

Sequenze finite di elementi di uno stesso tipo:

```
# [1;2;3;4];;  
- : int list = [1; 2; 3; 4]  
# [true; false; 3>0 && 7<=0];;  
- : bool list = [true; false; false]  
# [(1,"pippo");(2,"pluto");(3,"topolino")];;  
- : (int * string) list =  
  [(1, "pippo"); (2, "pluto"); (3, "topolino")]
```

le liste non è un tipo, è un qualcosa che applicato al tipo degli elementi della lista mi costruisce il tipo delle liste con gli elementi di quel tipo. Questo operatore (lista) viene utilizzato in maniera post-fissa cioè dopo il suo argomento

- **list** è un **costruttore di tipi**: se **T** è un tipo, **T list** è il tipo delle liste con elementi di tipo **T**.

int list lista di int
bool list
(int * string) list lista di coppie int stringa
'a list

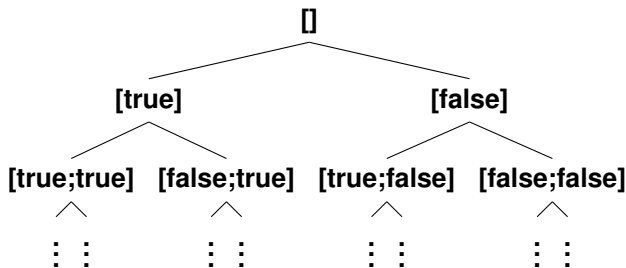
- La **lista vuota** è denotata da **[]**. È un valore polimorfo, di tipo **'a list**.
- L'operazione di **inserimento in testa** (**cons**) è denotata da **::**, infisso:
[1;2] = 1::[2] = 1::(2::[])

Definizione induttiva delle liste

induzione: riguarda la definizione di un insieme

- (i) La lista vuota, **[]**, è una α **list**;
- (ii) se **x** è di tipo α e **xs** è una α **list**, allora **(x::xs)** è una α **list**;
- (iii) nient'altro è una α **list**.

La definizione induttiva fornisce un metodo per generare tutte le liste di un dato tipo, per stadi:



Costruttori e selettori delle liste

I **COSTRUTTORI** del tipo **'a list** sono:

```
[] : 'a list  
(::) : 'a -> 'a list -> 'a list
```

```
[1;2;3] = 1 :: [2;3] = 1 :: (2 :: [3])  
        = 1 :: (2 :: (3 :: []))
```

I **SELETTORI** del tipo **'a list** sono:

```
List.hd : 'a list -> 'a  
List.tl : 'a list -> 'a list
```

Non sono definiti sulla lista vuota

```
# List.hd [1;2;3;4];;           # List.hd [];;  
- : int = 1                     Exception: Failure "hd".  
# List.tl [1;2;3;4];;          # List.tl [];;  
- : int list = [2; 3; 4]        Exception: Failure "tl".  
# List.hd(List.tl [1;2;3;4]);;  
- : int = 2
```

Definizione ricorsiva di operazioni sulle liste

Le liste si definiscono induttivamente, come i numeri naturali:

	numeri naturali	liste
base	0	[]
operazione per costruire oggetti “più complessi”	+1	::

Sulle liste si possono definire operazioni ricorsivamente:

- si definisce il valore dell'operazione per il caso base **[]**
- assumendo di saper calcolare il valore dell'operazione per una lista **rest** (ipotesi di lavoro), si determina come calcolarne il valore per la generica lista **x::rest** (di cui **rest** è la coda).

Esempio: Lunghezza di una lista

Problema: data una lista **lst** di tipo '**a list**', determinare il numero di elementi di **lst** (la lunghezza della lista).

length: 'a list -> int

Caso base: la lista è vuota. La sua lunghezza allora è 0.

Caso ricorsivo: assumiamo (ipotesi di lavoro) di saper calcolare la lunghezza della coda della lista data: sia n tale lunghezza.

La lunghezza della lista è allora $n + 1$, perché ha un elemento in più rispetto alla sua coda.

Procedimento:

```
length lst: se lst e' vuota, allora riportare 0
            altrimenti (* lst ha un elemento in piu'
                        rispetto alla sua coda *)
                    calcolare la lunghezza n della coda di lst
                    e riportare il valore n+1
```

La definizione è giustificata perché la coda della coda della coda ... della coda di qualsiasi lista è sempre la lista vuota. Quindi prima o poi si arriva al caso base.

Definizione di length

Possiamo usare il test “lista vuota” e il selettore List.tl:

```
(* length: 'a list -> int *)
let rec length lst =
  if lst=[] then 0
  else 1 + (length (List.tl lst))
```

Oppure utilizzando un'espressione **function** generale e il pattern matching:

```
(* length : 'a list -> int *)
let rec length = function
  [] -> 0
  | x::rest -> 1 + length rest
```

Pattern Matching con le liste (I)

```
(* length : 'a list -> int *)  
let rec length = function  
  [] -> 0  
  | x::rest -> 1 + length rest
```

Valutazione di **length [1;2;3;4]**:

- ❶ Il valore dell'argomento è **[1;2;3;4]**
- ❷ **[1;2;3;4]** non è conforme al pattern **[]** (non è la lista vuota), quindi andiamo avanti.
- ❸ **[1;2;3;4]** si può vedere come **1::[2;3;4]**, quindi è conforme al pattern **x::rest**.

Si aggiunge il legame provvisorio di **x** con **1** e di **rest** con **[2;3;4]**.
Con questo legame si calcola il valore di **1 + length rest** (cioè **1 + length [2;3;4]**), che viene riportato come valore di **length [1;2;3;4]**.
I legami provvisori vengono sciolti.

Pattern matching con le liste (II)

In un pattern possono occorrere solo variabili e costruttori:
i costruttori per le liste sono `[]` e `::`

Mediante pattern matching si possono distinguere il caso base e il caso ricorsivo di una definizione ricorsiva sulle liste:

- caso base: `[]` (lista vuota)
- caso ricorsivo: `x::rest` (lista con testa **x** e coda **rest**)

Il pattern matching costituisce un'alternativa all'uso di selettori: i legami determinati dal pattern matching servono a identificare testa e coda di una lista.

Alcuni pattern per le liste:

- `[]`: lista vuota
- `[x]`: lista con un solo elemento, **x**
- `[x;y]`: lista con esattamente due elementi
- `x::rest`: lista con almeno un elemento
- `x::y::rest`: lista con almeno due elementi (**x** è il primo, **y** il secondo, **rest** è la coda della coda).

Alcuni pattern per liste (I)

<i>pattern</i>	<i>espressione</i>		<i>successo e legami</i>
[]	[]		successo
[]	[1]		fallimento
[]	[1;2]		fallimento
[x]	[]		fallimento
[x]	[1]		x=1
[_]	[1;2]		fallimento
x::[]	[]		fallimento
x::[]	[1]	1::[]	x=1
x::[]	[1;2]	1::[2]	fallimento
x::y::[]	[]		fallimento
x::(y::[])	[1]	1::[]	fallimento
x::y::[]	[1;2]	1::2::[]	x=1, y=2
x::y	[]		fallimento
x::y	[1]	1::[]	x=1, y=[]
x::y	[1;2]	1::[2]	x=1, y=[2]
x::y	[1;2;3]	1::[2;3]	x=1, y=[2;3]

Alcuni pattern per liste (II)

<i>pattern</i>	<i>espressione</i>		<i>successo e legami</i>
x::rest	[]		fallimento
x::rest	[1]	1::[]	x=1, rest=[]
x::rest	[1;2]	1::[2]	x=1, rest=[2]
x::rest	[1;2;3]	1::[2;3]	x=1, rest=[2;3]
x::y::rest	[]		fallimento
x::y::rest	[1]	1::[]	fallimento
x::y::rest	[1;2]	1::2::[]	x=1, y=2, rest=[]
x::y::rest	[1;2;3]	1::2::[3]	x=1, y=2, rest=[3]
x::y::rest	[1;2;3;4;5]	1::2::[3;4;5]	x=1, y=2, rest=[3;4;5]

La schedina del superenalotto

Problema: Date le ultime X estrazioni del superenalotto (sequenze di 6 numeri compresi tra 1 e 90), determinare i 6 numeri che più probabilmente usciranno alla prossima estrazione.

In generale: data una lista contenente liste di interi **estrazioni: int list list**, dove ogni sottolista contiene DIM elementi, compresi tra 1 e HIGHER, determinare i DIM numeri che occorrono nella lista un minor numero di volte.

Problema principale:

super: int list list -> int -> int -> int list

super estrazioni dim higher:

estrazioni è una lista di liste rappresentante le ultime estrazioni,

dim è il numero di interi di ogni estrazione

higher è il massimo numero che può essere estratto

Riporta i **dim** numeri che sono stati estratti meno volte

La schedina del superenalotto: sottoproblemi (I)

- 1) contare, per ogni numero **n** da 1 a **higher**, quante volte **n** occorre in estrazioni.
 - 1a) costruire la lista **[1;2;...;higher]**
Sottoproblema **upto: int -> int -> int list**
upto n m = [n;n+1;...;m]
 - 1b) “appiattare” la lista **estrazioni**, trasformandola in una lista di interi
Sottoproblema **flatten: 'a list list -> 'a list**
 - 1c) “contare” le occorrenze di ciascun elemento di **[1;2;...;higher]** nella lista **flatten estrazioni**:
Sottoproblema **contatutti : 'a list -> 'a list -> ('a * int) list**
contatutti elementi listona = lista di coppie (**ele,n**), dove **ele** è un elemento di **elementi** e **n** è il numero di occorrenze di **ele** in **listona**.
 - 1c-1) contare le occorrenze di un elemento in una lista
Sottoproblema **conta : 'a -> 'a list -> int**
conta n lista = numero di occorrenze di **n** in **lista**

La schedina del superenalotto: sottoproblemi (II)

- 2) ordinare la lista di coppie

contatutti (upto 1 higher) (flatten estrazioni)

secondo valori non decrescenti del secondo elemento

Sottoproblema **sort: ('a * 'b) list -> ('a * 'b) list**

- 3) prendere le prime **dim** coppie della lista ordinata

sort (contatutti (upto 1 higher) (flatten estrazioni))

Sottoproblema **take: int -> 'a list -> 'a list**

take n lista = primi **n** elementi di **lista** (o la lista intera se non ce ne sono abbastanza)

- 4) dalla lista di **dim** coppie ottenuta al punto 3:

take dim (sort (contatutti (upto 1 higher) (flatten estrazioni)))

estrarre la lista con i primi elementi di ciascuna coppia:

Sottoproblema **primi: ('a * 'b) list -> 'a list**

primi [(x1,y1);...;(xn,yn)] = [x1;...;xn]

La schedina del superenalotto: soluzione

[Vai al codice](#)

Merge Sort (ordinamento per fusione)

Algoritmo: Sia C l'insieme (lista) che si vuole ordinare
Si divide l'insieme C in due parti pressappoco uguali.
Ciascuno dei due sottoinsiemi viene ordinato, ricorsivamente.
Si applica l'algoritmo di **fusione** alle liste risultanti dalle chiamate ricorsive

Casi base: lista vuota o di un solo elemento

Caso ricorsivo: per ordinare **lst** di lunghezza > 1 :

- Si suddivide la lista in due parti di dimensioni pressappoco uguali (± 1): siano **xs** e **ys** le due liste ottenute
- Il risultato è la fusione di **(mergesort xs)** e **(mergesort ys)**:
merge (mergesort xs) (mergesort ys)

Vai al codice

<http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html>

- **List.hd** : 'a list -> 'a
- **List.tl** : 'a list -> 'a list
- **List.length** : 'a list -> int
- **List.flatten** : 'a list list -> 'a list
- **List.sort** : ('a -> 'a -> int) -> 'a list -> 'a list

Sort a list in increasing order according to a comparison function. The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller. For example, **compare** is a suitable comparison function.

compare: 'a -> 'a -> int

Vai al codice

Implementazione di procedimenti iterativi: length

Versione ricorsiva

```
(* length : 'a list -> int *)  
let rec length = function  
  [] -> 0  
  | x::rest -> 1 + length rest;;
```

Versione iterativa

```
let len lst =  
  (* aux: int -> 'a list -> int *)  
  (* aux n lst = *)  
  let rec aux result = function  
    [] -> result  
    | _::rest -> aux (result+1) rest  
  in aux 0 lst
```

```
len [1;2;3] = aux 0 [10;20;30] = aux 1 [20;30]  
            = aux 2 [30] = aux 3 [] = 3
```

Specifica dichiarativa di aux?

Implementazione di procedimenti iterativi: length

Versione ricorsiva

```
(* length : 'a list -> int *)  
let rec length = function  
  [] -> 0  
  | x::rest -> 1 + length rest;;
```

Versione iterativa

```
let len lst =  
  (* aux: int -> 'a list -> int *)  
  (* aux n lst = n + lunghezza di lst *)  
  let rec aux result = function  
    [] -> result  
    | _::rest -> aux (result+1) rest  
  in aux 0 lst
```

```
len [1;2;3] = aux 0 [10;20;30] = aux 1 [20;30]  
            = aux 2 [30] = aux 3 [] = 3
```

Specifica dichiarativa di aux?

Calcolo del prodotto degli interi in una lista

Versione ricorsiva:

```
(* prodof : int list -> int *)  
let rec prodof = function  
  [] -> 1  
  | x::xs -> x * prodof xs
```

Versione iterativa

```
let prodof_it lst =  
  (* aux: int -> int list -> int  
    aux n lista = *)  
  let rec aux result = function  
    [] -> result  
    | x::rest -> aux (x*result) rest  
  in aux 1 lst
```

L'accumulatore viene inizializzato con l'elemento neutro per il prodotto

Specifica dichiarativa di aux?

Calcolo del prodotto degli interi in una lista

Versione ricorsiva:

```
(* prodof : int list -> int *)  
let rec prodof = function  
  [] -> 1  
  | x::xs -> x * prodof xs
```

Versione iterativa

```
let prodof_it lst =  
  (* aux: int -> int list -> int  
    aux n lista = n * prodotto degli interi in lista *)  
  let rec aux result = function  
    [] -> result  
    | x::rest -> aux (x*result) rest  
  in aux 1 lst
```

L'accumulatore viene inizializzato con l'elemento neutro per il prodotto

Specifica dichiarativa di aux?

Versione iterativa di upto (I)

```
(* upto : int -> int -> int list *)  
let rec upto m n =  
  if m > n then []  
  else m :: upto (m+1) n
```

Versione iterativa?

```
let upto_it m n =  
  let rec aux result m n =  
    if m > n then result  
    else aux (m::result) (m+1) n  
  in aux [] m n
```

`aux [] 3 5 = aux [3] 4 5 = aux [4;3] 5 5 = ...`

È un **downto** !

Attenzione quando le operazioni che si fanno sull'accumulatore non commutano.

Versione iterativa di upto (II)

Versione iterativa corretta di **upto**: anziché incrementare il limite inferiore, decrementiamo quello superiore:

```
(* upto_it : int -> int -> int list *)
let upto_it m n =
  (* aux: int list -> int -> int -> int list
    aux lista m n = *)
  let rec aux result m n =
    if m > n then result
    else aux (n::result) m (n-1)
  in aux [] m n

upto_it 3 5 = aux [] 3 5
            = if 3>5 then [] else aux [5] 3 4
            = aux [5] 3 4
            = aux [4;5] 3 3
            = aux [3;4;5] 3 2 = [3;4;5]
```

Specifica dichiarativa di aux?

Versione iterativa di upto (II)

Versione iterativa corretta di **upto**: anziché incrementare il limite inferiore, decrementiamo quello superiore:

```
(* upto_it : int -> int -> int list *)
let upto_it m n =
  (* aux: int list -> int -> int -> int list
     aux lista m n = [m;m+1;...;n] @ lista *) lista da m a n concatenata con lista
  let rec aux result m n =
    if m > n then result
    else aux (n::result) m (n-1)
  in aux [] m n
```

```
upto_it 3 5 = aux [] 3 5
            = if 3>5 then [] else aux [5] 3 4
            = aux [5] 3 4
            = aux [4;5] 3 3
            = aux [3;4;5] 3 2 = [3;4;5]
```

Specifica dichiarativa di aux?

Take: versione iterativa?

```
(* take : int -> 'a list -> 'a list *)
let rec take n = function
  [] -> []
| x::xs -> if n<=0 then [] else
            x::take (n-1) xs;;
```

Versione iterativa?

```
take rovesciato
let rtake n lst =
  let rec aux result n = function
    [] -> result
  | x::rest -> if n<=0 then result else
               aux (x::result) (n-1) rest
  in aux [] n lst

# rtake 4 [1;2;3;4;5;6];;
- : int list = [4; 3; 2; 1]
```

Il risultato è rovesciato!

Versione iterativa di take con inserimento in coda

```
let take_it n lst =  
  (* aux : 'a list -> int -> 'a list -> 'a list  
    aux result n lst = *)  
  let rec aux result n = function  
    [] -> result  
    | x::rest -> if n<=0 then result else  
                  aux (result@[x]) (n-1) rest  
  in aux [] n lst
```

Specifica dichiarativa di aux?

Versione iterativa di take con inserimento in coda

```
let take_it n lst =  
  (* aux : 'a list -> int -> 'a list -> 'a list  
    aux result n lst = result @ (take n lst) *)  
  let rec aux result n = function  
    [] -> result  
    | x::rest -> if n<=0 then result else  
                  aux (result@[x]) (n-1) rest  
  in aux [] n lst
```

Specifica dichiarativa di aux?

Ma la concatenazione è un'operazione costosa: conviene inserire gli elementi in testa e rovesciare il risultato alla fine

Rovesciare una lista

Versione ricorsiva

```
(* reverse: 'a list -> 'a list *)
let rec reverse = function
  [] -> []
  | x::xs -> (reverse xs) @ [x];;
```

Versione iterativa

```
(* rev: 'a list -> 'a list *)
let rev lst =
  (* revto : 'a list -> 'a list -> 'a list *)
  (* revto result lst = *)
  let rec revto result = function
    [] -> result
    | x::rest -> revto (x::result) rest
  in revto [] lst

rev [1;2;3] = revto [] [1;2;3] = revto [1] [2;3]
           = revto [2;1] [3] = revto [3;2;1] [] = [3;2;1]
```

Specifica dichiarativa di revto?

Rovesciare una lista

Versione ricorsiva

```
(* reverse: 'a list -> 'a list *)  
let rec reverse = function  
  [] -> []  
  | x::xs -> (reverse xs) @ [x];;
```

Versione iterativa

```
(* rev: 'a list -> 'a list *)  
let rev lst =  
  (* revto : 'a list -> 'a list -> 'a list *)  
  (* revto result lst = (reverse lst) @ result *)  
  let rec revto result = function  
    [] -> result  
    | x::rest -> revto (x::result) rest  
  in revto [] lst  
  
rev [1;2;3] = revto [] [1;2;3] = revto [1] [2;3]  
           = revto [2;1] [3] = revto [3;2;1] [] = [3;2;1]
```

Specifica dichiarativa di revto?

Versione iterativa di take con il reverse

```
let take_it n lst =  
  (* aux : 'a list -> int -> 'a list -> 'a list *)  
  (* aux result n list = *)  
  let rec aux result n = function  
    [] -> List.rev result  
    | x::rest -> if n<=0 then List.rev result else  
                  aux (x::result) (n-1) rest  
  in aux [] n lst
```

Specifica dichiarativa di aux?

Versione iterativa di take con il reverse

```
let take_it n lst =  
  (* aux : 'a list -> int -> 'a list -> 'a list *)  
  (* aux result n list = (rev result) @ (take n lst) *)  
  let rec aux result n = function  
    [] -> List.rev result  
    | x::rest -> if n<=0 then List.rev result else  
                  aux (x::result) (n-1) rest  
  in aux [] n lst
```

result rovesciata concatenato con take

Specifica dichiarativa di aux?