LISTE IN CAML

Lista: sequenza di valori dello stesso tipo

Notazione: [3 ; 5 ; 10 ; 21 ; 5]

['a' ; 'c' ; 'd' ; 'b'] .

[1 ; 'a' ; 2] NO

```
# [1;2;3];;
- : int list = [1;2;3]
```

list ← costruttore di tipo (*, -> ....)

```
# [];;
```

LISTA VUOTA

```
- : 'a list = []
```

OPERATORE <u>CONS</u> (cons deriva dalla terminologia usata in <u>LISP</u> (<u>LIST</u> <u>P</u>rocessing))

```
:: : 'a * 'a list -> 'a list
```

```
# 3 :: [1;2];;
```

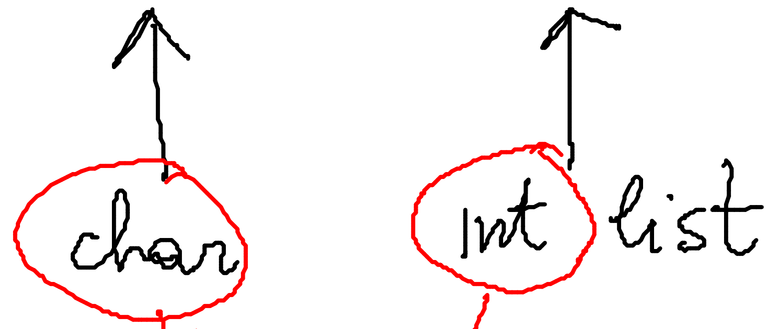
3 ↑ int — [1;2] ↑ int list

```
- : int list = [3;1;2]
```

[3;1;2] è una abbreviazione per

<u>3::(1::(2::[]))</u>

'a' :: [1;2];; **NO** errore di tipo



'a * 'a list → 'a list

o ———— o

Parentesi: come faccio in CAML a conoscere il tipo degli operatori PREDEFINITI?

con **prefix**

prefix + ;;

- : int → int → int = <fun>

prefix :: ;;

- : 'a * 'a list → 'a list

```
# [ [1] ; [] ; [3;4] ] ;;
  - : int list list = [ - - - - ]

# let f x = x + 1 ;
  - ; int -> int = <fun>

# [ f ] ;;
  - : (int -> int) list = [<fun>]
```

:: : 'a * 'a list → 'a list

'a = int

int * int list → int list

int list * int list list → int list list

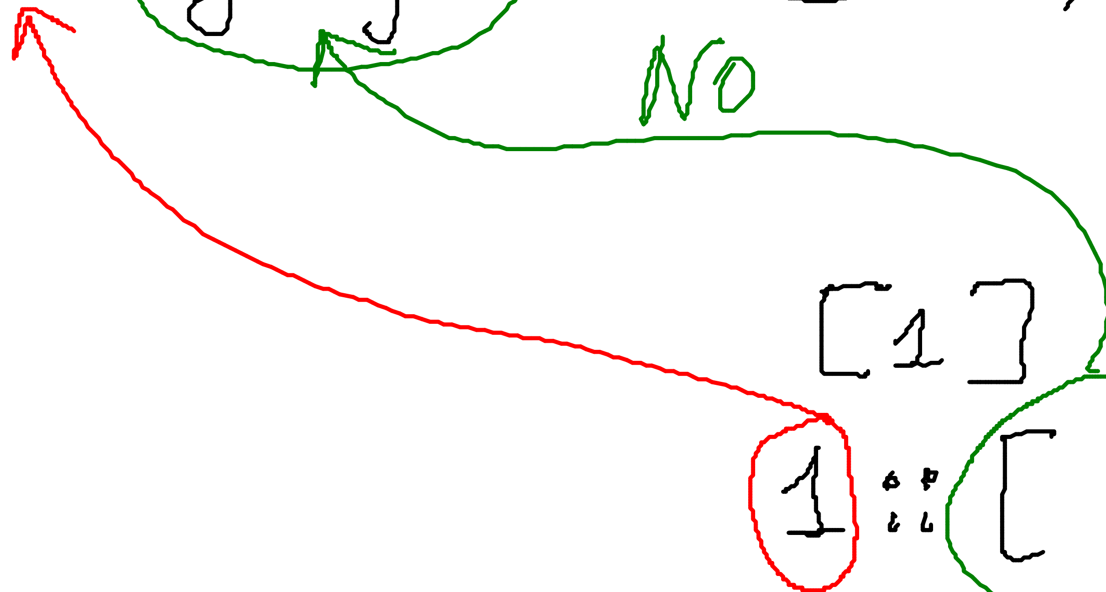
int list * ~~int list~~ → int list

non con ::

PATTERN SU LISTE

- il pattern [] indica la lista vuota
- il pattern x :: xs indica una generica lista con ALMENO un elemento
- il pattern x :: y :: ys indica una generica lista con ALMENO due elementi
- il pattern [x] rappresenta una lista con ESATTAMENTE un elemento

PATTERN	VALORI	ESITO DEL P.M.
$x :: xs$ (x: 'a, xs: 'a list)	$[1; 2; 3]$	$x \leadsto 1$, $xs \leadsto [2; 3]$
	$[1]$	$x \leadsto 1$, $xs \leadsto [\,]$
	$[\,]$	NO
$x :: y :: ys$	$[1; 2; 3]$	$x \leadsto 1$, $y \leadsto 2$, $ys \leadsto [3]$
	$[1]$	NO



ESEMPI

Funzione che controlla se una lista è vuota

```
let null x = match x with
                 [] -> true |
               _ :: _ -> false ;;

null : 'a list -> bool = <fun>
       \______/   \__/
        tipo x   tipo ris.
```

Controlla che una lista abbia un solo elemento

```
let single x = match x with
    [] -> false |
    [y] -> true
    y::z::zs -> false ;;

single : 'a list -> bool = <fun>
```

il cons associa a destra

y::z::zs equivale a y::(z::zs)

OPERATORE @ PREDEFINITO

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

# [1;2] @ [3;1] ;;
- : int list = [1;2;3;1]
```

Non si può usare @ nei PATTERN

```
match l with
  x @ y -> ...
```

NO

Definiamo noi una funzione

append : 'a list → 'a list → 'a list

in modo che

append l1 l2 = l1 @ l2

- se l1 = [] ⟶ l1 @ l2 = l2
- se l1 = x :: xs   l1 @ l2 = x :: (xs @ l2)

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

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

o ———— o

PARENTESI : i nomi usati nel pattern sono LOCALI al pattern

```
z::zs  ->  z :: (append zs l2)
z::l1  ->  z :: (append l1 l2)
```

In una lista che ha la struttura X :: XS

X — head (testa) della lista

XS — tail (coda) della lista

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

# tl ;;
- : 'a list -> 'a list = <fun>

# let hd x = match x with
      y :: _ -> y ;;
hd : 'a list -> 'a = <fun>
```

DIMOSTRAZIONE DI CORRETTEZZA DI APPEND

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

$\forall\, l1, l2 \in$ 'a list . (append l1 l2) = l1 @ l2

Relazione di precedenza INDOTTA :

$x \sqsubset_f y$ se $f(y)$ è definita usando $f(x)$

$(xs, l2) \sqsubset (x::xs, l2)$

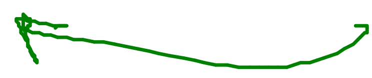
$$([x_1; x_2; \ldots; x_n], \ell)$$

$$([x_2; \ldots; x_n], \ell)$$

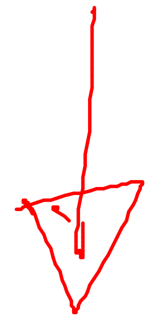
$$\vdots$$

$$([x_n], \ell)$$

$$([\,], \ell)$$



MINIMALE



$$(\forall \ell, \ell' \in \text{'a list}.\ \ell \sqsubset \ell' \equiv \ell' \neq [\,] \land \ell = tl\ \ell')$$

$$(\forall \ell, \ell', \ell_1, \ell_1'.\ (\ell, \ell_1) \sqsubset_{app} (\ell', \ell_1') \equiv \ell' \neq [\,] \land \ell = tl\ \ell' \land \ell_1 = \ell_1')$$

$(\forall l1, l2 \in \text{'a list. append } l1\ l2 = l1 @ l2).$

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

Useremo nella dim. le seguenti proprietà di @

$$x :: (l1 @ l2) = (x :: l1) @ l2 \qquad [\,] @ l = l$$

<u>Caso base</u> considero $([\,], l)$

append [] l

= { def. di append, 1° pattern }

l

= { prop. di @ }

$[\,] @ l$

Caso induttivo $l1, l2$ con $l1 \neq []$ $l1 = x::xs$

append $x::xs$ $l2$

= { def. di append, 2° pattern }

$x :: (\text{append } xs\ l2)$

= { Ip. induttiva : append xs $l2$ = $xs @ l2$ $(xs, l2) \sqsubset (x::xs, l2)$ }

$x :: (xs @ l2)$

= { proprietà precedente }

$x :: xs @ l2$

append xs $l2$ = $xs @ l2$

$\Downarrow$

append $x::xs$ $l2$ = $x::xs @ l2$

GENERALIZZA TAIL

Cancellare da una lista i primi n elementi

drop : int → 'a list → 'a list

(drop n l) è lista ottenuta da l cancellando i primi n elementi

```
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
```

GENERALIZZIAMO HEAD

take n l restituisce la lista formata dai primi n elementi di l

take n l : int → 'a list → 'a list

```
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 2 [3; 5; 8; 1]

= { def. di take, 3° pattern

3 :: (take 1 [5; 8; 1])

= { def. di take, 3° pattern }

3 :: (5 :: (take ∅ [8; 1]))

= { def. di take, 1° pattern }

3 :: 5 :: []

= [3; 5]

1° pattern ∅, l

2° pattern n, []

3° pattern

n = 2 x = 3

xs = [5; 8; 1]

n = 1

x = 5 xs = [8; 1]

n = 0 l = [8; 1]

take 5 [1;2]

= {def. di take 3° patt}

1 :: (take 4 [2])

= {def. di take 3° patt}

1 :: (2 :: (take 3 []))

= {def. di take , 2° patt}

1 :: 2 :: []

= [1;2]

$\forall n, l .\ (\text{take } n\ l)\ @\ (\text{drop } n\ l) = l$

per induzione ben fondata (DOMATTINA)

o ———————— o

LUNGHEZZA DI UNA LISTA

```
let rec length l = match l with
    [] -> 0 |
    _ :: xs -> 1 + (length xs);;

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

N-esimo elemento di una lista

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

let rec nth n l = match (n, l) with
    (1, x::_) -> x |
    (n, _::xs) when n>1 -> nth (n-1) xs

let primo = nth 1;;
  primo : 'a list -> 'a = <fun>
let terzo = nth 3;;
  terzo : 'a list -> 'a = <fun>
```

```
# nth 2 [10; 40; 30];;
- : int = 40
                                   'a = int
nth : int -> 'a list -> 'a          . 'a = int list
# nth 2 [[10]; [40]; [30]].
- : int list = [40]
```