Liste e Pattern Matching

Liste

In OCaml le liste sono un tipo di dato predefinito
Lista = sequenza finita e immutabile di valori dello stesso tipo

```
let numeri = [3; 5; -1; 9; 14; 21] ;;
```

```
val numeri : int list = [3; 5; -1; 9; 14; 21]
```

Inoltre, la lista vuota si rappresenta semplicemente come
`[]` e ha tipo generico:

```
let lista_vuota = [] ;;
```

```
val lista_vuota : 'a list = []
```

Da questi esempi è evidente che una lista si rappresenta come una sequenza di valori racchiusi tra parentesi quadre e separati da punto e virgola.
Il fatto che sia una sequenza immutabile significa che non sarà possibile modificare (aggiungere, rimuovere o modificare) gli elementi della lista.

Ogni operazione può leggere e processare gli elementi, e costruire nuove liste a partire da quelle esistenti, ma sempre senza possibilità di modificarle.

```
let stringhe = ["cane"; "gatto"; "rana"; "gnu"];;
```

```
val stringhe : string list = ["cane"; "gatto"; "rana"; "gnu"]
```

```
let tuple = [ (1,"lun"); (2,"mar"); (3,"mer") ] ;;
```

```
val tuple : (int * string) list = [(1, "lun"); (2, "mar"); (3, "mer")]
```

```
let funzioni = [ (fun x -> x+1) ; (fun x -> x-1) ; (fun x -> x) ] ;;
```

```
val funzioni : (int -> int) list = [<fun>; <fun>; <fun>]
```

```
let liste = [ [1; 2; 3] ; [1; 3; 2] ; [2; 1; 3] ; [2; 3; 1] ; [3; 1; 2] ; [
```

```
val liste : int list list =
    [[1; 2; 3]; [1; 3; 2]; [2; 1; 3]; [2; 3; 1]; [3; 1; 2]; [3; 2; 1]]
```

☝🏻 OCaml inferisce il tipo di una lista unificando i tipi dei suoi elementi

```
[] ;;
```

```
- : 'a list = []
```

```
[[]; []] ;;
```

```
- : 'a list list = [[]; []]
```

```
[[]; []; [3]] ;;
```

```
- : int list list = [[]; []; [3]]
```

```
[[]; []; ["ciao"]] ;;
```

```
- : string list list = [[]; []; ["ciao"]]
```

💡 L'unificazione dei tipi è trovare il tipo più generico possibile che sia compatibile con tutti gli elementi della lista

L'operatore cons ::

Data una lista `l` di tipo `T list` e un elemento e di tipo `T`, si denota con `e :: l` la lista in cui il primo elemento è `e` seguito dagli elementi in `l`

```
let l1 = [3;2;1] ;;
let l2 = 4 :: l1 ;;
```

```
val l1 : int list = [3; 2; 1]
val l2 : int list = [4; 3; 2; 1]
```

La notazione `[1; 2; 3; 4]` è in realtà ***zucchero sintattico*** per la notazione

```
let lis = 1 :: (2 :: (3 :: (4 :: [] ))) ;;
```

```
val lis : int list = [1; 2; 3; 4]
```

che può essere scritta più semplicemente così (essendo `::` associativo a destra):

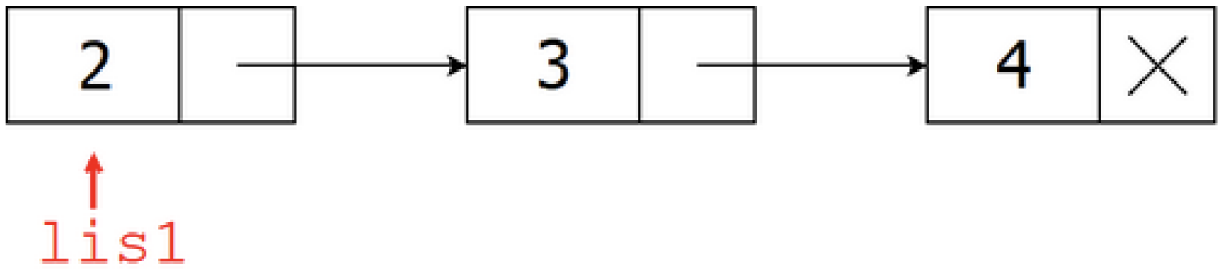
```
let lis = 1 :: 2 :: 3 :: 4 :: [] ;;
```

```
val lis : int list = [1; 2; 3; 4]
```

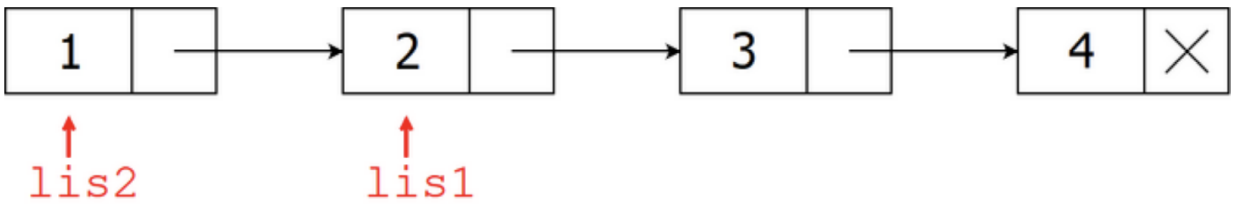
☝️ Le liste sono immutabili, quindi `e :: l` concettualmente è una lista diversa con dentro e e gli elementi di `l` e quindi non è esatto dire che `e` viene aggiunto in testa ad `l`

Le liste in OCaml sono concepite come *liste concatenate singole*

La lista lis1 = [2; 3; 4] corrisponde a:



La lista lis2 = 1 :: lis può essere ottenuta concatenando in testa:



Questa caratteristica delle liste in OCaml, ovvero l'essere immutabili, permette al programmatore di trattarle come liste diverse e quindi di conseguenza risparmiare memoria.

Concatenazione di liste

Date due liste ***lis1*** e ***lis2***, l'operazione `append lis1 @ lis2` descrive la loro concatenazione in un'unica lista

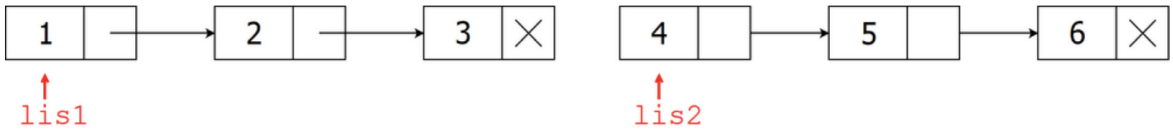
```
let lis1 = [1;2;3] ;;
let lis2 = [4;5;6] ;;
let lis3 = lis1 @ lis2 ;;
```

```
val lis1 : int list = [1; 2; 3]
val lis2 : int list = [4; 5; 6]
val lis3 : int list = [1; 2; 3; 4; 5; 6]
```

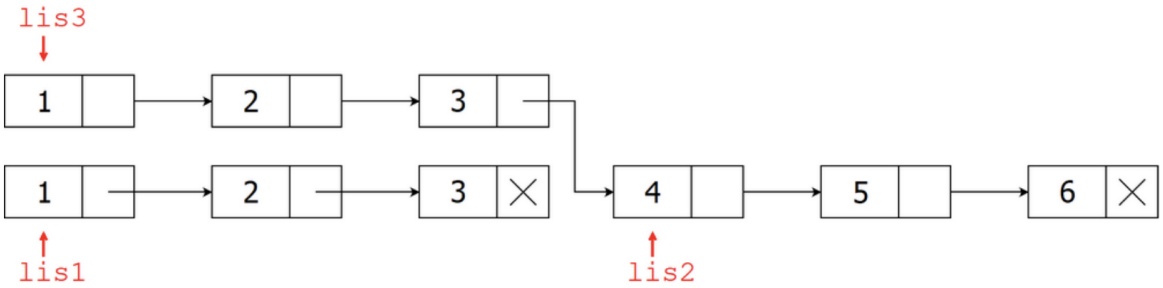
☝️ Internamente, la concatenazione crea una copia della prima lista

esempio:

Date `lis1 = [1; 2; 3]` e `lis2 = [4; 5; 6]`



Avremo il seguente risultato:



Altre operazioni su liste

Il modulo List dell'OCaml API fornisce moltissime funzioni per l'elaborazioni di liste

```
List.length [5;2;1] ;; (* lunghezza della lista *)
```

```
- : int = 3
```

```
List.hd [5;2;1] ;; (* head - primo elemento della lista *)
List.tl [5;2;1] ;; (* tail - elementi successivi al primo *)
```

```
- : int = 5
- : int list = [2; 1]
```

```
List.rev [5;2;1] ;; (* rovescia la lista *)
```

```
- : int list = [1; 2; 5]
```

Pattern Matching

Per accedere agli elementi di una lista è necessaria un'operazione di **destrutturazione**

Sintassi:

$$
\begin{array}{c}
match\ EXP\ with \\
|P_1 -> EXP_1 \\
|P_2 -> EXP_2 \\
\dots \\
|P_N -> EXP_N
\end{array}
$$

Semantica Informale:

- il risultato di EXP viene confrontato con i pattern $P_1, ..., P_n$
- se P_i è il primo pattern con cui fa match, si valuta l'espressione EXP_i

Sintassi dei Pattern

Considerando i tipi visti fino ad ora (tipi di base, tuple e liste) la sintassi dei pattern è data da:

valori di tipi base (non funzioni): true,false,0,1,2,2.3,4.5,'a','b','c',"abc"

- variabili: x,y,z,. . .
- tuple: (P_1,...,P_N)
- liste di lunghezza fissata: [P_1,...,P_N]
- liste con cons: P_1 :: P_2
- wildcard: _

dove P_1,. . . .P_N sono a loro volta dei pattern

Match

Un valore ***v*** fa match con un pattern P se:

- P=_
- P=v
- esiste un modo di istanziare le variabili in P ottenendo P' tale che P'=v

Pattern	Fanno match	Non fanno match
`1`	`1`	`0,2,3`
`(3,2)`	`(3,2)`	`(2,3),(4,2)`
`(x,y)`	`(3,2),(2,3),(4,2)`	`(1,3,2),(1,4,3,2)`
`(x,2)`	`(3,2),(4,2),("ciao",2)`	`(2,3),(4,2,1)`
`[]`	`[]`	`[3],[1;5],['a','b','c']`
`[3]` o `(3::[])`	`[3]`	`[],[1;5],['a','b','c']`
`3::x`	`[3],[3;4;5]`	`[],[1;5],['a','b','c']`
`x::y`	`[3],[3;4;5],[1;5],['a','b','c']`	`[]`
`_`	`5,true,"abc",(3,"hello"),[3;2;4],[]`	

esempi:

```
let negazione b =
    match b with
     | true -> false
     | false -> true ;;
```

```
val negazione : bool -> bool = <fun>
```

```
let rec fibonacci n =
    match n with
     | 0 -> 0
     | 1 -> 1
     | _ -> fibonacci (n-1) + fibonacci (n-2);;
```

```
val fibonacci : int -> int = <fun>
```

La vera potenza del pattern matching è che consente di “smontare” le strutture dati:

- processare i singoli elementi di una tupla
- processare i singoli elementi di una lista
- estrarre una sottolista

```
let primo t =
    match t with
     | (x,_) -> x ;;
```

```
val primo : 'a * 'b -> 'a = <fun>
```

```
let somma t =
    match t with
     | (x,y) -> x+y ;;
```

```
val somma : int * int -> int = <fun>
```

Utilizzando il pattern matching implicito nel `let` queste ultime due funzioni possono essere abbreviate come:

```
let primo (x,y) = x ;;
let somma (x,y) = x+y ;;
```

```
val primo : 'a * 'b -> 'a = <fun>
val somma : int * int -> int = <fun>
```

Accorpare casi del pattern matching

```
let giorno_festivo n =
    match n with
     | 1 | 2 | 3 | 4 | 5 -> "feriale"
     | 6 | 7 -> "festivo"
     | _ -> "ERRORE" ;;
```

```
val giorno_festivo : int -> string = <fun>
```

Funzioni ricorsive su liste

Grazie al pattern matching possiamo ora scrivere funzioni ricorsive su liste

```
let rec length lis =
    match lis with
     | [] -> 0
     | x::lis' -> 1 + length lis' ;;
```

```
val length : 'a list -> int = <fun>
```

```
let rec somma lis =
    match lis with
     | [] -> 0
     | x::lis' -> x + somma lis' ;;
```

```
val somma : int list -> int = <fun>
```

```
let rec contains x lis =
    match lis with
     | [] -> false
     | y::lis' -> if y=x then true
                         else contains x lis'
```

```
val contains : 'a -> 'a list -> bool = <fun>
```

Possiamo implementare `@`:

```
let rec append l1 l2 =
    match l1 with
     [] -> l2
     | x :: l1' -> x :: (append l1' l2)
```

```
val append : 'a list -> 'a list -> 'a list = <fun>
```

possiamo utilizzare `@` anche per rovesciare una lista

```
let rec rev lis =
    match lis with
     | [] -> []
     | x::lis' -> (rev lis') @ [x] ;;
```

```
val rev : 'a list -> 'a list = <fun>
```

ma ciò è poco efficiente, infatti possiamo utilizzare un parametro `lis2` come accumulazione del risultato:

```
let rev lis =
    let rec rev_accum lis1 lis2 =
        match lis1 with
         | [] -> lis2
         | x::lis1' -> rev_accum lis1' (x::lis2)
    in
         rev_accum lis [] ;;
```

```
val rev : 'a list -> 'a list = <fun>
```

La funzione `rev_accum` ad ogni chiamata ricorsiva prende l'elemento in testa alla lista `lis1` e lo "sposta" in testa alla lista `lis2`.
Quindi gli elementi vengono presi uno dopo l'altro da
`lis1` e aggiunti uno prima dell'altro in `lis2`, con il risultato di rovesciare la lista.

Exist - funzione Higher-Order

Lo schema seguente di questa funzione può essere utilizzato per verificare una determinata condizione su un elemento di una lista

```
let rec exists p lis =
    match lis with
     | [] -> false
     | x::lis' -> if p x then true
                              else exists p lis' ;;
```

```
val exists : ('a -> bool) -> 'a list -> bool = <fun>
```

Le seguenti funzioni:

```
let rec contiene_zero lis =
    match lis with
    | [] -> false
    | x::lis' -> if x=0 then true
                              else contiene_zero lis' ;;
```

```
let rec contiene_positivo lis =
    match lis with
     | [] -> false
     | x::lis' -> if x>0 then true
                              else contiene_positivo lis' ;;
```

```
let rec contiene_pari lis =
    match lis with
     | [] -> false
     | x::lis' -> if x mod 2 = 0 then true
                              else contiene_pari lis' ;;
```

possono essere riscritte tramite schema
exist in questo modo:

```
let contiene_zero lis = exists (fun x -> x=0) lis ;;
let contiene_positivo lis = exists (fun x -> x>0) lis ;;
let contiene_pari lis = exists (fun x -> x mod 2 = 0) lis ;;
```

```
val contiene_zero : int list -> bool = <fun>
val contiene_positivo : int list -> bool = <fun>
val contiene_pari : int list -> bool = <fun>
```

Forall

Funzione higher-order che testa un predicato su tutti gli elementi della lista

```
let rec forall p lis =
    match lis with
     | [] -> true
     | x::lis' -> if p x then forall p lis'
                         else false ;;
```

```
val forall : ('a -> bool) -> 'a list -> bool = <fun>
```

Filter

Funzione higher-order in grado di selezionare (filtrare) gli elementi secondo una condizione restituendo la lista degli elementi che la soddisfano

```
let rec filter p lis =
    match lis with
     | [] -> []
     | x::lis' -> if p x then x::filter p lis'
                         else filter p lis' ;;
```

```
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Possiamo usare filter per definire le funzioni `estrai_zeri`, `estrai_positivi` e `estra_pari`, sempre usando la ricorsione in modo implicito.

```
let estrai_zeri lis = filter (fun x -> x=0) lis ;;
let estrai_positivi lis = filter (fun x -> x>0) lis ;;
let estrai_pari lis = filter (fun x -> x mod 2 = 0) lis ;;
```

```
val estrai_zeri : int list -> int list = <fun>
val estrai_positivi : int list -> int list = <fun>
val estrai_pari : int list -> int list = <fun>
```

Map

Funzione Higher-Order che permette di applicare la stessa operazione a tutti gli elementi ma a discapito del fatto che:

- produce una nuova lista con tanti elementi quanto quella processata
- il tipo degli elementi può essere diverso
- per astrarre sull'operazione abbiamo bisogno di una funzione, non di un predicato

```
let rec map f lis =
    match lis with
     | [] -> []
     | x::lis' -> f x::map f lis' ;;
```

```
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

esempi d'uso:

```
map (fun x -> x+1) [1;2;3] ;;
```

```
- : int list = [2; 3; 4]
```

```
let primo lis = map (fun (x,y) -> x) lis ;;
primo [ (3,2); (4,7); (9,2)] ;;
```

```
val primo : ('a * 'b) list -> 'a list = <fun>
```

```
- : int list = [3; 4; 9]
```

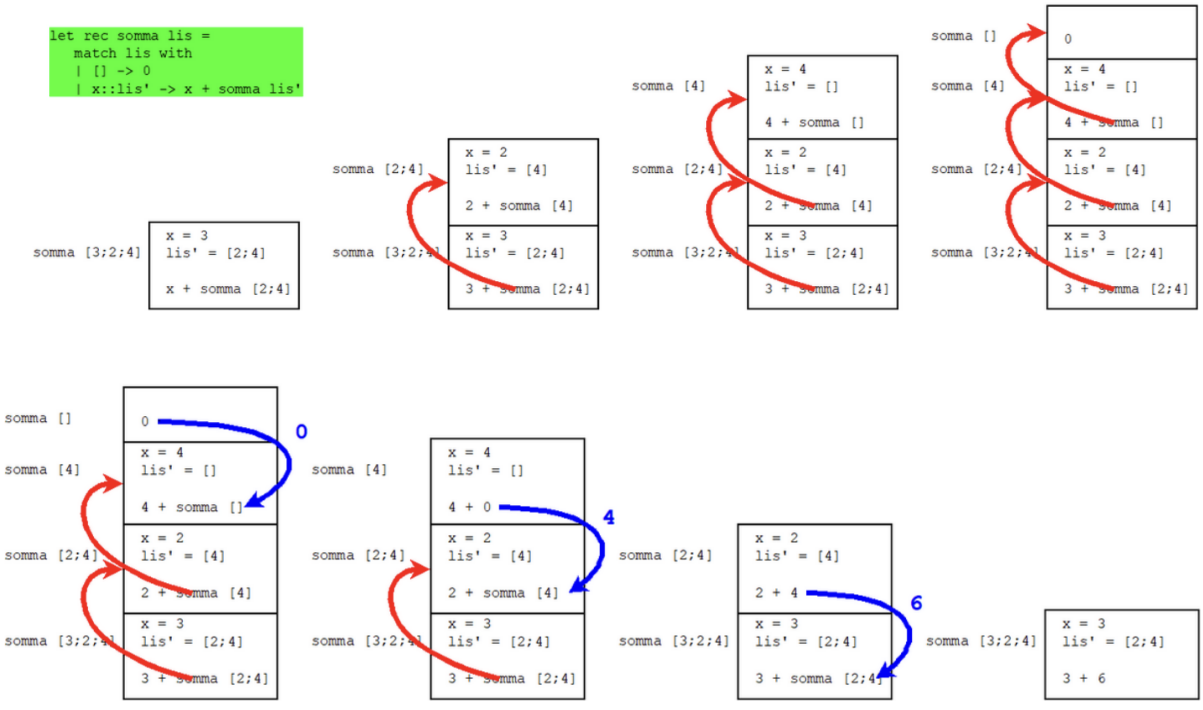
Fold-right

Funzione Higher-Order che permette di elaborare tutti gli elementi della lista per calcolare un unico risultato

```
let rec somma lis =
    match lis with
     | [] -> 0
     | x::lis' -> x + somma lis' ;;
somma [3;2;4] ;;
```

```
val somma : int list -> int = <fun>
- : int = 9
```

Vediamo cosa succede nello stack:



Grazie a questa funzione Higher-order è possibile trovare facilmente anche massimo e minimo in una lista avente come idea: estrarre il primo elemento e utilizzarlo come base

```
let minimo_massimo lis =
    let f x (min,max) =
        if x<min then (x,max)
        else if x>max then (min,x)
                 else (min,max)
    in
        match lis with
        | [] -> (0,0)
        | x::lis' -> fold_right f lis' (x,x) ;;

minimo_massimo [3;2;4] ;;
```

```
val minimo_massimo : int list -> int * int = <fun>
- : int * int = (2, 4)
```

Fold-left

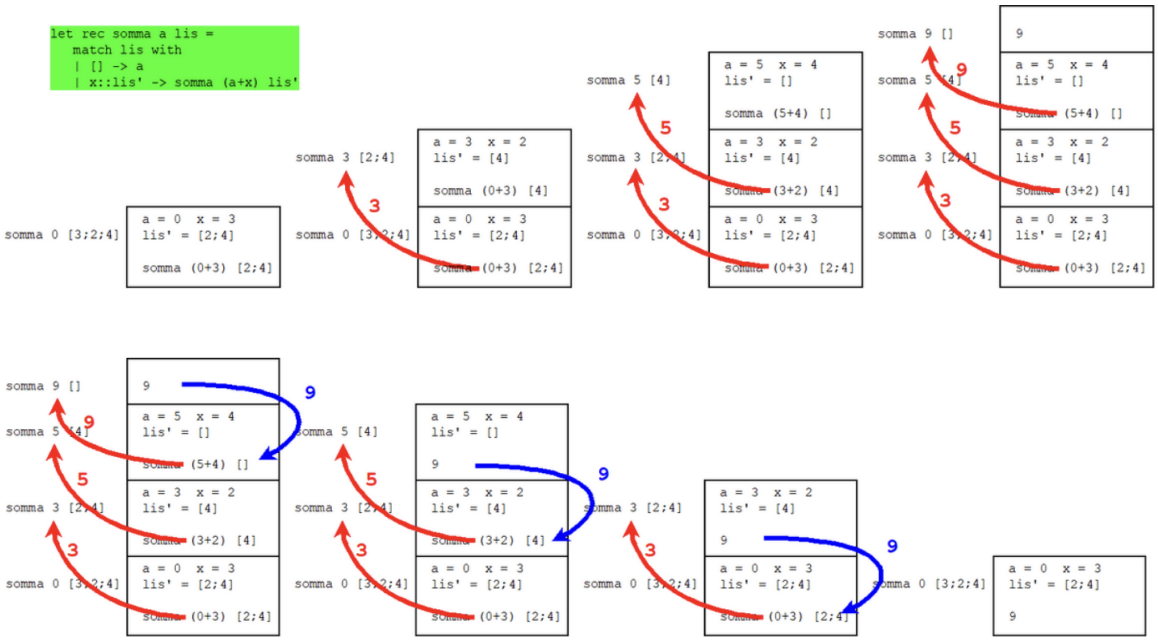
```
let rec somma a lis =
    match lis with
     | [] -> a
     | x::lis' -> somma (a+x) lis' ;;

somma 0 [3;2;4] ;;
```

```
val somma : int -> int list -> int = <fun>
- : int = 9
```

La caratteristica di questa funzione è che il risultato viene passato come parametro (inizialmente zero) e viene aggiornato man mano che si incontrano nuovi elementi scandendo la lista dall'inizio alla fine (da sinistra)

Vediamo che cosa succede nello stack:



Quindi:

- gli elementi della lista vengono sommati dal primo all'ultimo (da sinistra)
- ad ogni passo si applica la funzione/operatore (+) che somma due numeri
- il risultato "scende" immutato lungo lo stack (la funzione è tail recursive, quindi l'uso dello stack si può ottimizzare. . .)

```
let minimo_massimo lis =
    let f (min,max) x =
        if x<min then (x,max)
        else if x>max then (min,x)
                 else (min,max)
    in
        match lis with
         | [] -> (0,0)
         | x::lis' -> fold_left f (x,x) lis' ;;

minimo_massimo [3;2;4] ;;
```

```
val minimo_massimo : int list -> int * int = <fun>
- : int * int = (2, 4)
```