Esercizi: funzioni CAML su liste

Titolo nota 24/09/2015

- data una lista di interi, costruire due liste, una con i valori negativi e una con quelli positivi o nulli

split [-2; 3; 4; 5; -7; -10] = ([-2; -7; -10], [3; 4; 5])

l'ordine non ha importanza

split : int list → int list * int list

suggerimento

let (n1, n) = ... in ...

```
#let rec split l = match l with
      [] -> ([],[])
    | x::xs -> let (l1,l2) = split xs
               in if x<0 then (x::l1, l2)
                         else (l1, x::l2);;
split : int list -> int list * int list = <fun>
```

date una lista cancellare gli elementi ripetuti contigui

Titolo nota 24/09/2015

conc [1;1;2;1;3;3;3;7] = [1;2;1;3;7]

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

```
x::y::ys when x=y
  -> conc (y::ys)

x::y::ys when x<>y
  -> x:: conc (y::ys)
```

conc : 'a list -> 'a list = <fun>

Titolo nota 24/09/2015

Cancellare (lasciandone solo uno) tutti gli elementi ripetuti in una lista

conc [1;2;1;3;4;2;2] = [1;2;3;4] oppure [1;3;4;2]

member

member : 'a → 'a list → bool

member 3 [1;2;3] = true

member 4 [1;2;4;4] = true

member 2 [1;3;4;4] = false

Titolo nota 24/09/2015

```
let rec member a l = match l with
        []  -> false
      | x::xs when x = a -> true
      | x::xs when x <> a -> member a xs ;;
member : 'a -> 'a list -> bool = <fun>
```

```
let rec canc l = match l with
        []  -> []
      | x::xs when member x xs -> canc xs
      | x::xs when not(member x xs) -> x :: canc xs ;;
canc : 'a list -> 'a list = <fun>
```

```
x::xs ->
if x = a then true
   else member a xs ;;
```

canc [1;2;3;1] = [2;3;1]

Titolo nota 24/09/2015

data una lista di interi costruire una lista con gli stessi elementi in cui i valori negativi precedono quelli positivi o nulli

ord [-2; 3; 4; -5; 7; 0] = [-2; -5; 3; 4; 7; 0] l'ordine non è importante

```
let rec ord l = match l with
    [] -> []
  | x::xs when x<0 -> x :: ord xs
  | x::xs when x>=0 -> (ord xs)@[x] ;;
```

(x::[]) ≡ [x]

(ord xs)@[x] ⟵ (ord xs)@(x::[])

ord : int list -> int list = <fun>

ord [1; 2; -3]
= {def ord, 3° p}
(ord [2; -3])@[1]
= {def. ord 2° p}
((ord [-3])@[2])@[1]
= {def ord, 3° p}
(((ord [])@[-3])@[2])@[1]
= -3::(((ord [])@[2])@[1])
= [-3; 2; 1]

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

ins 3 [-2;-3;4]

=

-2 :: (ins 3 [-3;4])

=

-2 :: -3 :: (ins 3 [4])

=

-2 :: -3 :: [3;4]

= [-2;-3;3;4]

```
let rec ord l = match l with
        [] -> []
      | x::xs when x<0 -> x::(ord xs)
      | x::xs when x>=0 -> ins x (ord xs) ;;
```

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

Titolo nota 24/09/2015

inserire un elemento in una lista ordinata (non decrescente) in modo che il risultato sia ancora una lista ordinata.

ins 5 [-2; 3; ø; ø; 7] = [-2; -3; ø; ø; 5; 7]

```
let rec ins n l = match l with
        [] -> [n]
      | x::xs when n <= x -> n :: x :: xs
      | x::xs when n > x -> x :: (ins n xs) ;;
```

ins 3 [-2; -1; 5; 7] = -2 :: -1 :: [3; 5; 7]

= -2 :: (ins 3 [-1; 5; 7]) = [-2; -1; 3; 5; 7]

= -2 :: -1 :: (ins 3 [5; 7])

Titolo nota 24/09/2015

Sfruttando ins (inserisce in modo ordinato)
definire una funzione che ordina una lista (non decrescente)

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

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

Titolo nota 24/09/2015

merge prende due liste ordinate e restituisce una lista ordinata che contiene tutti gli elementi delle due

merge [-3; 2; 5] [1; 7] = [-3; 1; 2; 5; 7]

```
let rec merge l1 l2 = match (l1, l2) with
    ([], l) -> l
  | (x::xs, []) -> l1
  | (x::xs, y::ys) -> if x<y then x :: (merge xs (y::ys))
                      else y :: (merge (x::xs) ys);;
```