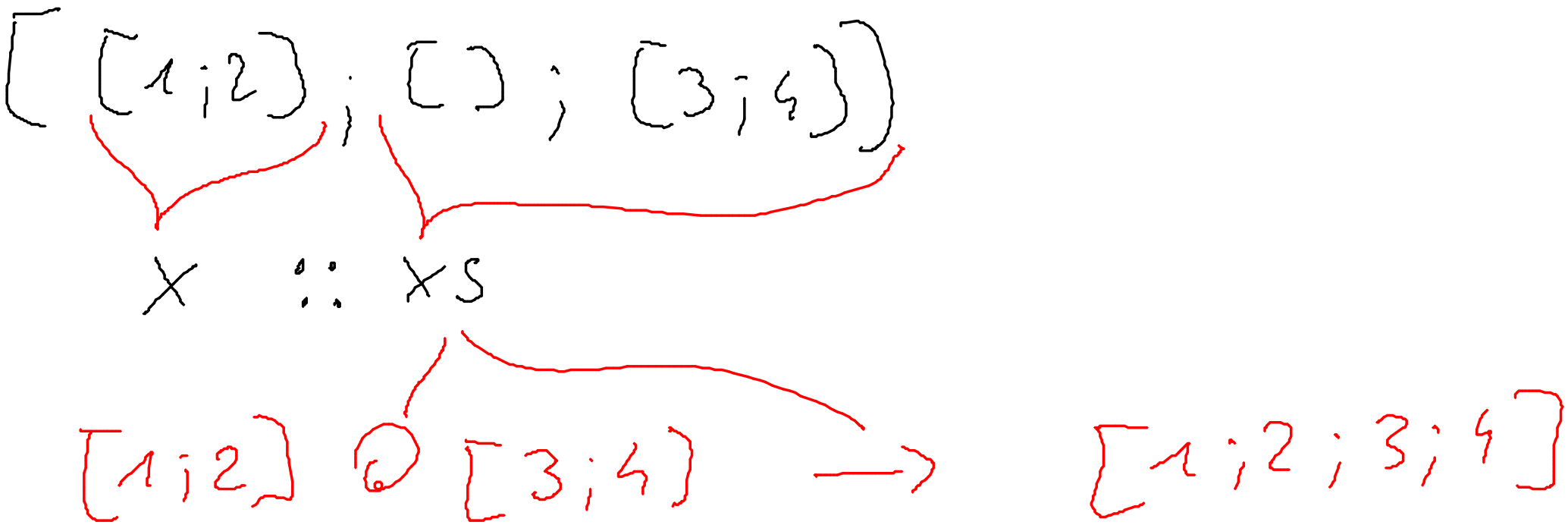
flatten : 'a list list → 'a list

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

- ricorsivamente
- folder


[[1;2] ; [] ; [3;4]]
x :: xs
[1;2] ⊙ [3;4] → [1;2;3;4]

```
let rec flatten l = match l with
    [] -> []
  | x::xs -> x @ flatten xs ;;
flatten : 'a list list -> 'a list = <fun>
```

```
let flatten l =
  let f x y = x @ y
  in foldr f [] l ;;
flatten : 'a list list -> 'a list
```

split : 'a list → 'a → 'a list * 'a list

split [3; 4; 25; 2; -3; 30) 10 = ([3;4;2;-3] , [25; 30)

≤ 10 > 10

nic.

Polish

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

```
let rec split l el =
    let rec f cmp ls el = match ls with
                          [] -> []
                        | x::xs when cmp x el ->
                                  x :: f cmp xs el
                        | x::xs when not cmp x el ->
                                  f cmp xs el
in (f (pref x >=) l el, f (pref x <) l el);;
```

```
let split l x =
    let f el (l1, l2) = if el <= x then (el :: l1, l2)
                        else (l1, el :: l2)
    in fold_right f (□,□) l ;;
```

,,

```
f x y = let (l1, l2) = y
        in

f x y = match y with
        (l1, l2) ->
```

automap : ('a → 'b) * 'a list → 'b list

ogni elemento della lista risultato è ottenuto applicando il primo elemento della coppia al secondo.

```
ma1   ma2        let ma2 x = x+2;
automap [(ma1, 5); (ma2, 5); (ma1, 6)]
        = [6; 7; 7]
```

Nic folder

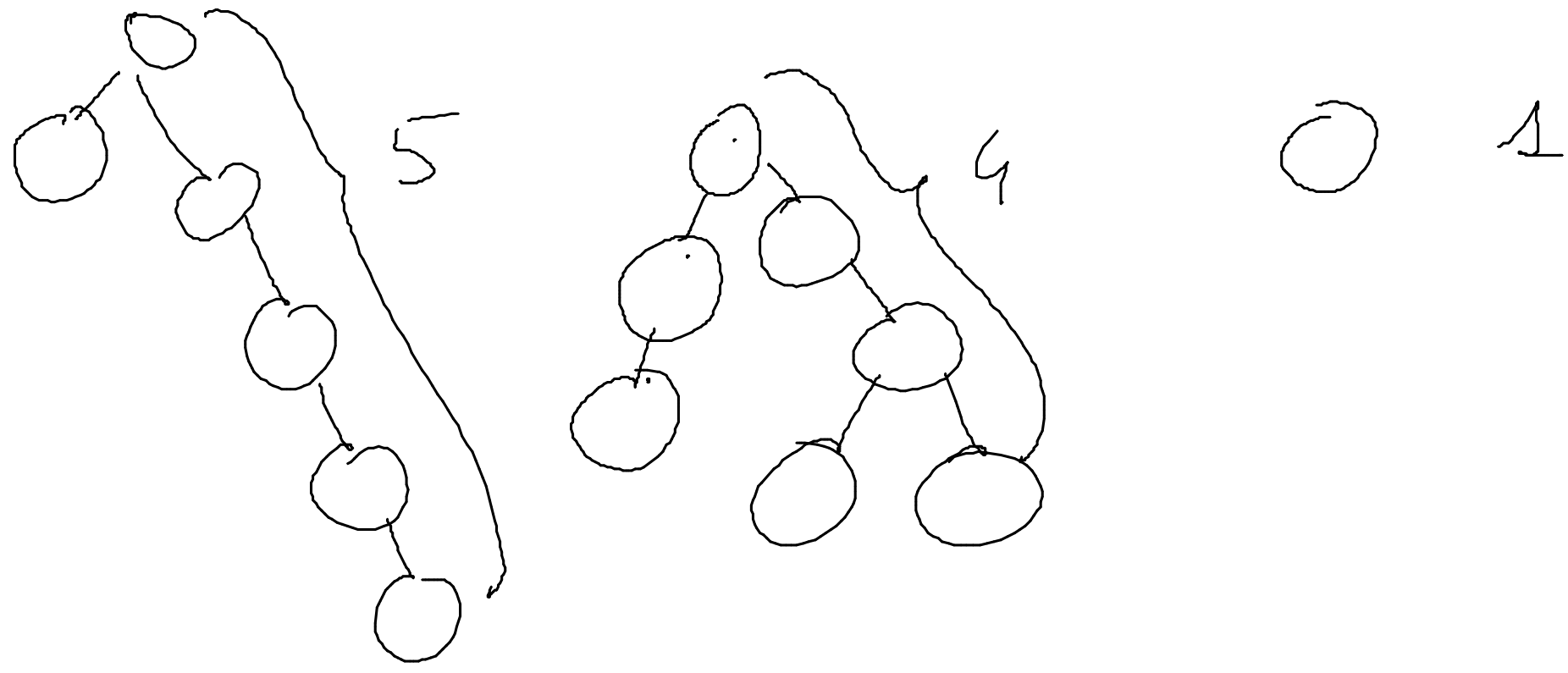
```
let rec automap l = match l with
    [] -> []
  | (f,x)::xs      f x :: automap xs;;
```

```
let automap l =
    let f (g,x) y = g x :: y
    in fold f [] l;;
```

x::ys -> let (g,w) = x in (g w)::automap ys

type 'a btree = Void | Node of 'a * 'a btree * 'a btree ;;

let rec depth bt =

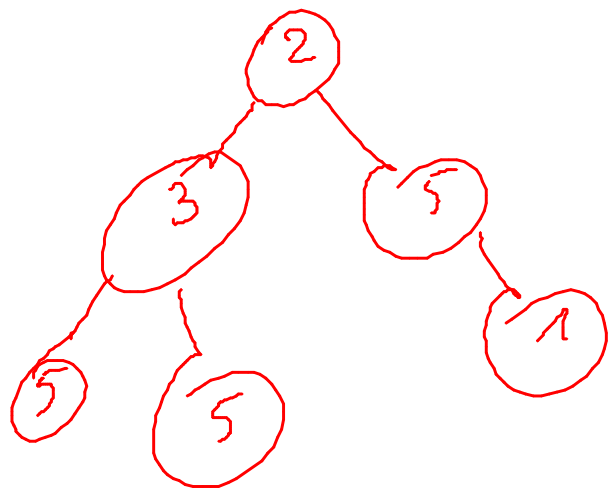


```
let rec depth bt = match bt with
    Void -> 0
  | Node(x, lbt, rbt) -> let p1 = depth lbt
                         and p2 = depth rbt
                         in if (p1 < p2) then p2 + 1
                            else p1 + 1 ;;
```

```
Node (x, lbt, rbt) -> if (depth lbt) < (depth rbt)
                      then (depth rbt) +1
                      else (depth lbt) +1
```

let rec conta bt x = numero di occorrenze di x in bt

conta



5 = 3

20 = ∅

1 = 1

```
let rec conte bt x = match bt with
    Void -> 0
  | Node (y, lbt, rbt) when x=y -> 1+(conte lbt x)+
                                   (conte rbt x)
  | Node (y, lbt, rbt) when x<>y -> (conte lbt x)+
                                   (conte rbt x) ;;
```

let rec ...

```
| Node (y, lbt, rbt) -> (conte lbt x) +
                        (conte rbt x) +
(if x = y then 1 else 0)
```