LINGUAGGIO FUNZIONALE

CAML (camel)

↳ Meta Language

- INTERPRETATO (vs COMPILATO)
- TIPATO - tutti i valori hanno un TIPO
  (= il dominio dei valori)

In CAML si valutano ESPRESSIONI

```
# 3+4;;
- : int = 7
```

- : → (inferenza di tipo)
int → tipo
7 → valore dell'espressione

```
# true & false;;
- : bool = false
```

& → AND (∧)

```
# 'a';;
- : char = 'a'
```

```
# "abcd1234";;
- : string = "abcd1234"
```

```
# "a";;
- : string = "a"
```

```
# 3.0 +. 1.2 ;;
- : float = 4.2
```

```
# let x = 3.5 ;;
x : float = 3.5
```

```
# x +. 2.6 ;;
- : float = 6.1
```

```
# let x = 4 ;;
x : int = 4
```

```
# x + 5 ;;
- : int = 9
```

```
# let y = x + 6 ;;
y : int = 10
```

COSTRUTTORI DI TIPO

permettono di definire valori di tipi ottenuti a partire da altri tipi

COSTRUTTORE → del tipo FUNZIONE

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

f applicata a x

$f(x) = x + 1$

```
# let g x = x +. 1.2 ;;
g : float -> float = <fun>
# x +. 1.2 ;;
```

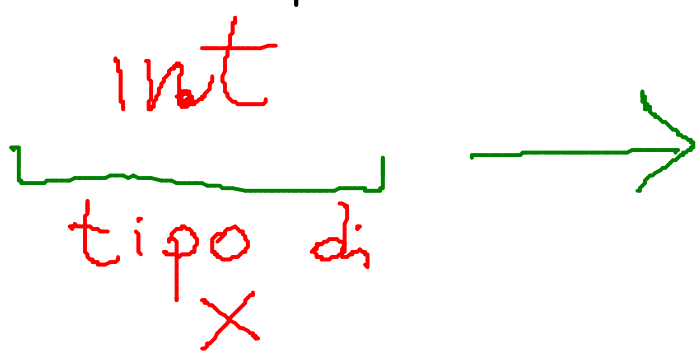
non hanno nulla a che fare l'una con l'altra

```
# let x = 4 ;;
x : int = 4

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

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

Il tipo di una funzione ha questa struttura


```
# let f x = x + 1 ;;
```

1) È una funzione



f x

x è int perché primo operando di +

```
# let x = 4;;
x : int = 4

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

```
# let h z = z + 3 - 2 + z - z;;
h : int -> int = <fun>
```

È facile convincersi che, per ogni intero a,

$$g\ a = h\ a$$

Come si usano le funzioni

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

# f 3 ;;
  - : int = 4
```

```
# f (5+8) ;;
```

← 13 + 1

```
  - : int = 14
```

```
# (f 5) + 8 ;;
```

← (5+1) + 8

```
  - : int = 14
```

```
# f (f 5);;
- : int = 7
```

f 5 = 5+1 = 6

f (f 5) = f 6 = 7

PRODOTTO CARTESIANO *

```
# (3, 5);;
- : int * int = 3, 5
# (3, "abcd", 3.5);;
int * string * float = 3, "abcd", 3.5
```

let g (n, m) = n + m + 1

g : int * int ⟶ int

tipo di (n, m) ⇓ int * int

tipo del risultato

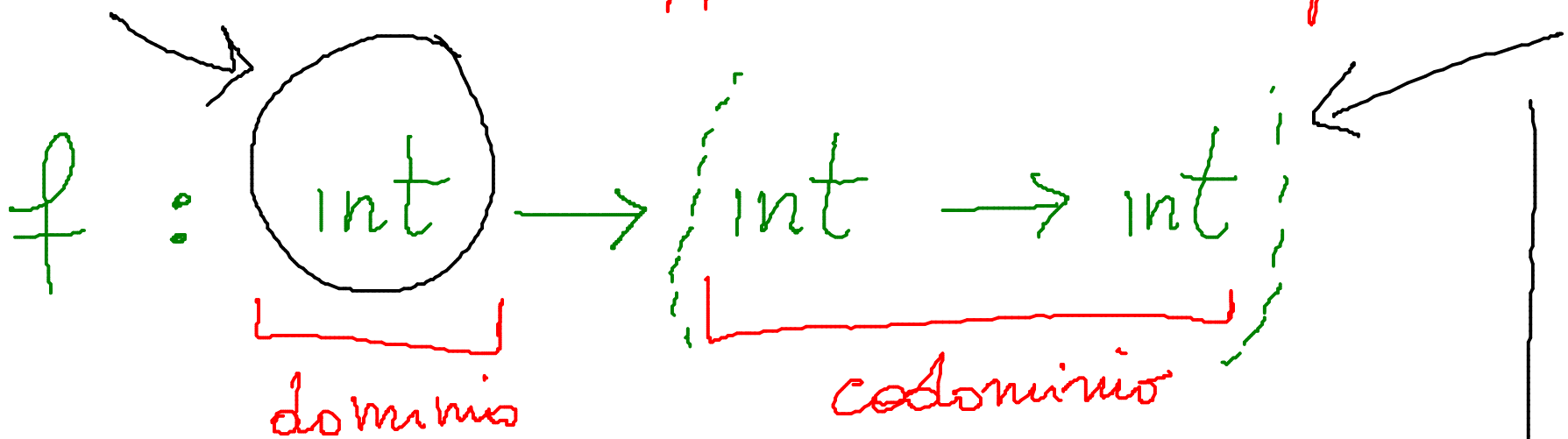
tipo n * tipo m

g 3 ;; No !!

g (3, 5) ;;
- : int = 9

```
# let f n m = n + m + 1 ;;
```

si può applicare al 1° parametro n e calcola "qualcosa" che può essere applicato al 2° parametro




```
# f 26 ;;
- int = 32
```

```
# let h = f 5 ;;
h : int → int = <fun>
```

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

curryed Curry

```
# let g (x,y) = x+y+1;;
g : int * int -> int
```

Uncurryed

```
# f 3;;
- : int -> int = <fun>

# g 3;;
```

ERRORE

```
# let checksum (x,y,z) = x+y = z ;;
```

operatore di uguaglianza predefinito

```
checksum : int * int * int -> bool = <fun>
```

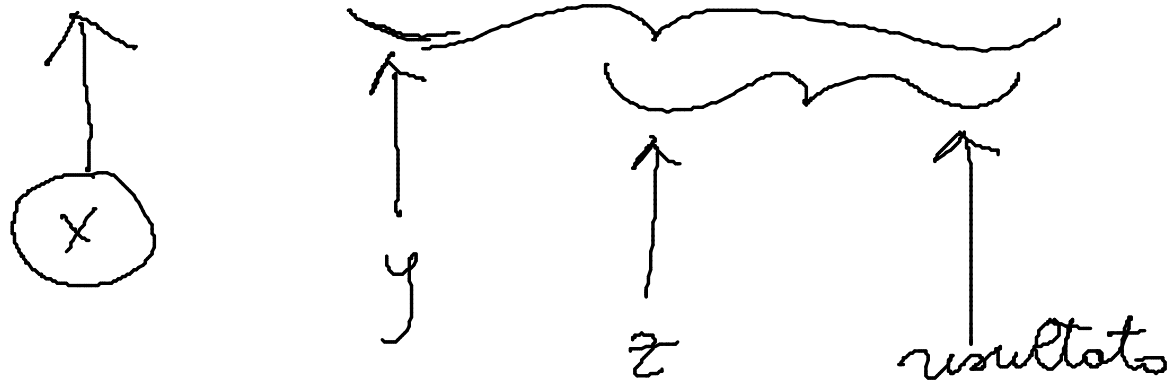
tipo x tipo y tipo z

```
# checksum (10,6,16) ;;
- : bool = true
```

```
# checksum (10,6,25) ;;
- : bool = false
```

let ((cs x) y) z = x + y = z ;;

cs : int → int → int → bool = <fun>



let cs1 = cs 10 ;;

cs1 : int → int → bool = <fun>

let cs2 = cs1 6 ;;

cs2 : int → bool = <fun>

cs2 16 ;;

- : bool = true - 10 + 6 = 16

cs2 16

= (cs1 6) 16

= ((cs 10) 6) 16

= true ;

```
# let x = 42;
  ----
# let y = 26;
  ------
# x + y;;
```

$$= x + y$$

$$= 42 + y$$

$$= 42 + 26$$

$$= 68$$

let CS1 = CS 10;;

è una funzione che, dati y e z, calcola

$10 + y = z$

C2S = CS1 6

è una funzione che, dato z, calcola

$10 + 6 = z$

TIPO DI FUNZIONI

```
let eq (x:int) (y:int) =  x=y ;;
```

```
eq : int → int → bool = <fun>
```

→ x e y devono essere dello stesso tipo

```
let eqq x y = x=y ;;
```

```
eqq : 'a → 'a → bool = <fun>
```

'a → 'a → bool : una famiglia di funzioni

'a tipo x

'a tipo y

bool tipo ris.

variabile di tipo: sta per un generico tipo

eqq è una funzione POLIMORFA

```
# eqq 3 5 ;;
- : bool = false
```

```
# let eq3 = eqq 3 ;;
eq3 : 'a -> bool = <fun>
```

'a → int

```
# let eqa = eqq 'a' ;;
eqa : char -> bool = <fun>
```

FUNZIONI COME ARGOMENTI

```
# let apply x y = x y ;;
```

```
apply : ('a -> 'b) -> 'a -> 'b
```

('a -> 'b) : x

'a : y

'b

x y significa x "applicato" a y

quindi x è una funzione

```
# let f x = x + 10 ;;
f : int -> int = <fun>
# apply f ;;
```

int -> int

let app f x y = f (x y) ;;

app : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

('a * 'b -> 'c) : tipo di f

'a : tipo x

'b : tipo y

'c : risultato