[] lista vuota

:: cons permette di aggiungere un elemento in testa a una lista

3 :: [] = [3]

4 :: [3] = [4;3] ≠ [3;4]

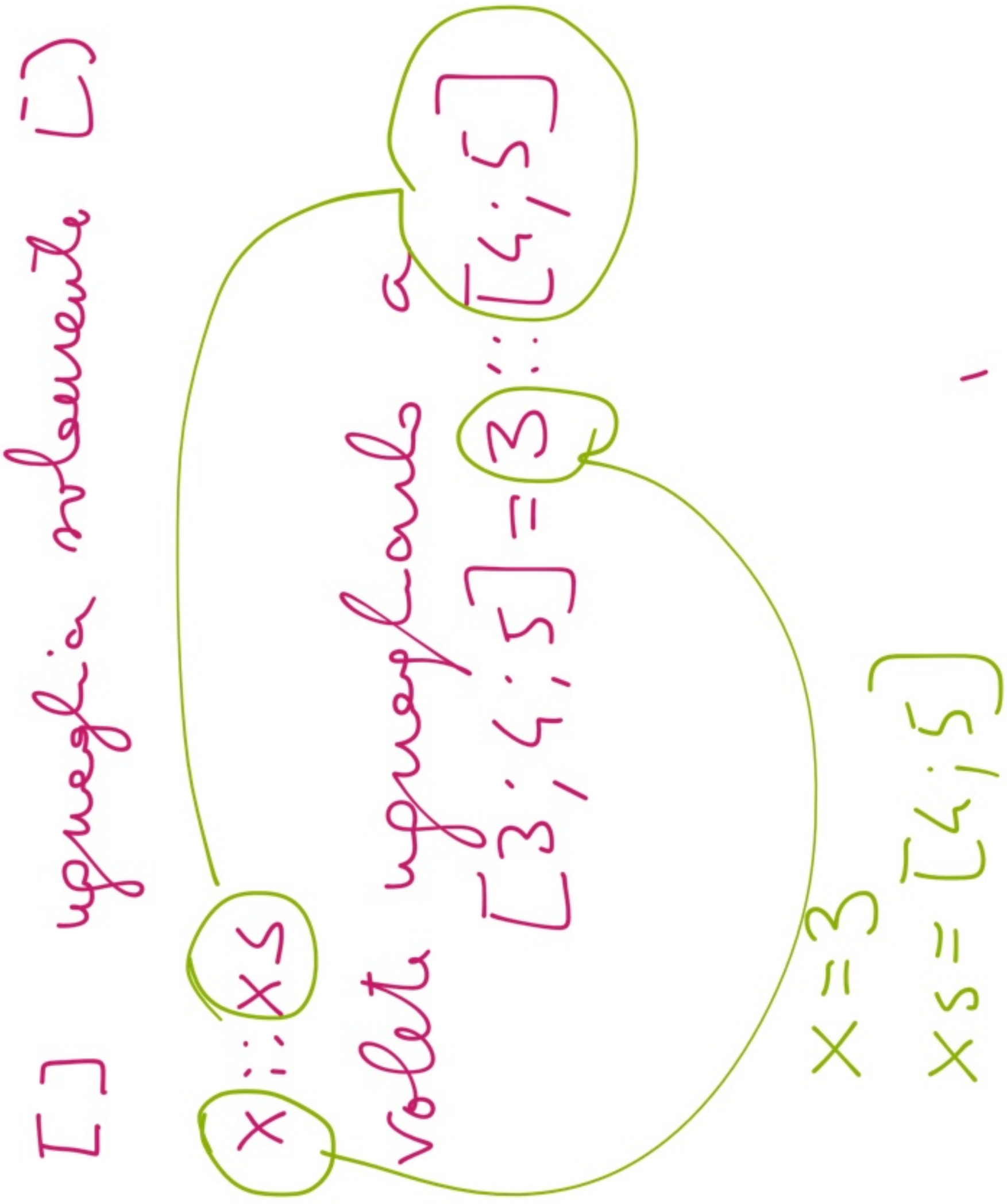
pattern per liste

[] ← uguaglia lista vuota

x :: xs

variabili (nomi che possono essere istanziati a valori)

uguagliare liste non vuote



$x :: (y :: ys)$ ← più impegnativo

- $[]$ no
- $[3] = 3 :: []$ no

$x = 3$
$y = ?$ ←
$ys = []$

solamente a liste che hanno almeno 2 elementi

$[3;4] = 3 :: 4 :: []$

$x = 3$
$y = 4$
$ys = []$

Simple che è vero
lista che hanno
elementi

simple [3;2] =
simple [ ] =
simple [3] =

```
let simple l = match l with
  [] -> false
  x::[] -> no
  [3;4] -> [3;4]
```

el lista

lista

[3;4] :: [] =

[[3;4]]

[3;4]

↳ x = [3;4]

[] = []

```
let simple l = match l with
    [] -> false
  | x::[] -> true
  | x::y::ys -> false ;;
  | x::xs when xs<>[] ->
```

simple : 'a list → bool = <fun>

l ns

Uniamo le funz. predefinite hd tl

```
hd : 'a list -> 'a
tl : 'a list -> 'a list
```

```
let rec simple l =
    if l = [] then false
    else if tl l = [] then true
    else false ;;
```

simple : 'a list -> bool = <fun>

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

```
len [3;4] = 2
len [] = 0
```

len [3;4] 2° caso

= { def len. l = [3;4], l deve unificare
x :: xs, x = 3 ; xs = [4]

1 + len [4]

= { def len, 2° caso, l = [4], l = 4 :: [], l deve unif. x :: xs, x = 4, xs = [] }

1 + 1 + len []

= { def f, 1° caso, l = [] }

1 + 1 + ∅ = { calcolo } 2

Dominio 'a list

$$\left(\forall l, l' \in \text{'a list}.\ l \prec l' \equiv \left(\exists x \in \text{'a}.\ \exists\, l'' \in \text{'a list}.\ l' = x :: l'' \wedge l = l''\right)\right)$$

$$l' = x :: l''$$

$$l = l''$$

tl hd definita 'a lista

$(\forall l, l' \in$ 'a lista$)$

$l \leq l' \equiv$

$l =$ testa'

Concatenazione tra liste

@ predefinita

[3;4] @ [4;5] =

[3;4;4;5]

prefix @ ;;

- : 'a list -> 'a list -> 'a list

=(fun)

- $[] \oplus l = l$
  $l = l \oplus []$

$[3;4] \oplus [] = [3;4]$

$x :: (l \oplus l') = (x :: l) \oplus l'$

proprietà di $\oplus$

append

```
let rec append l l1 =
  match l with
  [] -> l1
| x::xs -> x :: (append xs l1) ;;
```

corrected

```
let rec append l l1 =
  match l with
  [] -> l1
| x::xs -> (xs, l1) < (x::xs, l1)
```

```
append : 'a list -> 'a list -> 'a list
```

@ [3;4) [5;5]

= { def a, l = [3;4]; l1 = [5;5];
x = 3; xs = [4]; caso 2 }

3 :: (a [4] [5;5])

= { def a, l = [4], l1 [5;5]; x = 4; xs = [];
caso 2° }

3 :: (4 :: (a [] [5;5]))

= { def a, l = [], l1 = [5;5], 1° caso }

3 :: (4 :: [5;5])

= { calcolo }
[3;4;4;5] .

dimostra
$(\forall l, l' \in$ a list.
append $(l @ l')$

corrupted

- $[] @ e = e$
- $x :: (l @ l') = (x :: l) @ l'$

Dominio 'a list * 'a list
di append

Relazione di precedenza mediante
dalla definizione di append.

$(\forall l, l', l'', l''' \in$ 'a list. $\langle l, l' \rangle \prec \langle l'', l''' \rangle \equiv l' = l''' \wedge l = tl\ l'')$

$\langle l, l' \rangle$

$\langle tl\ l, l' \rangle$

$\langle tl(tl\ l), l' \rangle$

$\langle [\,], l' \rangle$

Caso base

append [] l = l

Caso induttivo

append xs l = ...

append (x::xs) l = x :: append xs l

Caso base

append [] l =

= { def. append [] l }

l

= { mon. @ , [] l = l }

[] @ l

append (x::xs) l =

= { def. append (x::xs) l }

x :: append xs l

= { 1° passo }

x :: (xs @ l)

C.I. append xs l = xs ⊙ l ind. ipot.

⟹ append (x :: xs) l = (x :: xs) ⊙ l

append (x :: xs) l

= { def. append, 2° patt. }

x :: (append xs l)

= { ip. ind }

x :: (xs ⊙ l)

= { prop. ⊙, x :: (l ⊙ l') = (x :: l) ⊙ l' }

(x :: xs) ⊙ l .

Cerca l'n-simo elemento di una lista

```
nth 2 [3;4;5;6] = 4

nth 2 [1] = undefined

nth (-3) [1;2] = useless
```

```
let rec nth n l =
  match (n, l) with
  | (1, x :: xs) -> x
  | (n, x :: xs) when n > 1 -> nth (n-1) xs
  | _ -> failwith "nth: out of bounds"
```

```
let secondo = nth 2 ;;
secondo : 'a list -> 'a = <fun>
```

```
let secondo x = nth 2 x ;;
secondo : 'a list -> 'a = <fun>
```

```
secondo [2;3;4];;
- : int = 3
```

take 2 [3;4;5;6] = [3;4]

take 3 [3;4] = [3;4]

take ∅ [3;4] = []

take (-3) [3;4] = undef.

```
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) ;;
take : int -> 'a list -> 'a list
=<fun>
```

take 2 [3;4;5];;

= {def take, 3° p}

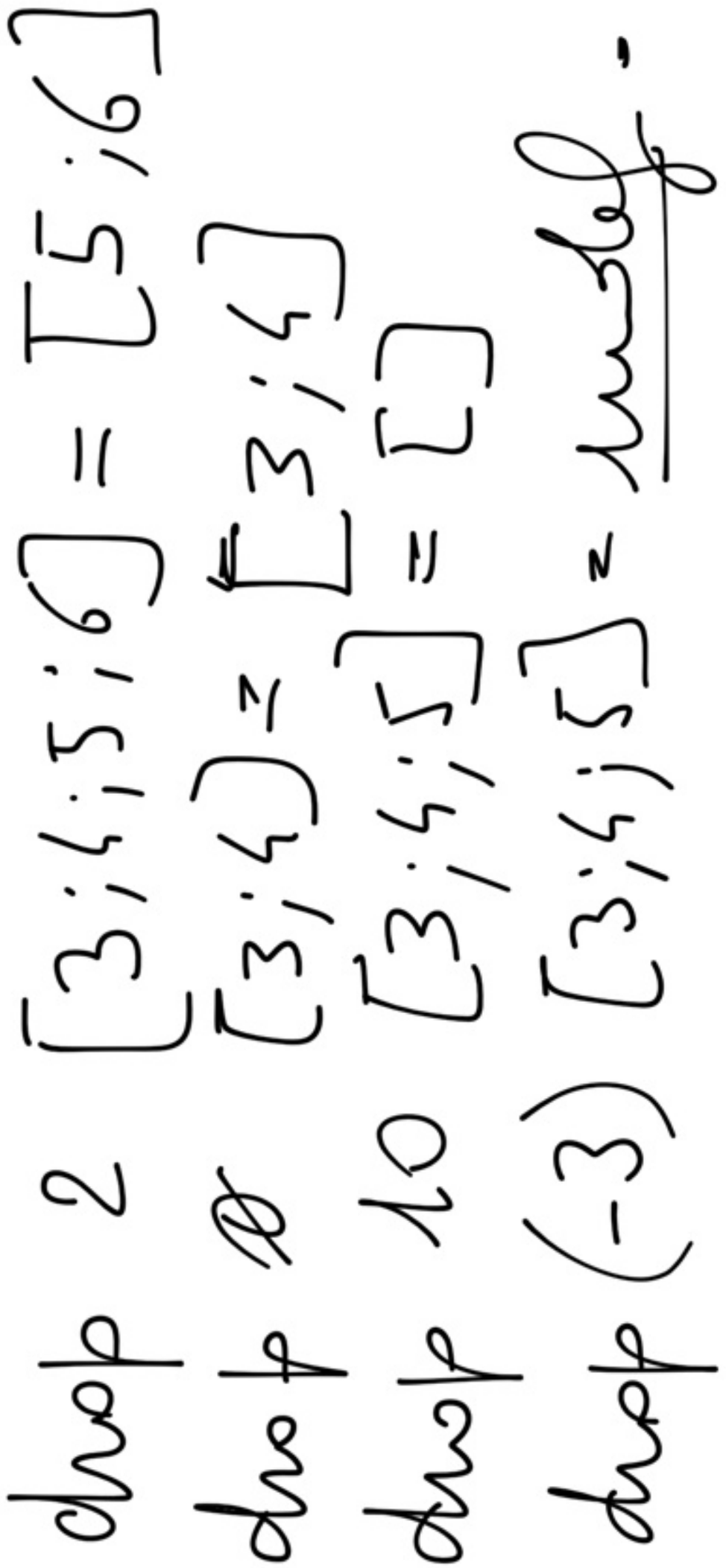
3::(take 1 [4;5])

= {def take, 3° p}

3::(4::(take ∅ [5]))

= {def take, 1° p}

3::(4::[]) = [3;4]



```
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>
```

$(n, [\,])$ when $n > 0 \rightarrow$

(n, l) when $n > 0$ & $l \neq [\,]$

$\rightarrow$ drop $(n-1)$ (tl l)