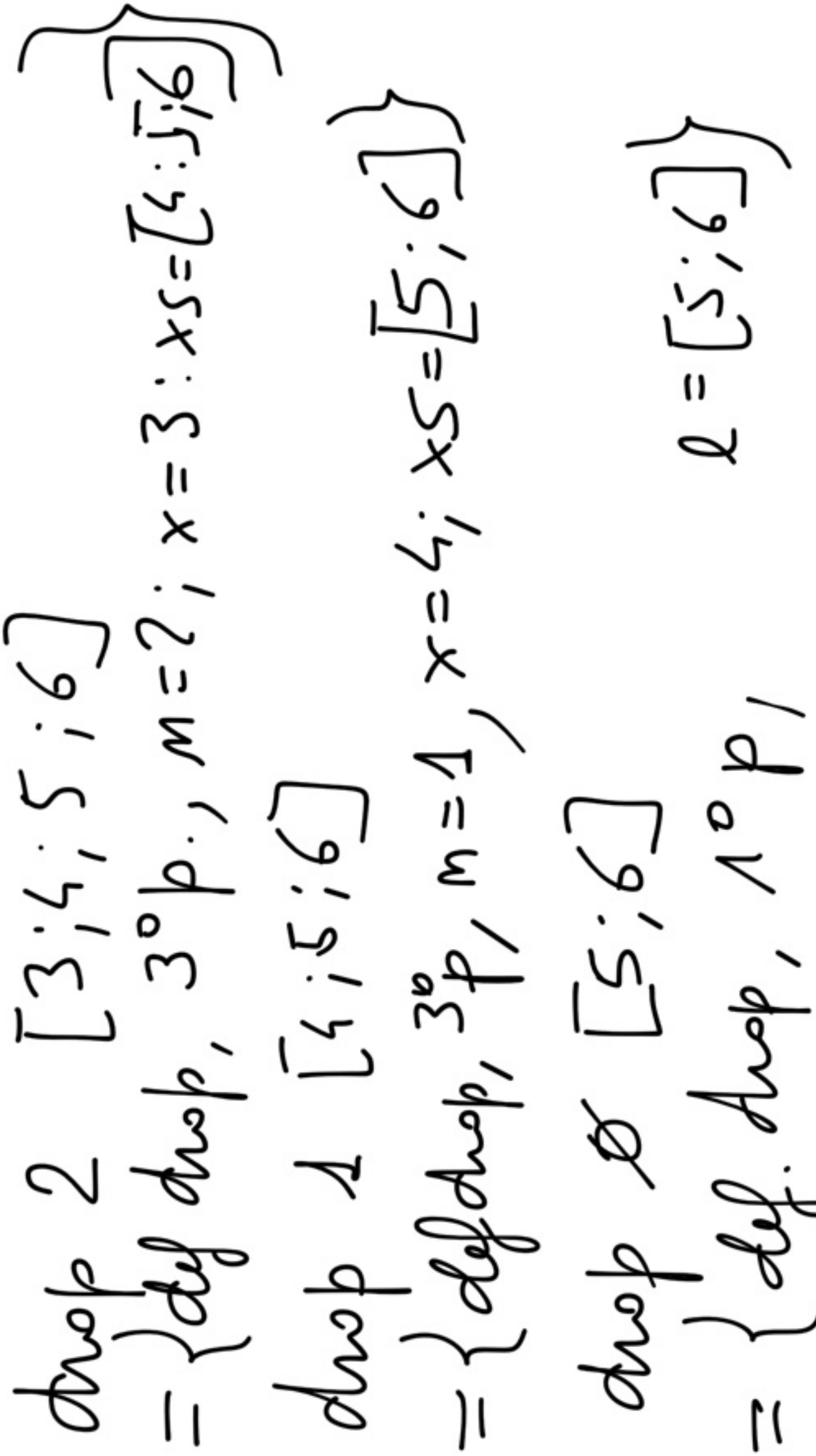
```
let rec take n l =
        match (n, l) with
  (0, l) -> []
| (n, []) when n>0 -> []
| (n, x::xs) when n>0 ->
        x :: take (n-1) xs ;;

domini  ℕ × 'a list
```

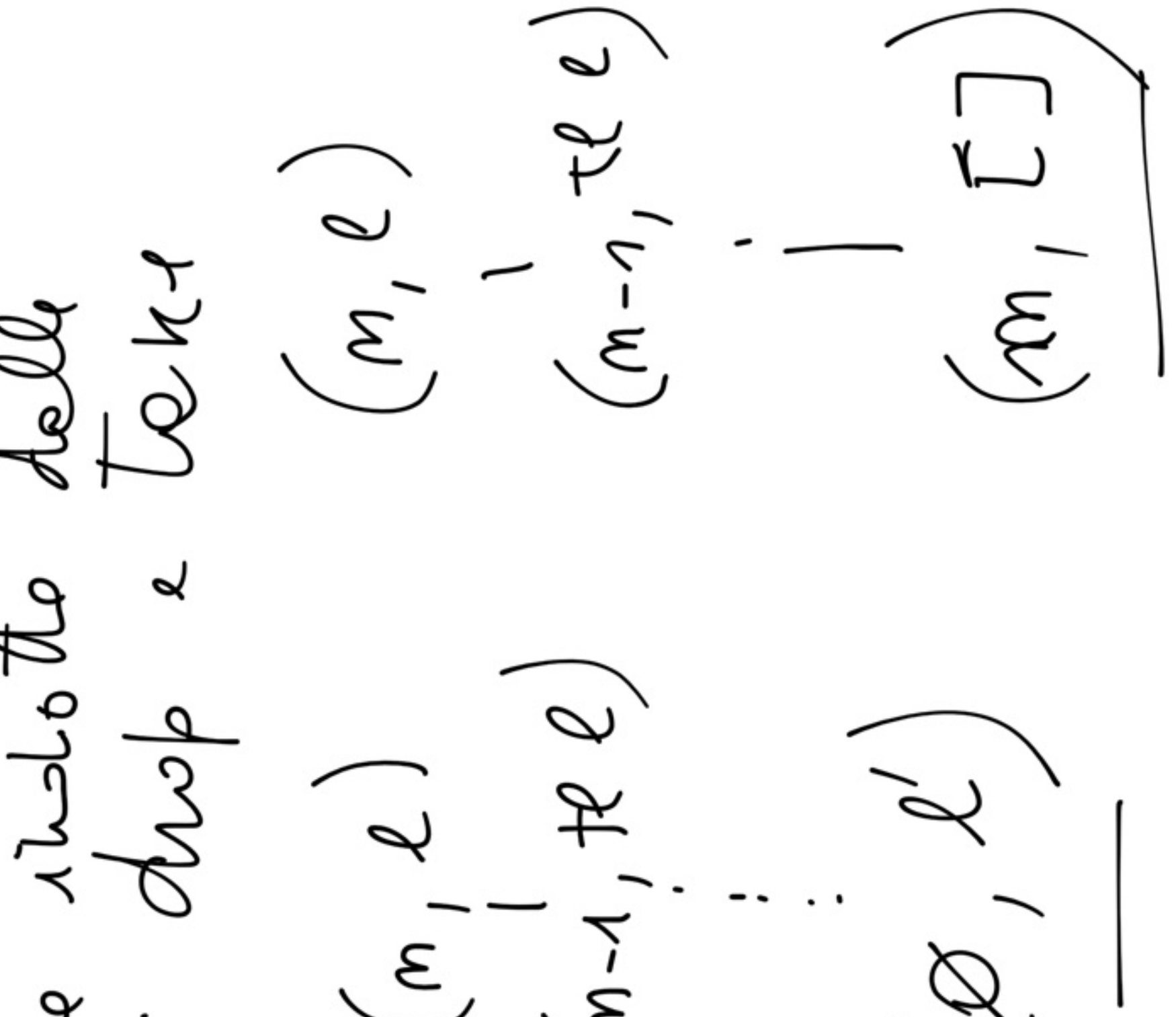
```
let rec drop n l =
        match (n, l) with
  (0,l) -> l
| (n, []) when n>0 -> []
| (n, x::xs) when n>0 ->
  drop (n-1) xs ;;
```

```
drop : int -> 'a list -> 'a list
```


dnop 2 [3;4;5;6]
= { def dnop, 3° p., n=2; x=3: xs=[4;5;6] }
dnop 1 [4;5;6]
= { def dnop, 3° p, n=1; x=4; xs=[5;6] }
dnop ∅ [5;6]
= { def. dnop, 1° p
[5;6]
q = [5;6] }
p /

Procedura ricorsiva della tabella di drop

def.



$$\Big(\forall n, n' \in \mathbb{N},\ l, l' \in \text{'a list}.$$

$$\langle n, l\rangle \prec \langle n', l'\rangle \equiv$$

$$n = n' - 1 \land$$

$$l = \text{tl } l' \land n' > 0 \land$$

$$l' \neq [\,]\Big)$$

si possono omettere

$\forall n \in \mathbb{N}, l \in$ 'a list'.
(take n l) (drop n l)
(take n l) @
= l

Caso base

$$(\text{take}\ 0\ l) \mathbin{@} (\text{drop}\ 0\ l) = l$$

$$(\text{take}\ n\ []) \mathbin{@} (\text{drop}\ n\ []) = []$$

Caso induttivo

$$(\text{take}\ n\ xs) \mathbin{@} (\text{drop}\ n\ xs) = xs$$

$\Longrightarrow$ ip. ind.

$$(\text{take}\ (n+1)\ (x :: xs)) \mathbin{@} \text{drop}\ (n+1)\ (x :: xs) = x :: xs$$

$$
\begin{aligned}
\text{ben ban } & 1 \\
(\text{take } \varnothing\ l) & \otimes (\text{dwp} \otimes l) \\
(\text{take } \varnothing\ l) & \otimes (\text{dwp} \otimes l) \\
&= \{\text{def. take, dwp, } \wedge \circ p\} \\
& \exists \otimes l \\
&= \{\text{mph.} \otimes, \\
& l
\end{aligned}
\qquad
\left\{ e = e \right\}
$$

Caso base 2

$$(\text{take } n\ [\,])\ @\ (\text{drop } n\ [\,]) = [\,] \quad n > 0$$

$$(\text{take } n\ [\,])\ @\ (\text{drop } n\ [\,])$$

$$= \{ \text{def. take, drop, } 2^{\circ}\ p. \}$$

$$[\,]\ @\ [\,]$$

$$= \{ \text{nop. } @ \}$$

$$[\,]$$

C. Ind.

(take n xs) ++ (drop n xs) = xs ip. ind.

⇒ (take (n+1) (x :: xs)) ++ (drop (n+1) (x :: xs))
= x :: xs

(take (n+1) (x :: xs)) ++ ((drop (n+1) (x :: xs))

= { def. take, drop, 3° p }

(x :: take n xs) ++ (drop n xs)

= { pop. ++, (x :: l) ++ l' = x :: (l ++ l') }

x :: ((take n xs) ++ (drop n xs))

= { ip. ind. }

x :: xs

Reverse

reverse [1;2;3] = [3;2;1]

reverse [] = []

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

reverse : 'a list → 'a list = <fun>

tipo l — tipo ris

```
let rec reverse l =
    if l = [] then []
    else reverse (tl l) @ [hd l];;
```

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

reverse [1;2;3]
= {def. rev., 2° p}
reverse [2;3] @ [1]
= {def. rev. 2° p.}
(reverse [3] @ [2]) @ [1]
= {def. reverse, 2° p}
((reverse [] @ [3]) @ [2]) @ [1]
= {def rev. 1° p.
(([] @ [3]) @ [2]) @ [1]

= {calcolo
[3;2;1]

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

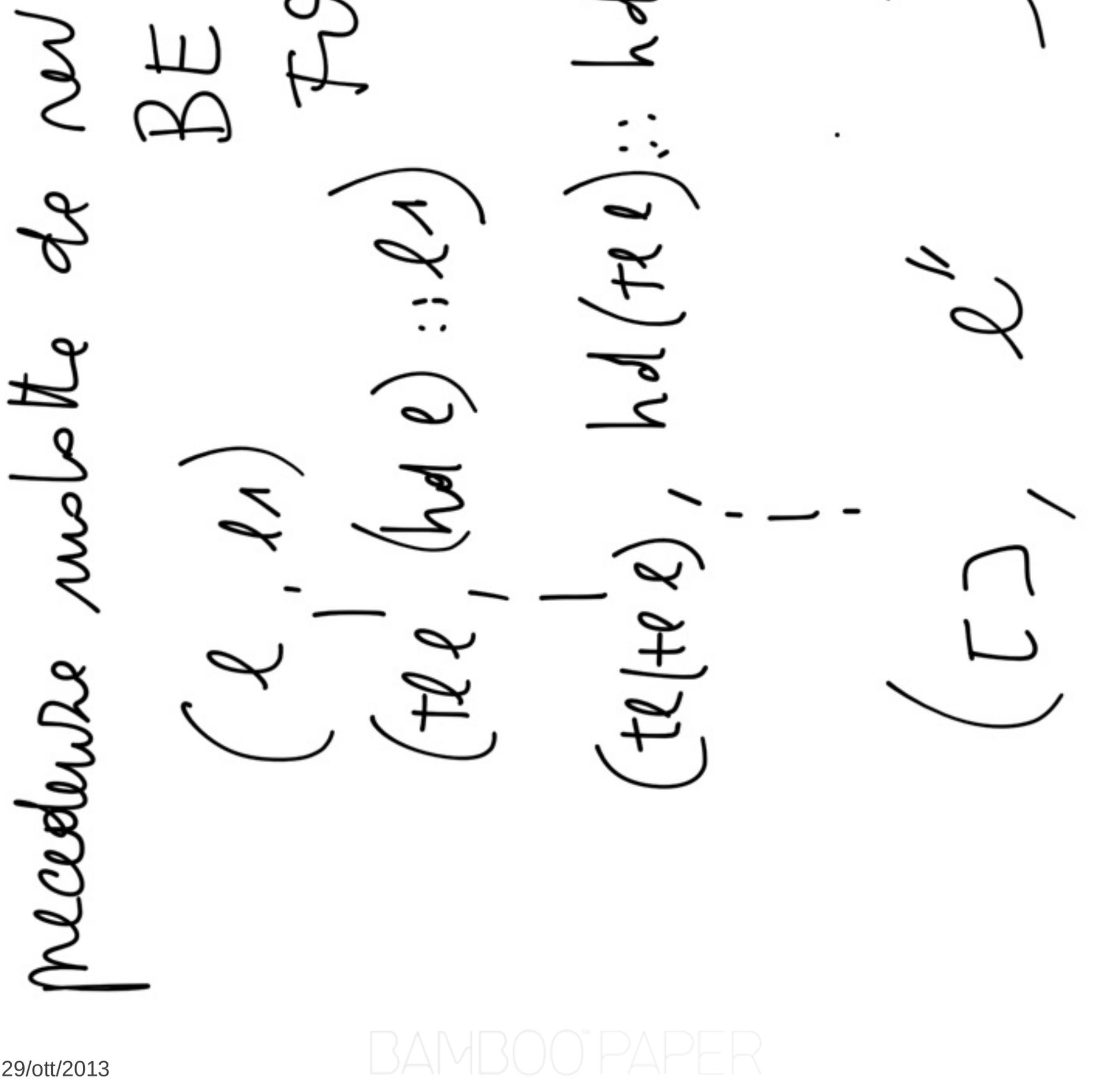
rev [1;2;3] []
= {def rev, 2° p}
rev [2;3] 1::[]
[1]
= {def rev, 2° p}
rev [3] [2;1]
= {def rev, 2° p}
rev [] [3;2;1]
= {def rev. 1° p}
[3;2;1]

accumulatore

rev [1;2;3] [4;5]

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

rev l l1 = (reverse l) @ l1


procedure molto
de
nel
BEN
FONDATA

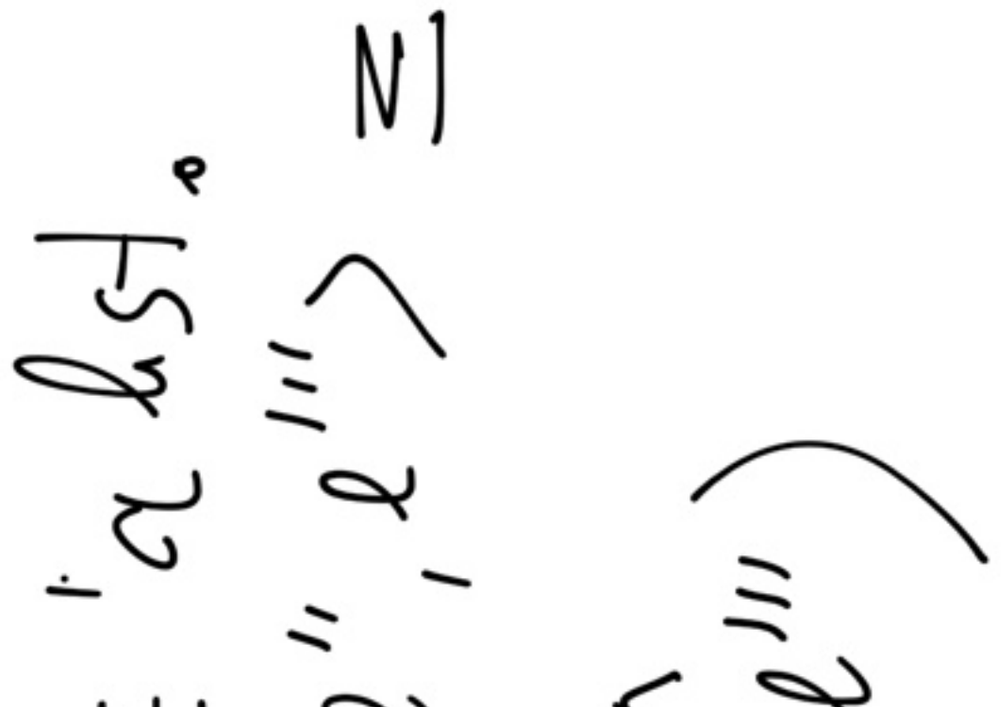
formalmente

$(\forall l, l', l'', l''' \in$

$\langle l, l' \rangle \prec \langle$

$l = \mathrm{Tr}\ l'' \wedge$

$l' = \mathrm{hd}\ l ::$



```
let rec nel l l1 = match l with
    [] -> l1
  | x::xs -> nel xs (x::l1);;
val nel : 'a list -> 'a list -> 'a list = <fun>
let rec rev l = nel l [];;
val rev : 'a list -> 'a list = <fun>
```

```
let rec rev l1 l = match l with
    [] -> l1
  | x::xs -> rev (x::l1) xs ;;
rev : 'a list -> 'a list -> 'a list
```

accumulatore è il primo argomento

rev [] [1;2;3] = [3;2;1]

rev [4;5] [1;2;3] = [3;2;1;4;5]

```
let revs l = rev [] l ;;
revs : 'a list -> 'a list = <fun>
```

≡

```
let revs = rev [] ;;
revs : 'a list -> 'a list = <fun>
```

Funzioni di ordine superiore "classiche"

Quantificatori su liste.

$$(\forall x \in A .\ P(x))$$

$$(\exists x \in A .\ p(x))$$

29/ott/2013

p(x) → vero / falso

→ predicato: funzioni che hanno un risultato booleano

```
let gt0 x = x > 0;
gt0 : int -> bool = <fun>
```


2013_10_29

[-1 ; 3 ; 25 ; 10]

- qto non vale in tutti
  malsiote un elemento in
  cui qto vale

```
let rec forall p l =
  match l with
    [] -> true
  | x::xs -> p x && forall p xs;;
```

```
forall : ('a -> bool) -> 'a list -> bool = <fun>
```

p l ns

forall gto [1;2;3]

=

false

forall gto [true;true;true];;

errore di tipo

int -> bool

'a list

forall : ('a -> bool) -> 'a list -> bool

```
let rec exists p l =
    match l with
    [] -> false
  | x::xs -> p x || exists p xs ;;
```

```
exists : ('a -> bool) -> 'a list -> bool
```

```
let rec forall p l = match l with
  [] -> true
| x::xs when p x -> forall p xs
| x::xs when not(p x) -> false;;
forall : ('a -> bool) -> 'a list -> bool
```

```
let rec exists p l =
    match l with
    [] -> false
  | x::xs when p x -> true
  | x::xs when not (p x) ->
      exists p xs;;

exists : ('a -> bool) -> 'a list -> bool
= <fun>
```

filtro 2

filtro gto [-10 ; 3 ; -2 ; 0 ; 5 ; 1 ; 7]

= [3 ; 5 ; 1 ; 7]

```
let rec filter p l =
    match l with
    [] -> []
  | x::xs when p x ->
        x :: filter p xs
  | x::xs when not (p x) ->
        filter p xs ;;

filter : ('a -> bool) -> 'a list -> 'a list
           p              e          s
```