ESERCIZI CAML

Titolo nota 03/12/2015

Scrivere una funzione RICORSIVA che data una lista ORDINATA inserisca un nuovo elemento mantenendo l'ordinamento

insord [1;2;3;4;6;7;10] 5 = [1;2;3;4;5;6;7;10] "<"

insord [1;2;3;4;6;7;10] 4 = [1;2;3;4;4;6;7;10]

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

```
x::z::xs -> if y>x & y<z then x::y::z::xs else insord (z::xs) y
```

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

si può omettere

```
match l with
x::xs when y<x -> y::l
```

dopo il pattern matching
l e x::xs sono "la stessa cosa"
x e (hd l) " "
xs e (tl l) " "

Scrivere una funzione che ordina una lista, utilizzando insord

ord [3; 1; 4; 2] = [1; 2; 3; 4]

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

```
x::xs   ->  insord (ord xs) x ;;
```

```
let rec ord l =
    let rec insord x l = match l with
          []    ->  [x]
        | y::ys -> if x<y then x::y::ys
                          else x::(insord x ys)
    in   match l with
           []    ->  []
         | x::xs ->  insord x (ord xs);;
```

ord [1;3;2]

= { 2° pott. di ord , x=1, xs=[3;2] }
insord 1 (ord [3;2])

= { 2° pott. di ord x=3, xs=[2] }
insord 1 (insord 3 (ord [2]))

= { 2° pott. di ord x=2, xs=[] }
insord 1 (insord 3 (insord 2 (ord [])))

= { 1° pott. di ord }
insord 1 (insord 3 (insord 2 []))

= { 1° pott. di insord }
insord 1 (insord 3 [2])

= { 2° pott. di insord }
insord 1 (2::(insord 3 []))

= insord 1 (2::3::[])

= 1::2::3::[]

= [1;2;3]

Date due liste ordinate comporle in una sola lista ordinata

merge : 'a list → 'a list → 'a list

Ip: merge l1 l2 , l1 è ordinata
l2 è ordinata

Voglio che il risultato sia ordinato

merge [1; 10] [2; 3; 20] = [1; 2; 3; 10; 20]

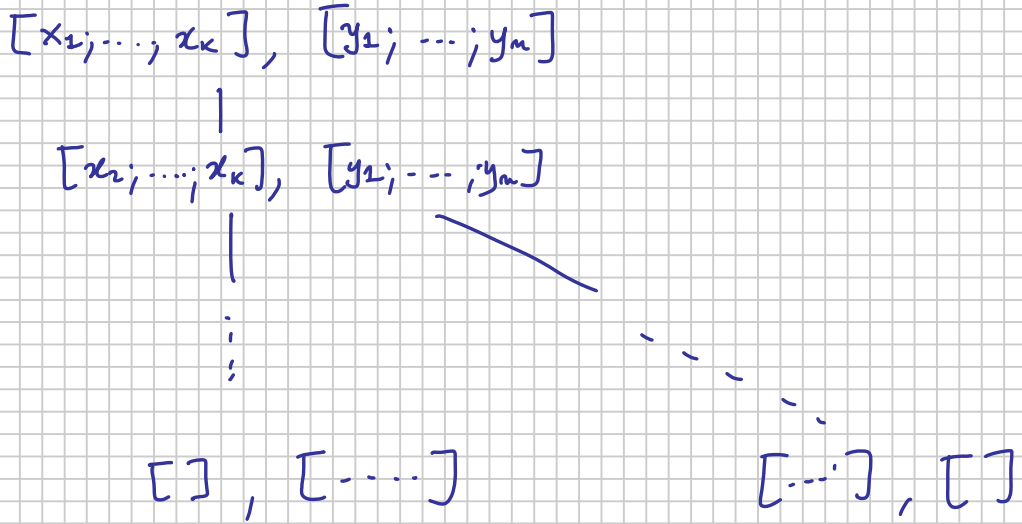
```
let merge l1 l2 = ord (l1@l2);;
```

```
let rec merge l1 l2 = if l1 = [] then l2
                      else if l2 = [] then l1
                      else if (hd l1) < (hd l2) then (hd l1) :: (merge (tl l1) l2)
                                                else (hd l2) :: (merge l1 (tl l2));;
```

```
let rec merge l1 l2 = match (l1, l2) with
    [], _ → l2 | _, [] → l1
  | x::xs, y::ys when x<y → x :: (merge xs y::ys)
  | x::xs, y::ys when y<=x → y :: (merge x::xs ys);;
```

$$(l, l') \sqsubset (l_1, l_1') \equiv$$
$$(l = tl\ l1 \wedge l' = l_1') \vee$$
$$(l' = tl\ l_1' \wedge l = l_1)$$

quasi....


[x1; ...; xk], [y1; ...; ym]
[x2; ...; xk], [y1; ...; ym]
[], [...]
[...], []

Spezzare in due una lista

split l = l1, l2

l1 contiene tutti gli elementi positivi di l

l2 " " " negativi o nulli di l

split [1; 0; -3; 5; 2] =
[1; 5; 2], [0; -3]

```
let rec split l = match l with
    [] -> [], []                          <-- 'a list * 'a list
  | x::xs when x>0 -> x::(split xs), split l
```

'a list * 'a list

'a list * ('a list * 'a list) NO

```
let rec split l = match l with
    []      ->  [], []
  | x::xs when x>0 -> let l1,l2 = split xs
                      in x::l1, l2
  | x::xs when x<=0 -> let l1,l2 = split xs
                       in l1, x::l2 ;;
```

```
x::xs -> let l1,l2 = split xs in
         if x>0 then x::l1,l2
                else l1, x::l2
```

```
# let (a,b) = (3,4) in
  a+b ;;
- : int = 7
```

```
let split l =
    let rec splitacc l (l1,l2) = match l with
                []   ->  l1,l2
      |x::xs -> if x>0 then splitacc xs (x::l1,l2)
                       else splitacc xs (l1,x::l2)
    in
        splitacc l [],[] ;;
```

l1 @ [x]

l2 @ [x]

---

splitacc [1;-1;3;ø] [],[]

= splitacc [-1;3;ø] [1],[]

= splitacc [3;0] [1],[-1]

= splitacc [ø] [3;1],[-1]

= splitacc [] [3;1],[ø;-1]

= [3;1],[ø,-1]

[1;3],[-1;ø]

Data una lista, vogliamo lasciare 1 sola occorrenza di ciascun elemento
se un elemento compare più volte nella lista, nel risultato deve comparire una volta sola

concdup [3;2;1;2;3;5] = [3;1;2;5]

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

```
(*) let rec member x l = match l with
        []  ->  false
    | y::ys when x=y  ->  true
    | y::ys when x<>y ->  member x ys
    in
```

concdup [3;2;1;2;3;5]
=
concdup [2;1;2;3;5]
=
concdup [1;2;3;5]
=
1::(concdup [2;3;5])
=
1::2::(concdup [3;5])
=
⋮
[1;2;3;5]

Cancellare da una lista gli elementi CONTIGUI ripetuti.

conc [2; 3; 2; 2; 5; 1; 1; 1; 5; 5] = [2; 3; 2; 5; 1; 5]

```
let rec conc l = match l with
      []  ->  []
    | [x] ->  [x]
    | x::y::zs when x=y  -> conc (y::zs)
    | x::y::zs when x<>y -> x::(conc (y::zs));;
```

---

① minocc l = (m, n)

m è il minimo valore in l
n è il numero di occorrenze di m in l

② primapos l = l'

l' contiene tutti i valori di l, ma tutti i valori positivi devono comparire in l' prima dei valori negativi o nulli

primapos [3; 0; -1; 5] = [3; 5; 0; -1]