Funzioni

Abbiamo visto tre tipologie di forme sintattiche

(1) espressioni → oggetti sintattici con una nozione di riduzione
(2) valori → espressioni non ulteriormente riducibili
(3) definizioni → oggetti sintattici che vincolano valori a nomi

Introduciamo ora una ulteriore categoria: le funzioni

FUNZIONI ANONIME

```
fun x -> e
```

λ-expressions

Corrisponde alla notazione matematica $x \mapsto e$ o $\lambda x.e$ o $\lambda x \to e$

Es. `fun x -> x+1` è la funzione successore

Una funzione anonima è un' espressione

(fun x -> x+1) + (if 3>2 Then 0 else 42) ...

Cosa succede se valutiamo fun x -> x+1 ?

➡ - : int -> int = <fun>

leggendo da dx a sx, abbiamo che

① Il valore di: fun x -> x+1 è una funzione unprintable (<fun>)

② Il tipo della funzione è int -> int

~> Funzioni sono valori

Prima di approfondire...

Es. `fun x y -> x +. y`

2 parametri

↳ somma di float → Ocaml non fa type casting

eval → `float -> float -> float = <fun>`

2 argomenti

ordine superiore

int { + − * / : 0 1 : } float { +. −. *. /. : 0.0 1.0 : }

`fun x y z -> e` avrà tipo

$T_1 \to T_2 \to T_3 \to T_{exp}$

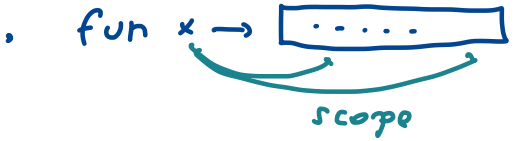
Def. Una funzione anonima è una espressione della forma

$$\texttt{fun } x \to e$$

SINTASSI $\texttt{fun } x \to e$] espressione / corpo della funzione

variabile / parametro formale

La variabile x è legata in e



- $\texttt{fun } x \to e \cong_\alpha \texttt{fun } y \to e[y/x]$

STATICA. Le funzioni hanno tipo freccia

$T, S ::= int \mid float \mid bool \mid \ldots \mid T \to S$

Intuizione: una funzione ha tipo $T \to S$ se prende in input qualcosa di tipo T, e restituisce qualcosa di tipo S

Regola: $fun\ x \to t : T \to S$ se

t ha tipo S assumendo che x abbia tipo T

$$\frac{\Gamma, x : T \vdash t : S}{\Gamma \vdash fun\ x \to t : T \to S}$$

DINAMICA $\quad fun\ x \to t \implies fun\ x \to t$

una funzione è un valore!

Esattamente come lo sono true, 0, 118, ...

➡ Quando si esegue una funzione $fun\ x \to t$

non viene valutato il corpo della funzione

$fun\ x \to x+0 \not\Longrightarrow fun\ x \to x$

anche se $x+0 \implies x$

CONSEGUENZE

- Possiamo scrivere funzioni che restituiscono funzioni

```
fun x -> (fun y -> x + y)
```

(fun y -> x + y) : espressione

zucchero sintattico: `fun x y -> x + y`

2 parametri, ma sintassi forza 1 parametro

`fun x -> t`

- Statica di: `fun x -> (fun y -> x + y)`

`int -> (int -> int)`

Questo processo si chiama **Currying** (dal nome del logico Haskell Curry).

Teorema. Una funzione in n-parametri:

$$\text{fun } x_1 \dots x_n \to t$$

è **simulata** (vedremo dopo)

da una funzione di: **ordine superiore** in 1 parametro

$$\text{fun } x_1 \to (\text{fun } x_2 \to \dots (\text{fun } x_n \to t) \dots)$$

Applicazione di Funzioni

D. Come possiamo usare una funzione? ⇝ Applicazione
(passaggio di argomento)

Def. Date espressioni t, s, l'**applicazione** di t ad s è l'espressione

$$t\ s$$

NB. Non si usano parentesi: $t(s)$, $(t)s$ ...

Ex. $(\text{fun } x \to x + 1)\ 5$

$f\ 2$

:

SINTASSI $t, s ::= \ldots \mid t\ s$

STATICA Se t è un'espressione di tipo $S \to T$ e s è un'espressione di tipo S, allora $t\ s$ ha tipo T

$$\frac{t : S \to T \qquad s : S}{t\ s : T}$$

DINAMICA Per valutare $t\ s$ si fa:

1. Si valuta t, ottenendo un valore necessariamente (se tipaggio ok) $\text{fun } x \to t_0$
2. Si valuta s, ottenendo un valore v
3. Si valuta $t_0[v/x]$

Simbolicamente

$$\frac{t \Rightarrow \text{fun } x \to t_0 \qquad s \Rightarrow v \qquad t_0[v/x] \Rightarrow v_0}{t\,s \Rightarrow v_0}$$

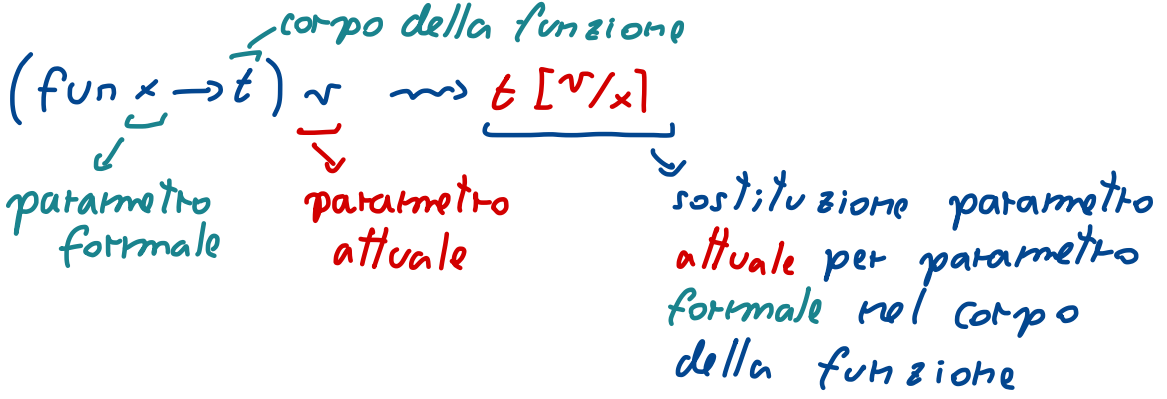
Ex. $(\text{fun } x \to x+1)\ (2+3)$

$\rightsquigarrow (\text{fun } x \to x+1)\ 5$

$\rightsquigarrow 5+1$

$\rightsquigarrow 6$

REGOLA CRUCIALE (β-REGOLA)



➡ Essenza del modello di: computazione per sostituzione

Programmazione di Ordine Superiore

Un linguaggio di programmazione è di: ordine superiore
dati e programmi del linguaggio coincidono

⇝ Un programma può prendere in input e restituire
in output un altro programma

- $\text{fun } x \to (\text{fun } y \to x + y)$

output è funzione

$(\text{fun } x \to (\text{fun } y \to x + y))\ 5 \Rightarrow \text{fun } y \to 5 + y$

- $\text{fun } f \to (\text{fun } x \to f\ (f\ x))$ } doppia iterazione

$(\text{fun } f \to (\underbrace{\text{fun } x \to f\ (f\ x)}_{\text{funzione in output}}))\ (\underbrace{\text{fun } x \to x+1}_{\text{funzione in input}})$

$\cong_\alpha (\text{fun } f \to (\text{fun } y \to f\ (f\ y)))\ (\text{fun } x \to x+1)$

$\overset{\beta}{\rightsquigarrow}\ \underbrace{\text{fun } y \to (\text{fun } x \to x+1)\ ((\text{fun } x \to x+1)\ y)}_{\text{questo è un valore}}$

- Programmazione di ordine superiore è un potentissimo meccanismo di **astrazione**

- Anima dell'informatica
  - Program ≅ Data
  - **Universalità**

- Ma anche comportamenti strani

```
let  Δ = fun x → x x
in   Δ Δ
```

Questo programma si chiama Ω

Definizioni di Funzioni

Spesso conviene dare un nome alle funzioni anonime

```
let add = fun x y -> x + y ;;
```

OCaml offre zucchero sintattico → Definizione: non espressione

```
let add x y = x + y ;;
```

eventualmente con annotazioni di tipo

```
let add (x : int) (y : int) : int = x + y
```

Domanda. Perché parliamo di: programmazione funzionale?

... varie categorie sintattiche

- espressioni
- definizioni
- definizioni di funzione

} → espressione

... varie espressioni:

- 0, 1, 2, ...
- let x = t in s
- fun x → t

} → funzione

Ex. `let x = t in s`

è zucchero sintattico per

`(fun x -> s) t`

$$\frac{t \Rightarrow v \qquad s[v/x] \Rightarrow w}{\texttt{let } x = t \texttt{ in } s \Rightarrow w}$$

$$\frac{\texttt{fun } x \to s \Rightarrow \texttt{fun } x \to s \qquad t \Rightarrow v \qquad s[v/x] \Rightarrow w}{(\texttt{fun } x \to s)\ t \Rightarrow w}$$

} stesso risultato

→ $\texttt{let } x = t \texttt{ in } s \cong_\alpha \texttt{let } y = t \texttt{ in } s$

deriva da $\texttt{fun } x \to s \cong_\alpha \texttt{fun } y \to s[y/x]$ (α-regola)

Piccolo nucleo concettuale

$\underbrace{\text{fun } x \to t}_{\text{astrazione}}$ $\underbrace{t\ s}_{\text{applicazione}}$

$\text{fun } x \to t \cong_\alpha \text{fun } y \to t[y/x]$ α-regola

- scoping
- renaming
- ⋮

$(\text{fun } x \to t)\ v \to_\beta t[v/x]$ β-regola

- computazione

} λ-Calcolo

Ex.

```
let f x y = x - y ;;
f 3 2 ;;
```

nested let-exp

```
let f x y = x - y
in  f 3 2
```

fun. def

```
let f = fun x -> (fun y -> x - y)
in  f 3 2
```

let sotto fun

```
(fun g -> g 3 2) (fun x -> (fun y -> x - y))
```

Ora possiamo applicare la β-regola