## 2023-10-02 - Ricorsione, Ricorsione in coda, Tipizzazione, Int, Float, Booleani, Stringhe, Liste, Tuple, Array, Record, Aliasing, Varianti

### Ricorsione

Una funzione è detta ricorsiva se è *definita* attraverso *sé stessa*.

Vengono definiti due casi:

- **base**: caso che ha un risultato noto e termina la ricorsione
- **passo ricorsivo *(induttivo)***: viene richiamata la funzione stessa *(con argomento diverso, spesso diminuito)*

Una funzione può essere ricorsiva **direttamente** (il corpo della funzione chiama la funzione stessa) o **indirettamente** (il corpo della funzione chiama un'altra funzione che a sua volta richiama la funzione originale - **mutual recursion**).

In OCaML è necessario specificare esplicitamente quando una funzione è ricorsiva attraverso la keyword `rec`.

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

Ogni invocazione di una funzione crea un **frame di attivazione** sullo stack, che ha *tutte le informazioni* per eseguire la funzione.

Una *peculiarità* delle funzioni ricorsive (e di OCaML grazie al punto di ritorno sempre alla fine) è che *impilando* varie funzioni sullo stack il **valore di ritorno** di una funzione e il **parametro della successiva** sono **adiacenti**.

La ricorsione spesso è più intuitiva e naturale da scrivere dell'equivalente iterativa, ad esempio:

```
let rec fibo(n) =
    if n <= 1 then n
    else fibo(n-1) + fibo(n-2);;
```

```
let fibo(n) =
    let fib' = ref 0 and fib'' = ref 1 and fib = ref 1 in
        if n <= 1 then n
        else
            (for i=2 to n do
                fib := !fib' + !fib'';
                fib' := !fib'';
                fib'' := !fib;
            done;
            !fib);;
```

Le **performance** però sono **molto differenti**: la versione ricorsiva utilizza *1605* microsecondi per calcolare `fibo(50)`, mentre quella iterativa *0* microsecondi.

Ma cosa **differenzia** un frame dagli altri per cui è **necessario crearli** (oltre ai parametri)?
Le variabili locali (locals), che obbligano il compilatore a creare ad allocarle e quindi a creare il frame.

```
let rec trfibo goal current last next =
    if (goal = current) then next
    else (trfibo goal (current + 1) next (last + next));;

let fibo n =
    if n <= 1 then 1
    else trfibo n 1 0 1;;
```

Le variabili locali (locals) vengono **estratte** ed utilizzate come **parametri** ed utilizzate come **accumulatori**. Questo rende tutti gli stack di esecuzione *uguali*, il compilatore lo *riconosce* e **non** li crea tutti, ottimizzando il tempo di runtime.

```
let rec trfact goal current last_res =
    if (current = (goal+1)) then last_res
```

```
    else trfact goal (current + 1) (current * last_res);;

let fact n =
    if n <= 2 then n
    else trfact n 2 1;;
```

Grazie all'**ottimizzazione** della **ricorsione in coda** le performance sono *paragonabili* con quelle iterative *(0 microsecondi)*.

## Tipizzazione

ML (e quindi anche OCaML) è **fortemente** (il tipo *non cambia* durante l'esecuzione) e **staticamente** (il tipo è *già noto* a tempo a tempo di compilazione) **tipizzato**.

I tipi sono **inferiti dall'uso** che si fa di quei valori.

### Interi e Float

I tipi int e float non sono interscambiabili e non avviene il cast automatico.

```
let a = 4.0 * 1.0;;
(* This espression has type float but an expression was expected of type float *)
```

Dato che non avviene il cast automatico, quindi anche gli **operatori sono tipizzati**. Ogni operazione ha il suo *equivalente* per i float utilizzando il simbolo `.`: `*.`, `+.`, `/.`, `-.`.

Un float letterale viene specificato aggiungendo il `.` dopo il numero esattamente come per le operazioni.

```
let square x = x *. x;;
val square : float -> float = <fun> (* sono dei float dato che si usa *. *)

square 5;; (* errore *)
square 5.;; (* valido *)
```

### Booleani

`true` e `false` (minuscoli)

Gli operatori logici sono: `&&` (and), `||` (or), `not` (not).

### Stringhe

Il tipo stringa è nativo in OCaML. Sono **immutabili** *(esisteva un metodo per modificare un carattere, deprecato dalla versione 4)*.

Operazioni su stringhe: `^` (concatenazione), `.[index]` (accedere ad un carattere). Altre operazioni sono disponibili nel pacchetto `String`.

```
let s1 = "walter" and s2 = "cazzola";; (* s1: walter, s2: cazzola *)
let s = s1 ^ " " ^ s2;; (* s : walter cazzola *)
s.[9];; (* 'z' *)
String.length(s);; (* 14 *)
```

Per modificare una string è necessario passare attraverso i **bytes** (dato che sono immutabili).

```
let b = Bytes.of_string s;;
Bytes.set b 0 'W';; (* modificare carattere 0 di b *)
Bytes.set b 7 'C';; (* modificare carattere 7 di b *)
let s = Bytes.to_string b;; (* Walter Cazzola *)
```

### Liste

Le liste in OCaML sono **omogenee** (tutti gli elementi sono dello stesso tipo), gli elementi sono separati da `;` e sono racchiusi da `[ ]`. **Non** sono ad **accesso diretto**.

Operatori su liste: `::`, pronunciato *cons* (prepend, aggiungere un elemento all'inizio), `@` (concatenazione di due liste, **inefficiente**).

Il modulo `List` contiene molte funzioni per effettuare operazioni su liste:

- `length 'a List -> int`: lunghezza di una lista
- `nth 'a List int -> 'a`: restituisce l'elemento nella posizione passata *(tempo lineare O(n))*
- `init int -> (int -> 'a) -> 'a list`: crea una generata con la funzione passata
- `rev 'a list -> 'a list`: restituisce la lista invertita
- `rev_append 'a list -> 'a list -> 'a list`: aggiunge un elemento alla fine della lista (invertendola, appendendo in testa e re-invertendo)
- `equal ('a -> 'a -> bool) -> 'a list -> 'a list -> bool`: compara due liste attraverso il comparatore passato come funzione
- `compare ('a -> 'a -> int) -> 'a list -> 'a list -> int`: comparatore su due liste comparando gli elementi singoli attraverso il comparatore passato come funzione
- `iter -> ('a -> unit) -> 'a list -> unit`: effettua l'operazione passata come funzione su tutti gli elementi della lista (l'operazione deve restituire `unit`)
- `map -> ('a -> 'b) -> 'a list -> 'b list`: effettua l'operazione passata come funzione su tutti gli elementi della lista, restituisce la lista modificata
- `fold_left ('acc -> 'a -> 'acc) -> 'acc -> 'a list -> 'acc`: applica l'operazione passata su tutti gli elementi della lista, accumulandoli in `acc`
- `iter2 ('a -> 'b -> unit) -> 'a list -> 'b list -> unit`: effettua l'operazione passata sugli elementi di due liste parallelamente (l'operazione deve restituire `unit`)
- `map2 -> ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list`: effettua l'operazione passata come funzione su tutti gli elementi delle due liste parallelamente, restituisce una lista risultato
- `fold_left2 ('acc -> 'a -> 'b -> 'acc) -> 'acc -> 'a list -> 'b list -> 'acc`: applica l'operazione passata su tutti gli elementi delle due liste, accumulandoli in `acc`
- `for_all ('a -> bool) -> 'a list -> bool`: restituisce se tutti gli elementi della lista rispettano la funzione passata
- `exists ('a -> bool) -> 'a list -> bool`: restituisce se esiste un elemento che soddisfa la funzione passata
- `find ('a -> bool) -> 'a list -> 'a`: restituisce l'elemento che soddisfa la funzione passata
- `find_index ('a -> bool) -> 'a list -> int`: restituisce l'indice dell'elemento che soddisfa la funzione passata
- `filter ('a -> bool) -> 'a list -> 'a list`: restituisce una lista di elementi che rispettano la funzione passata
- `sort ('a -> 'a -> int) -> 'a list -> 'a list`: ordina la lista usando il comparatore passato

### Tuple

Liste **fissate in lunghezza** ma **eterogenee** (possono contenere elementi di diversi tipi). Ogni elemento è separato da `,` e sono racchiusi da `( )`. I diversi tipi vengono indicati dall'interprete come il prodotto cartesiano di essi: `int * char` indica una tupla `(4, 'a')`.

Si accede agli elementi della tupla attraverso la sintassi `.[index]`.

Esistono delle tuple speciali, le **coppie**, che sono composte da solo *due elementi*. Solo sulle coppie esistono le funzioni `fst` e `snd` che accedono rispettivamente al *primo* e al *secondo* elemento della tupla.

### Array

Gli array in OCaML sono omogenei (tutti gli elementi devono avere lo stesso tipo) e mutabili. Ogni elemento è separato da `;` e sono racchiusi da `[| |]`.

Sono direttamente accessibili attraverso la sintassi `.(index)` ed è possibile modificare un elemento attraverso `.(index) <- newvalue`.

Il modulo `Array` contiene molte funzioni per effettuare operazioni su array:

- `length 'a array -> int`: restituisce la lunghezza dell'array
- `make int -> 'a -> 'a array`: restituisce un array di lunghezza passata con tutti gli elementi impostati all'elemento passato

- `init int -> (int -> 'a) -> 'a array`: restituisce un array inizializzato con la funzione passata
- `make_matrix int -> int -> 'a -> 'a array array`: restituisce una matrice (un array di array)
- `append 'a array -> 'a array -> 'a array`: concatena due array
- `sub 'a array -> int -> int -> 'a array`: restituisce il sub-array tra i due parametri passati
- `copy 'a array -> 'a array`: crea una copia di un array
- `to_list 'a array -> 'a list`: converte un array ad una lista
- `of_list 'a list -> 'a array`: converte una lista ad un array
- `iter ('a -> unit) -> 'a array -> unit`: applica la funzione passata a tutti gli elementi dell'array (la funzione deve restituire `unit`)
- `map ('a -> 'b) -> 'a array -> 'b array`: effettua la funzione passata su tutti gli elementi dell'array, restituisce l'array risultato
- `fold_left ('acc -> 'a -> 'acc) -> 'acc -> 'a array -> 'acc`: applica l'operazione passata su tutti gli elementi dell'array, accumulandoli in `acc`
- `iter2 ('a -> 'b -> unit) -> 'a array -> 'b array -> unit`: effettua l'operazione passata sugli elementi di due array parallelamente (l'operazione deve restituire `unit`)
- `map2 ('a -> 'b -> 'c) -> 'a array -> 'b array -> 'c array`: effettua l'operazione passata come funzione su tutti gli elementi dei due array parallelamente, restituisce un array risultato
- `for_all ('a -> bool) -> 'a array -> bool`: restituisce se tutti gli elementi dell'array rispettano la funzione passata
- `exists ('a -> bool) -> 'a array -> bool`: restituisce se esiste un elemento che soddisfa la funzione passata
- `find_opt ('a -> bool) -> 'a array -> 'a`: restituisce l'elemento che soddisfa la funzione passata
- `find_index ('a -> bool) -> 'a array -> int`: restituisce l'indice dell'elemento che soddisfa la funzione passata
- `sort ('a -> 'a -> int) -> 'a array -> unit`: ordina l'array utilizzando il comparatore

### Record

I record in OCaML sono delle strutture **eterogenee** accessibili attraverso il **nome** del campo. Possono essere **mutabili** se *espressamente* specificato.

```
type person = {name: string; mutable age: int};;
let p = {name = "Walter"; age = 35};;
p.name;;
p.age <- p.age+1;;
```

### Aliasing

È possibile definire dei **tipi di dato personalizzati** attraverso l'*aliasing*, ovvero associare un *nome* ad un tipo o a diversi tipi.

```
type int_pair = int * int;;
let a : int_pair = (1,3);; (* cast esplicito *)
```

### Varianti

Una variante in OCaML è un **tipo di dato personalizzato** che può essere definito attraverso **diverse possibilità** (*varianti*). Ognuna di queste varianti è definita attraverso il suo **costruttore** (che inizia con la *lettera maiuscola*) e può avere *nessuno* o *più parametri*.

*Esempio: tipo capacità*

```
type capacity = Empty | Partial of int | Full;;
```

Questo tipo personalizzato *capacità* (`capacity`) può essere *vuoto* (`empty`), *parzialmente pieno* (`partial`) o *pieno* (`full`).
Vuoto o pieno non hanno parametri, mentre parzialmente pieno specifica *quanto* è pieno attraverso un *intero* (`Partial of int`).

L'**inizializzazione** di una variabile di questo tipo avviene attraverso il suo **costruttore**.

```
let e = Empty;; (* val e : capacity = Empty *)
let p = Partial;; (* The constructor Partial expects 1 argument(s) *)
let p = Partial 40;; (* val p : capacity = Partial 40 *)
```

*Esempio: carte da gioco*

```
type card = Card of regular | Joker
    and regular = { suit : card_suit; name : card_name; }
    and card_suit = Heart | Club | Spade | Diamond
    and card_name = Ace | King | Queen | Jack | Simple of int;;
```

Nella definizione delle carte, vengono attribuiti come parametri di costruttori dei tipi che *ancora* **non esistono**, ma verranno dichiarati *più tardi* (`regular`, `card_suit`, `card_name`). Questo è possibile grazie alla **keyword** `and` *(dichiarati separatamente sarebbe andato in errore, andavano prima definiti i tipi utilizzati e solo dopo usati).*

È possibile utilizzare le varianti nel **pattern matching**, attraverso il **costruttore**.

*Esempio: valore carte da gioco*

```
let value card =
    match card with
    | Joker -> 0
    | Card {name = Ace} -> 11
    | Card {name = King} -> 10
    | Card {name = Queen} -> 9
    | Card {name = Jack} -> 8
    | Card {name = Simple n} -> n;;

value (Card {suit = Club; name = Simple 4});; (* 4 *)
value Joker;; (* 0 *)
value (Card {suit = Spade; name = Queen});; (* 9 *)
```