PATTERN

un pattern è una espressione con variabili (nomi a cui possono essere associati valori)

ES: `x::xs`

espressione di liste con le variabili x e xs

x può essere associata al primo elemento di una lista

xs alla lista senza il primo elemento

`x::xs` può essere uguagliato a:

`[3;4] = 3::[4]`

associando x al valore 3

e xs al valore [4]

NON può essere uguagliato alla lista []

NON può esser uguaglianza alla lista []

`x::[]` pattern

`[x]` pattern

entrambi possono essere ugualgliati a lista di un solo elemento

`[4]` anciando $x = 4$

↳ `4::[]`

`x::[]` oppure `[x]`

non può essere ugualgliato alla lista

`[3;4]` perché x dovrebbe essere

↳ `3::4::[]` anociato a `3::4`

non è un valore ma un errore di tipo

```
match e with
    pattern1 -> e1
| pattern2 -> e2
| pattern3 -> e3
|    :
```

Per eseguire l'espressione match:

si valuta e, si cerca di uguagliare il valore di con i pattern in sequenza.

Se patterni è il primo pattern che uguaglia il valore di e, allora il risultato è il valore dell'espressione ei

ES:

```
# match (3;4) with
    [] -> ∅
  | x::xs -> x ;;
- : int = 3
```

Lunghezza di una lista

```
# let rec len l =
    if l = [] then ∅
              else 1 + len (tl l) ;;
len : 'a list -> int = <fun>
```

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

ultimo elemento di una lista

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

equivalente `x::[]`

non mi piace

Per chiarezza vogliamo avere i pattern MUTUAMENTE ESCLUSIVI

Se sono uguagliati un valore a un pattern (nelle sequenze) non voglio che possa essere uguagliato ad altri.

`[x]` e `x::xs` non sono mutuamente esclusivi

`[3]` può essere uguagliata a entrambi!!

lunedì 26 novembre 2018 09:40

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

pattern mutuamente esclusivi.

In CAML è predefinito l'operatore di concatenazione tra liste

@

```
# [3;4] @ [5;6];;
- : int list = [3;4;5;6]
```

@ è un operatore infisso

```
# (@);;
```
errore di sintassi

prefix : una funzione predefinita che fa diventare un "operatore infisso" in "operatore prefisso"

```
# prefix @ ;;
- : 'a list -> 'a list -> 'a list = <fun>
```

```
# (prefix @) [3;4] [5;6];;
- : int list = [3;4;5;6]
```

ⓐ concatenazione tra liste

proprietà:

1) $[] @ l = l @ [] = l$

2) $x :: (l_1 @ l_2) = (x :: l_1) @ l_2$

Definiamola ricorsivamente

```
# let rec append l1 l2 =
    match l1 with
    [] -> l2
  | x::xs -> x :: append xs l2 ;;
append : 'a list -> 'a list -> 'a list = <fun>
```

tipo l1 — tipo l2 — tipo xs

```
append [3;4] [5;6]
= { def append ; 2° p  x = 3  xs = [4] }
  3 :: (append [4] [5;6])
= { def append ; 2° p  x = 4  xs = [] }
  3 :: (4 :: (append [] [5;6]))
```

$3 :: (4 :: (\text{append}\ []\ [5;6]))$

$= \{$ def append, 1° p $\}$

$3 :: (4 :: [5;6])$

$= \{$ calcolo $\}$

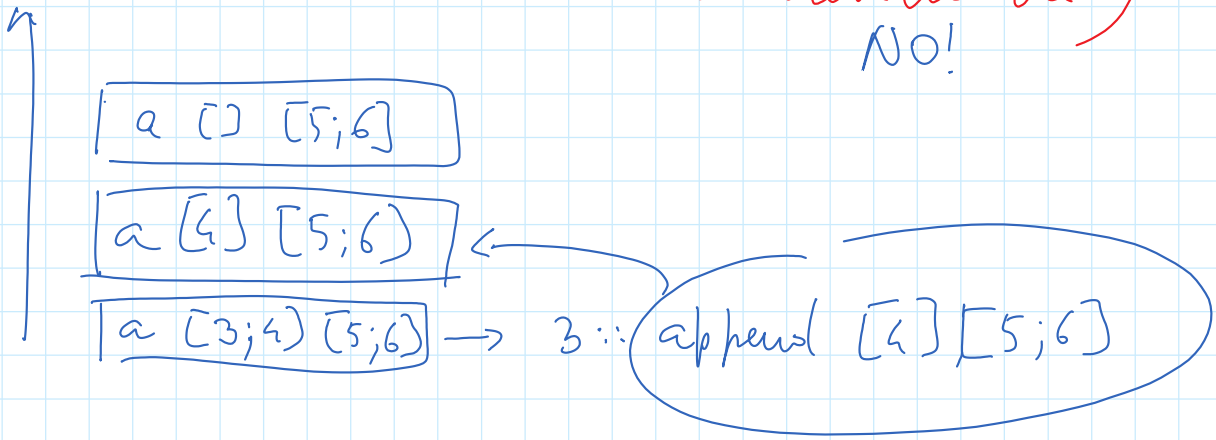
$[3;4;5;6]$

denotano la stessa lista

Ricorsione è realizzata (implementata) mediante uno STACK

(implementazione al posto di incrementare NO!)

STACK delle attivazioni



STACK OVERFLOW

append l_1 l_2 = $l_1 @ l_2$

da dimostrare con induzione BEN FONDATA

(una precedenza, $\sqsubset$, che non abbia catene infinite discendenti)

append l_1 l_2

dobbiamo fare induzione nel dominio di coppie di liste, cioè nel dominio CAML

'a list * 'a list

```
let rec append (l1, l2) =
    match l1 with
    [] -> l2
  | x::xs -> x :: append (xs, l2) ;;
```

(x::xs, l2) ⊐ (xs, l2)

append : 'a list * 'a list → 'a list = <fun>

$(\forall l_1, l_2 \in$ 'a list . append $(l_1, l_2) = l_1 @ l_2)$

Insieme 'a list * 'a list

Precedenza $\sqsubseteq$ $\in$ 'a list

$(\forall l_1, l_2, l_1', l_2'.\ (l_1, l_2) \sqsubseteq (l_1', l_2') \equiv$
$l_1 = tl\ l_1' \wedge l_2 = l_2')$ ← formalmente

$(xs, l_2) \sqsubset (x :: xs, l_2)$ ← intuitivamente

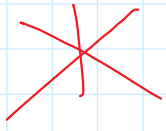
Minimali di $\sqsubseteq$ (infiniti)

$([], l_2)$ ←

$([3;4], [5;6])$

$([4], [5;6])$

$([], [5;6])$



lunedì 26 novembre 2018 10:30

$(\forall l_1, l_2 \in 'a\ list.$
$append(l_1, l_2) = l_1 @ l_2)$

1) $[] @ l = l @ [] = l$
2) $x :: (l_1 @ l_2) = (x :: l_1) @ l_2$

Caso base $append([], l_2) = [] @ l_2$

Caso induttivo

$append(xs, l_2) = xs @ l_2$
$\Rightarrow$
$append(x :: xs, l_2) = (x :: xs) @ l_2$

$(xs, l_2) \sqsubset (x :: xs, l_2)$

Dimostrazione

Caso base $append([], l_2) = [] @ l_2$

$append([], l_2)$
$= \{$ def. append, 1° p $\}$
l_2
$= \{$ neutro @, $[] @ l_2 = l_2 \}$

C) ω l2

lunedì 26 novembre 2018 10:36

Caso induttivo

$append(xs, l2) = xs \mathbin{@} l2$ ip. induttiva

$\Rightarrow$

$append(x :: xs, l2) = (x :: xs) \mathbin{@} l2$

$append(x :: xs, l2)$

$= \{$ def. append, 2° p $\}$

$x :: append(xs, l2)$

$= \{$ ip. induttiva $\}$

$x :: (xs \mathbin{@} l2)$

$= \{$ proprietà @ : $x :: (l1 \mathbin{@} l2) = (x :: l1) \mathbin{@} l2 \}$

$(x :: xs) \mathbin{@} l2$

Generalizzazione di hd

take n l - restituisca la lista composta dai primi n elementi di l (se ci sono, altrimenti tutti quanti possibile)

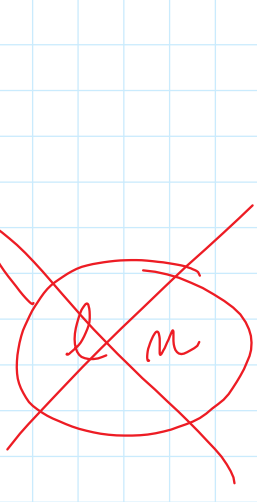
take 2 [3;5;7;8] = [3;5]
take 5 [//] = [3;5;7;8]
take 3 [] = []

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

take : int -> 'a list -> 'a list = <fun>

int: tipo di n; 'a list: tipo l; 'a list: tipo xs

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



Non sono mutuamente esclusivi

Clausole **when**

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

Viene preso se (l, n) uguaglia $(x::xs, n)$ e $n > 0$

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

Generalizzazione della tl

drop n l restituisce l senza i primi n elementi (se ci sono)

drop 3 [] = []
drop 3 [3;4;5;6] = [6]
drop 5 [//] = []

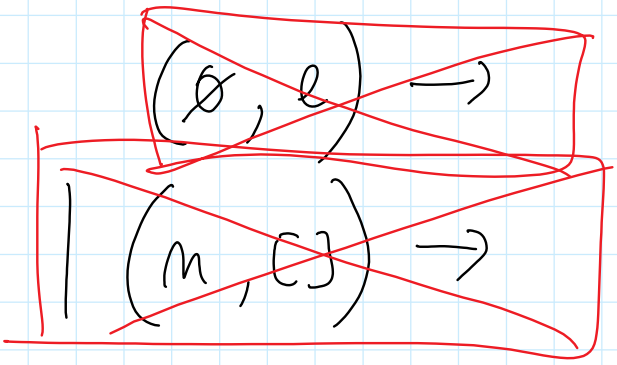
```
let rec drop n l = match (l,n) with
    ([],n) -> []
  | (x::xs,0) -> x::xs
  | (x::xs,n) when n>0 ->
                drop (n-1) xs;;
```

drop : int -> 'a list -> 'a list = <fun>

nth n l restituisce l'enesimo elemento di l

nth 3 [] indefinito
nth 3 [1;2] //
nth 3 [1;2;4;5] = 4

let rec nth n l = match (n, l) with



(0, l) ->
| (n, []) ->

(1, x::xs) -> x
| (n, x::xs) when n > 1 ->
nth (n-1) xs;;