↑ costosa a causa della concatenazione

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

```
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;;
```

(l, n)

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

ineffiziente!

```
take 2 [3;4;5] []
= take 1 [4;5] [3]
```

=

Tolgo a ~~$\emptyset$~~ $[5]$ $[4;3]$ ris

member n l

vale true sse n compare in l

```
let rec member n l = match l with
    []      -> false
  | x::xs   -> if x = n then true
               else member n xs;;
```

member : 'a → 'a list → bool = <fun>

'a: tipo n; 'a list: tipo l; bool: tipo xs

```
let rec member n l = match l with
    []          -> false
  | x::xs when n = x -> true
  | x::xs when n <> x -> member n xs;;
```

member : 'a → 'a list → bool = <fun>

```
let rec member x l =
    if l = [] then false
    else if x = hd l then true
    else member x (tl l);;
```

```
member : 'a -> 'a list -> bool = <fun>
```

Controllare se una lista è ordinata in modo crescente

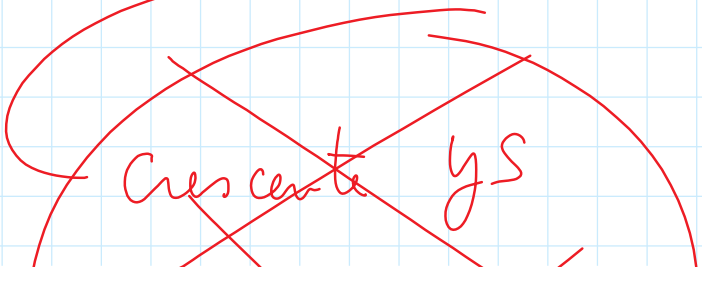
[1; 5; 15; 21] ⟶ true

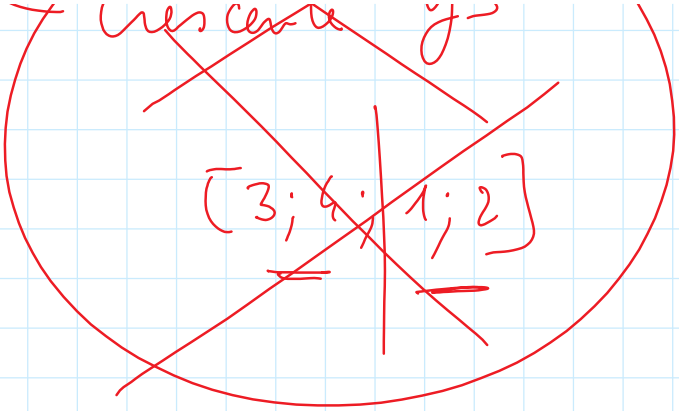
[1; 5; 21; 15] ⟶ false

[2; 2] ⟶ false

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

crescente : 'a list → bool = <fun>




[3; 4; 1; 2]

Contare in una lista gli elementi uguali a el

```
let rec conta el l = match l with
    [] -> 0
  | x::xs ->
      if x = el then 1 + conta el xs
                else conta el xs ;;
conta : 'a -> 'a list -> int = <fun>
```

```
let conta el l =
    let rec conta a el l a = match l with
                [] -> a
        | x::xs ->
            if x = el then conta el xs (a+1)
                else conta el xs a
    in conta el l 0 ;;
```

inserire un elemento in una lista ordinata in modo che la lista risultante sia ancora ordinata, in modo non decrescente.

6 [3; 5; 7; 15] →
[3; 5; 6; 7; 15]

```
let rec insord n l = match l with
    [] -> [n]
  | x::xs ->
      if n > x then x::((insord n) xs)
      else (n::x::xs);;
```

n :: l | ok

insord : 'a → 'a list → 'a list = <fun>

RIVELAZIONE!!
non tutti gli operatori si possono usare nei pattern!!

usare nei "pattern"!!

match n with

x+1 →

No!

---

match l with

l1 @ l2 →

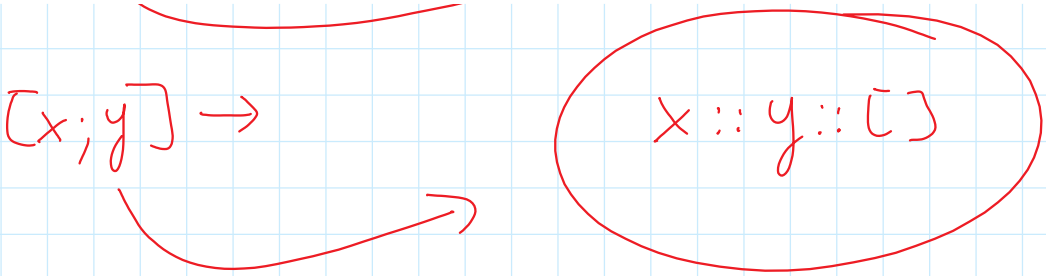
No!

Si possono usare nei pattern solamente gli operatori costruttori di valori (operatori di base).

Servono per costruire valori di un tipo: per le liste i costruttori di valori sono la costante [] e l'operatore ::

match l with

[x] → x::[]


[x;y] →
x :: y :: []

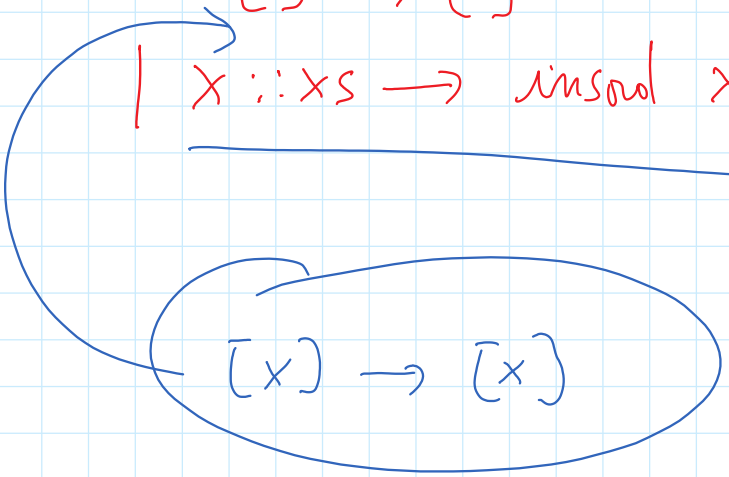
giovedì 29 novembre 2018 17:06

Vogliamo ordinare una lista mediante insord n l

- la lista vuota è ordinata
- se vogliamo ordinare una lista x::xs possiamo ordinare ricorsivamente xs e nel risultato inserire x mediante insord

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

Ok



ci può non mettere

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

```
let rec sort l = match l with
    [] -> []
  | x::xs -> insord x (sort xs);;
sort : 'a list -> 'a list = <fun>
```

sort [2; -1; 4; 3]

= {def sort 2° p}

insord 2 (sort [-1; 4; 3])

= {def sort, 2° p}

insord 2 (insord -1 (sort [4; 3]))

= {def sort, 2° p}

insord 2 (insord -1 (insord 4 (sort [3])))

= { " }

insord 2 (insord -1 (insord 4 (insord 3 (sort []))))

= {def sort, 1° p}

is 2 (is -1 (is 4 (is 3 [])))

= {def insord}

$$
\begin{aligned}
&= \{ \text{def unsnoc} \} \\
&\text{is } 2\ (\text{is } -1\ (\text{is } 4\ [3])) \\
&= \{ \; // \; \} \\
&\text{is } 2\ (\text{is } -1\ [3;4]) \\
&= \{ \; // \; \} \\
&\text{is } 2\ [-1;3;4] \\
&= \{ \; // \; \} \\
&\quad [-1;2;3;4]
\end{aligned}
$$

```
let sort l =
  let rec sorta l a = match l with
      [] -> a
    | x::xs -> sorta xs (insord x a)
  in sorta l [];;
```

```
sorta [2;-1;4;3] []
= { def sorta, 2° p }
sorta [-1;4;3] [2]
= { " }
sorta [4;3] [-1;2]
= { " }
sorta [3] [-1;2;4]
= { " }
sorta [] [-1;2;3;4]
```

$= \{ \text{def sota}, 1^{\circ} p \}$

$[-1; 2; 3; 4]$

Si definisca in CAML una funzione

cancella : int → 'a list → 'a list

che, dato un intero n e una lista, elimina gli ultimi n elementi della lista (se ci sono)

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

let rec cancella n l =
    if len l <= n then []
    else match l with
        x::xs -> x:: cancella n xs ;;
```

```
let rec cancella n l =
    if len l <= n then []
    else hd l :: cancella n (tl l) ;;
```

inserisci : 'a list → 'a → 'a → 'a list

tale che

inserisci l x y

inserisce nella lista l una nuova occorrenza del valore x immediatamente dopo l'ultima occorrenza del valore y.

Se y non è presente nella lista, x non viene inserito.

inserisci [3;4;5;4;2] 2 4 =

[3;4;5;4;2;2]

inserisci [3;4;5;4;2] 2 7 =

[3;4;5;4;2]

```
let rec inserisci l x y = match l with
    [] -> []
  | z :: zs ->
      if z = y then
        if not (member y zs)
        then z :: x :: zs
        else z :: inserisci zs x y
      else y :: x :: zs ;;
```

```
let rec l = .....

let ins l x y = match l with
    [] -> []
  | z::zs ->
      if z=y then x::z::zs
      else z:: ins zs x y;;

let inserisci l x y =
    rev (ins (rev l) x y);;
```