CAML

Meta Language ML

(descrivere le semantiche di linguaggi più complessi (tipicamente i linguaggi imperativi))

F#

CAML

Interprete CAML è in grado di valutare interattivamente espressioni (le chiamate di funzione è una espressione)

La dichiarazione di funzioni in CAML è anch'essa una espressione.

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

tipo del risultato

valore dell'espressione

nome che viene definito dall'espressione (in questo caso nessuno)

Ogni valore (o operatore) in CAML ha un unico tipo.

`+` operatore su interi

`+.` operatore di somma su float

Avere un unico tipo di valori e operatori permette di fare:

INFERENZA DEI TIPI

```
# 3+5;;
- : int = 8
```

float

```
# 3.5 +. 4.1 ;;
- : float = 7.6
```

```
# 3.5 + 4.1 ;;
```

errore di tipo

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

```
# x+1 ;;
- : int = 6
```

uguaglianza

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

```
# let x = 7;;
x : int = 7
# x;;
- : int = 7
```

Come si definiscono le funzioni

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

nome della funz

x viene interpretato come il nome del parametro

In generale non è decidibile dire qual'è la funzione calcolata da un programma

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

```
let f (x) = x + 1 ;;
```

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

nome della funzione

nome del parametro

```
let g (n, m) = n + m + 2 ;;
g : int * int -> int = <fun>
```

$$A \times B = \{(a,b) \mid a \in A \wedge b \in B\}$$

$A \times B = \{(a,b) \mid a \in A \wedge b \in B\}$

```
# g (3,4);;
- : int = 9

# g (3);;
errore
```

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

f: nome della funzione
n: parametri

```
let g n m = n+m+2;;
```

g: nome della funzione
n, m: nome dei parametri

Curry: in forma CURRY ed la funzione g prende gli argomenti in sequenza. Cioè prende il primo argomento e il risultato è una nuova funzione che prende il secondo argomento e dà il risultato

```
let g n m = n+m+1;;
g: int -> (int -> int) = <fun>
```

int → (int → int)
- 1° int: tipo di n
- 2° int: tipo di m
- 3° int: tipo del risultato

Nei tipi → associa a destra:

Nei tipi → associa a destra:

int → int → int ≡ int → (int → int)

```
# g 2 3;;
- : int = 6

# g 2;;
- : int → int = <fun>

# let f = g 2;;
f : int → int = <fun>

# f 3;;
- : int = 6

# let id x = x;;
id : 'a → 'a = <fun>
```

'a: variabili di tipo

polimorfa (si può applicare a valori di tipo diverso)

```
# id 3;;
- : int = 3
# id 3.5;;
- : float = 3.5
```

= per la definizione delle funzioni

let eq x y = x = y ;;

eq : 'a → 'a → bool = <fun>

'a int → tipo di x

'a int → tipo di y

bool → tipo del risultato

operatore di uguaglianza

```
# eq 3 4;;
- : bool = false
```

'a = int

```
# eq 3 4.5;;
```

4.5 : float

errore di tipo

```
# eq 3 3.0;;
```

3.0 : float

errore di tipo

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

# g 2 ;;
- : int -> int = <fun>
```

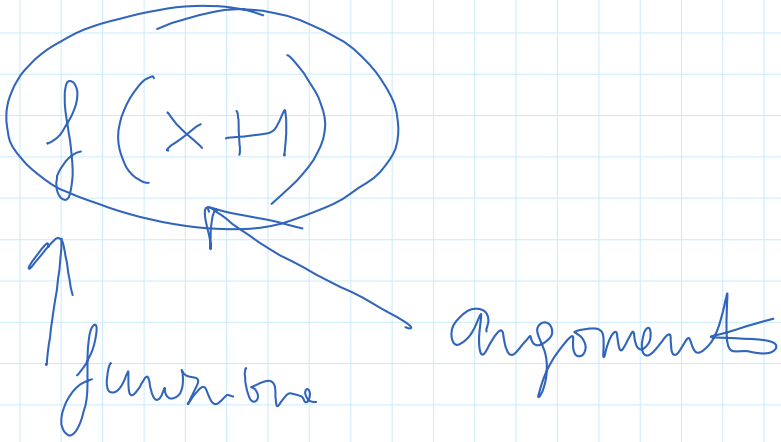
le funzioni possono essere risultato di una applicazione di funzione

È possibile avere funzioni come argomenti di altre funzioni?

Sì!

```
let g f x = (f (x+1)) + 1 ;;
g : (int -> int) -> int -> int
```

- `(int -> int)`: tipo di `f` (funzione che ha come parametro)
- `int`: tipo di `x` (parametro curryed)
- `int`: tipo del risultato


f (x+1)
funzione
argomento

giovedì 14 novembre 2019 17:14

```
let g f x = f (x+1) + 1;;
g : (int -> int) -> int -> int = <fun>
```

l'applicazione di funzione ha precedenza maggiore rispetto alle applicazione degli operatori

```
#let h x = x+2;;
h : int -> int = <fun>
#let s x = x-2;;
s : int -> int = <fun>
```

```
# g h 2;;
- : int = 6
```

il risultato si ottiene valutando

h(2+1) + 1

s(2+1) + 1

```
# g s 2;;
- : int = 2
```

```
#let t = g s;;
t : int -> int = <fun>
```

```
# t 2;;
- : int = 2
```

s(2+1) + 1

Definizioni ricorsive di funzioni

$$f(n) = \begin{cases} \emptyset & \text{se } n = \emptyset \\ f(n-1) + 2 & \text{se } n > 0 \end{cases}$$

$$\forall n \in \mathbb{N}.\ f(n) = 2 \cdot n$$

esiste l'espressione condizionale

if condizione then espressione else espressione

if $(n>0)$ then $n+1$ else $n-1$

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

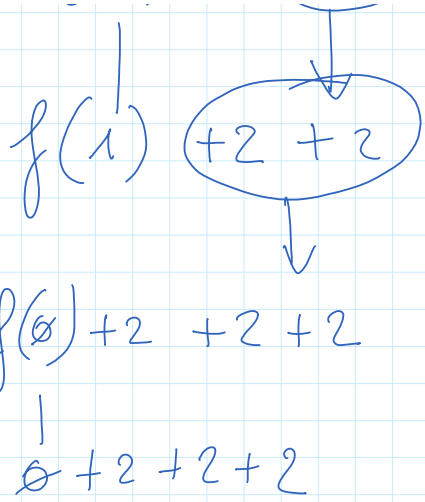
f : int → int = <fun>

tipo di n — tipo del risultato

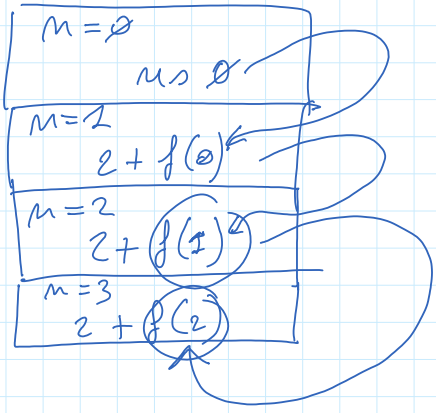
```
# f 3;;
```

f(2) → +2

" ;;
- : int = 6

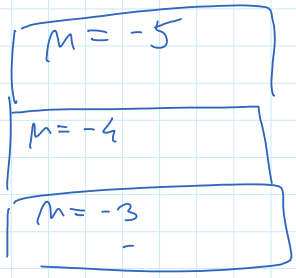


f(3);;
- : int = 6



STACK delle attivazioni

#f(-3);;
STACK OVERFLOW



Fattoriale in CAML

$0! = 1$

$n! = n \cdot (n-1)!$ def ricorsiva

```
#let rec fact n =
    if n=0 then 1
    else n * fact (n-1);;
```

```
fact : int -> int = <fun>
```

```
#fact 4
- : int = 24
```

$$\begin{cases} MCD(n,n) = n \\ MCD(n,m) = MCD(n-m, m) & \text{se } n > m \\ MCD(n,m) = MCD(n, m-n) & \text{se } m > n \end{cases}$$

```
let rec MCD (n,m) =
    if n = m then n
    else if n > m then MCD (n-m, m)
                  else MCD (n, m-n);;
```

MCD : int * int -> int = <fun>

```
# let rec MCD1 n m =
    if n=m then n
      else if n>m then MCD1 (n-m) m
        else MCD1 n (m-n) ;;
```

```
MCD : int -> int -> int = <fun>
```

```
# MCD (12,9);;
- : int = 3
```

```
# MCD1 12 9;;
- : int = 3
```

```
# MCD1 12 ;;
- : int -> int = <fun>
```