CAML

```
let n = 5;;
n : int = 5
```

```
let f n m = 3*n + m;;
f : int -> (int -> int) = <fun>
```

↑ Curryed

```
let g = f 3;;
g : int -> int = <fun>
```

g è la funzione f a cui abbiamo dato il primo argomento

```
g m = 3*3 + m
```

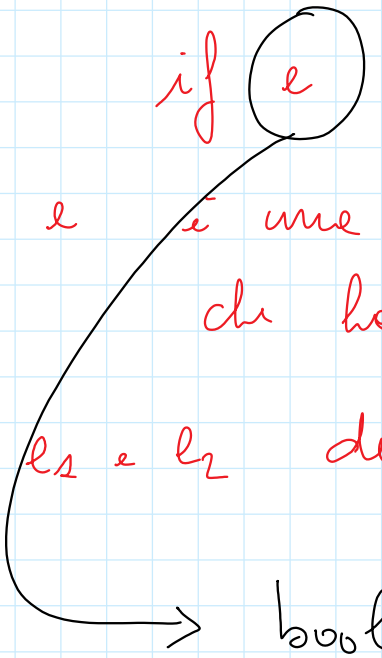
Usando le funzioni Curryed con parte dei loro argomenti abbiamo come RISULTATO una funzione.

Espressione condizionale

if e then e1 else e2

e è una espressione logica (boolean)
che ha come risultato true oppure false

e1 e e2 devono avere lo stesso tipo



Def. ricorsive di funzioni

$f : \mathbb{N} \to \mathbb{N}$

$$f(n) = \begin{cases} \emptyset & \text{se } n = \emptyset \\ 2 + f(n-1) & \text{se } n > \emptyset \end{cases}$$

let rec f n = if n = ∅ then ∅ else 2 + f (n-1) ;;

f : int → int = <fun>

tipo di n

f 3 ;;

- : int = 6

f (-3) ;;

stack overflow

f (3) ;;

= { def. f, remo else }

2 + f 2 ;;

= { def f, remo else }

2 + 2 + f 1 ;;

= { " }

2 + 2 + 2 + f 0 ;;

= { def f, remo then }

2 + 2 + 2 + 0

= { calcolo }

6

Liste in CAML

Sono strutture omogenee (valori dello stesso tipo) con un ordine.

Notazione

```
# [10; -3; 4];;
- : int list = [10; -3; 4]
```

C'è una lista particolare, lista vuota

[]

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

```
# [3; 3.5];;
```
errore di tipo ←

```
# [3.5; 3];;
```

E' definito un operatore di base (costruttore di valori)

(costruttore di valori)

:: (Cons)

operatore INFISSO (che si indica tra gli argomenti)

come +

3 + 4 = 7

4

il primo argomento è un valore di liste

il secondo è una lista con valori dello stesso tipo del primo argomento

es:

3 :: [-2;4]

valore — lista

il risultato è la lista (secondo argomento) con in testa il primo argomento

3 :: [-2;4] = [3;-2;4]

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

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

# 3 :: (-2 :: (4 :: [])) ;;
- : int list = [3; -2; 4]
```

I valori di una lista possono avere tipo qualsiasi (tutti lo stesso tipo)

```
# (3, 4) :: [] ;;
- : int * int list = [(3, 4)]

# (3, "ab") :: [] ;;
- : int * string list = [(3, "ab")]

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

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

# [3;4] :: [5];;
```

errore di tipo

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

# let g n = n+1;;
g : int -> int = <fun>
```

errore di tipo

```
# [g];;
- : int -> int list = [<fun>]

# [g;g];;
- : int -> int list = [<fun>; <fun>]
```

```
# let f n = n > 3 ;;
f : int -> bool = <fun>
# [f; 0];;
errore di tipo
```

Sulle liste sono predefinite due funzioni hd (head) tl (tail)

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

```
# hd [3;4];;
- : int = 3
```

hd non è definita su liste vuote

```
# hd [];;
errore
```

```
# tl;; (tail)
- : 'a list -> 'a list = <fun>
```

tl non è definita su liste vuote
cancella il primo elemento

```
# tl [true; false; false];;
- : bool list = [false; false]

# tl [3; -2; 7];;
- : int list = [-2; 7]
```

```
# let n = 5;;
n : int = 5

# n = n+1;;
- : bool = false
```

```
# n;;
- : int = 5

let n = 6;
n : int = 6;
```

[3; -2; 7]

```
# 4:: tl [3; -2; 7];;
- : int list = [4; -2; 7]
```

```
let n = n+1;;
n : int = 7
```

←

7

---

hd tl

data una lista voglio, come risultato della funzione che devo

risultato della funzione che devo definire, l'ultimo elemento della lista (ci deve essere)

3 :: [] = [3]

```
let rec last l =
    if Tl l = [] then hd l
    else last (Tl l);;
```

last : 'a list → 'a = <fun>

'a list: tipo l

```
last [3;4];;
= { def. last, ramo else }
last [4];;
= { def last, ramo then }
hd [4];;
= { def hd }
```

4

```
let rec removelast l =
    if tl l = [] then []
    else hd l :: removelast (tl l);;
removelast : 'a list -> 'a list = <fun>
```

tipo l

Tipo ris

removelast (tl l)

removelast [3;4;5];;

=

[]

```
removelast [3;4;5];;
= { def rl, ramo else }
3 :: rl [4;5]
= { def rl, ramo else }
3 :: (4 :: rl [5])
= { def rl, ramo then }
3 :: (4 :: [])
= { calcolo }
[3;4]
```

```
# removelast [];;
```

lunghezza di una lista

```
let rec len l =
    if l = [] then 0
    else 1 + len (tl l);;
```

len : 'a list → int = <fun>

'a list: tipo di l

int: tipo ris

PATTERN (MODELLI)

Un pattern è una espressione che contiene VARIABILI (nomi che possono essere instanziati (ai quali posso associare) a valori)

Es `x :: []` è un pattern

i pattern possono essere uguagliati a valori instanziando le variabili a valori (corretti dal punto di viste del tipo)

Es:
il pattern `x :: []` può essere uguagliato al valore `[5]` perché `[5] = 5 :: []` instanziando `x` al valore 5

il pattern `x :: [] = 5 :: [] = [5]`

x :: [] può essere uguagliato a [10]

---

x :: [] ~~può essere uguagliato alla lista~~

5 :: 7 :: []

si basta instanziare x a 5 :: 7

(ma è un valore ma un errore di tipo)

x :: [] è un pattern che può uguagliare a tutte le liste di un solo valore.

pattern

x :: xs — xs per ricordarmi che deve essere una lista

si uguaglia $[3;4;5]$ sì!

$\downarrow \equiv$

$3 :: [4;5]$

$x = 3$

$xs = [4;5)$

A quali liste si può uguagliare

$x :: xs$?

a tutte le liste non vuote

x :: y :: ys

a quali liste si può uguagliare?

3 :: [] no!

[] no!

2 :: 3 :: [] si

2 :: 3 :: [4;5] si

x = 2
y = 3
ys = [4;5]

a tutte le liste di almeno 2 valori

Come si usano i pattern

PATTERN MATCHING

match e with

| pattern1 → e_1
| pattern2 → e_2
| ...

si cerca di uguagliare il valore dell'espressione e con i pattern in sequenza.

Se il valore di e si uguaglia al pattern i il risultato è dato dall'espressione e_1