# 2023-09-26 - Programmazione funzionale, Lambda-Calculus, OCaML, Funzioni, Scoping, Composizione, Tuple, Pattern Matching

## Programmazione funzionale

L'idea di base è quella di modellare qualsiasi cosa come una **funzione matematica**.

I linguaggi *puramente* funzionali hanno le seguenti caratteristiche:

- Le funzioni sono **oggetti del primo ordine**: tutto quello che può essere fatto con un *"dato"* può essere fatto con una funzione *(passare una funzione ad un'altra funzione...)*
  - i nomi sono irrilevanti
- La **ricorsione** è il principale metodo di controllo del flusso (non esistono *cicli*)
  - le funzioni vengono messe insieme tramite le funzioni composte (**composizione**)
- Il focus principale è sulla **manipolazione di liste**: vengono spesso usate liste con ricorsione su **sotto liste** in sostituzione ai cicli
- Non esistono **side-effect**:
  - data una funzione ed un *input*, l'*output* è sempre **uguale**
  - **non esiste stato**
  - dato che non esiste stato allora **non esistono assegnamenti**
  - l'uso di *statement* è scoraggiato in favore delle *expression evaluation*
- Esistono solo due **costrutti linguistici**:
  - astrazione (**abstraction**): *definizione* delle funzioni
  - applicazione (**application**): *chiamare* le funzioni

*I linguaggi che useremo non sono puramente funzionali, ma queste regole rimangono comunque valide e da usare come best-practice.*

### Motivazioni e Benefici

La maggior parte degli errori in linguaggi "tradizionali" avvengono durante **assegnamenti**. Ma un assegnamento in un linguaggio imperativo o OOP può avvenire *molto* **distante** da dove viene prodotto l'errore.

Eliminando gli *assegnmaneti*, si riducono anche gli errori.

Gli errori, *ovviamente*, possono comunque avvenire, ma sono più **facilmente individuabili** dato che si verificano nel **passaggio di dati tra funzioni**.

## Lambda-Calculus ($\lambda$-calculus)

Sistema matematico definito da Churc e Kleene intorno al 1930.

Tutti i linguaggi funzionali derivano da $\lambda$-calculus.

### Simboli:

Tutte le espressioni sono composte da:

- costanti
- variabili
- $\lambda$
- .
- parentesi

### Regole:

1. Se $x$ è una variabile o una costante allora è una $\lambda$-espressione
2. Se $x$ è una variabile e $M$ è una $\lambda$-expressione allora $\lambda.M$ è una $\lambda$-espressione
3. Se $M, N$ sono $\lambda$-espressioni allora $(MN)$ è una $\lambda$-espressione

#### Astrazione ed Applicazione:

Esistono solo le **due operazioni** basiche di **astrazione** ed **applicazione**:

- $\lambda x.x + 1$ è un esembio di *astrazione* (definizione di una funzione che restituisce il successore)
- $(\lambda x.x + 1)7$ è un esempio di *applicazione* (applica la funzione successore alla costante 7).
  L'applicazione è associativa da sinistra (left-associative): $MNP = (MN)P$

#### Variabili libere (free) e legate (bound)

- In $\lambda x.xy$ $x$ è una variabile *legata* (**bound**), mentre $y$ è *libera* (**free**)
- In $\lambda x.\lambda y.xy$ entrambe le variabili $(x, y)$ sono *legate* (**bound**)
- In $(\lambda x.M)y$ tutte le occorrenze di $x$ in $M$ sono *rimpiazzate* da $y$

## ML

ML (MetaLanguage) è un linguaggio general-pupose sviluppato da Robin Milner negli anno '70. È un'astrazione polimorfa di $\lambda$-calculus.

#### Feature di ML

- una strategia di valutazione call-by-value
- le funzioni sono **oggetti del primo ordine** (first-class)
- polimorfismo parametrico (parametric polymorphism)
- tipizzazione statica (static typing)
- inferenza di tipo (type inference)
- tipi di data algebrici (algebraic data types)
- **pattern matching**
- gestione di eccezioni (pattern handling)
- eager evaluation: tutte le sotto-espressioni sono *sempre* valutate (contrario di lazy-evaluation)
  - è possibile ottenere lazy evaluation grazie alle chiusure (closures)

## OCaML

OCaML (Objective Cambdrige ML) è un'implementazione di ML con alcune funzionalità in più (object-orientation, modules, imperative statements, ...).

OCaML è **compilato** (`ocamlc`), ma ha anche un **interprete** (`ocaml`) e una **shell interattiva** (`utop`).

#### Hello World

```
let main() = print_string("Hello World");;
main();;
```

#### Funzioni

Le funzioni sono **indipendenti** dal loro nome (che è solo un etichetta).

Definizione di funzioni:

```
let succ = fun x -> x+1;;
let succ x = x+1;;
```

È possibile effettuare *aliasing* di una funzione:

```
let succ' = succ;;
```

Per chiamare una funzione è necessario *applicare un argomento* alla funzione (in qualsiasi sua forma, anche anonima):

```
succ 2;;
(fun x -> x+1) 2;;
```

Il nome di una funzione può essere *qualunque* (quindi anche `', -, _, ...`).
L'applicazione è associativa da sinistra (**left-associative**), se non specificato da parentesi.

```
let succ x = x+1;;
succ 5;; (* risultato = 6 *)

succ -3;; (*
  Error: This expresion ha type int -> int
  but an expression was expected of type int
*)

succ (-3);; (* risultato = -2 *)
```

`succ -3;;` va in errore in quanto viene applicato prima `succ` all'argomento `-`. Ma `succ` si aspetta un argomento di tipo *int*, mentre riceve un argomenti di tipo *int->int* (`-` è una funzione che da un intero va al suo intero negativo).
Specificando che prima deve essere applicato `-` a `3` e poi il risultato a `succ` si ottiene il risultato voluto.

Tutte le funzioni **devono** avere dei parametri e restituire qualcosa. In caso non servano parametri o non restituisca nulla esiste `unit`, identificato da `()`, una **tupla vuota**.

```
let do_nothing () = ()
```

**Non esiste** una keyword di `return` esplicità, il valore di ritorno è sempre **l'ultima espressione**. Questo porta ad un *unico punto di uscita* da ogni funzione (ad differenza di *infiniti punti di uscita* di linguaggi "tradizionali").

Il compilatore (o interprete) quando non conosce il tipo di un valore (non riesce a fare inferenza di tipo) lo segna come `'a`, `'b`, ...

```
let f x = 5;;
val f : 'a -> int = <fun> (* output interprete *)

let f g = g;;
val f : 'a -> 'a = <fun> (* output interprete *)
```

### Scoping

I nomi sono *solo etichette*, quando un nome viene "sovrascritto", allora quello vecchio viene sovrascitto e viene "perso".

È usato lo **static binding** al tempo di definizione di una funzione. Questo porta all'esistenza di chiusure sintattiche (**closures**).
Il valore di una costante o una variabile *viene valutato* quando viene *creata una funzione*. In caso quel valore venga *modificato dopo* la definizione della funzione, quello all'interno della funzione rimarrà invariato (chiusura).

```
let y = 5;;
let addy = fun x -> x+y;;
addy 8;; (* 13 *)
let y = 10;;
addy 8;; (* 13 *)
(fun x -> x+y) 10;; (* 18 *)
```

5 viene valutato a *tempo di definizione* di `addy` e rimane *"in pancia"* di `addy` (**chiusura / closure**), quindi anche se viene ridefinita non cambia dentro ad `addy`.

### Composizione

La valutazione di funzioni composte avviene **da sinistra a destra**.

```
let compose f g x = f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

La valutazione di ogni parametro (`f`, `g`, `x`) **dipende dalle operazioni** che vengono **definite nel corpo** della funzione (dopo `=`).

- `f` viene valutata come funzione (`tipo -> tipo`) dato che ha almeno un parametro

  - input: prende come parametro `(g x)`, di cui non sappiamo il tipo, quindi `'a`
  - output: restituisce un valore che di cui non sappiamo il tipo, quindi `'b`
  - `f : 'a -> 'b`
- `g` viene valutata come funzione (`tipo -> tipo`) dato che ha almeno un parametro
  - input: prende come parametro `x`, di cui non sappiamo il tipo, quindi `'c`
  - output: restituisce il valore che viene preso in input da `f`, quindi deve essere di tipo `'a`
  - `g : 'c -> 'a`
- `x` viene valutato come variabile dato che non gli viene applicato alcun paramentro
  - `x` viene passato come parametro (input) a `g`, quindi deve essere di tipo `'c`
  - `x : 'c`

Vengono *concatenati* separati da `->` i *tipi dei parametri* nell'ordine in cui sono definiti, quindi:
`('a -> 'b) -> ('c -> 'a) -> 'c`.

Viene infine valutato il *tipo restituito dalla funzione* `compose`, ovvero il tipo restituito dal *body della funzione*, quindi il tipo restituito da `f` (`'b`) e viene concatenato alla fine dei parametri:
`('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`

### Tuple

È possibile passare più parametri interpretati come un unico parametro attraverso le **tuple**, rappresentate da **parentesi** i cui membri sono separati da **virgole**: `(f, g)`.
Vengono rappresentate dall'*interprete* come i tipi dei vari elementi separati da un `*`: `('a -> 'b) * ('c -> 'a)`

```
let compose' (f, g) x = f (g x);;
val compose' : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>
```

La valutazione dei tipi avviene in modo esattamente *uguale* all'esempio precedente, dato che il *body della funzione è uguale* (e l'inferenza di tipo dipende solo da quello), quello che cambia è come sono stati passati i parametri.

- `f : 'a -> 'b`
- `g : 'c -> 'a`
- `x : 'c`

Dato che f e g sono passati in una *tupla* `(f, g)`, verranno rappresentati come `('a -> 'b) * ('c -> 'a)`.

### Pattern matching

Le funzioni possono essere definite per **pattern matching**.
I pattern possono contenere:

- costants, tuples, records, variant constructors, variable names
- *catch-all* pattern (`_`): *default* che cattura tutti i valori
- *sub-pattern*: contengono alternative `pat1|pat2`

Quando un pattern viene abbinato:

- se è presente la *clausola opzionale* `when` a guardia dell'abbinamento allora viene valutata
  - se positiva viene restituita l'espressione
  - altrimenti nulla
- se non è presente `when` allora viene *restituita l'espressione*

```
match expression with
  | pattern when boolean expression -> expression
  | pattern when boolean expression -> expression
  | _ -> expression ;;
```

```
let invert x =
  match x with
    | true -> false
    | false -> true ;;
```

```
let invert' = function
  true -> false | false -> true ;;
```