Funzioni di ordine superiore

Prendono come argomento o riportano come valore una funzione

Il tipo di una funzione di ordine superiore ha più di una “freccia”

```
let rec sum f lower upper =
  if lower > upper then 0
  else f lower + sum f (lower +1) upper
```

Tipo di **sum: (int -> int) -> (int -> (int -> int))**

```
(* square : int -> int *)
let square x = x*x
```

sum square: **int -> int -> int**
sum square 3: **int -> int**
sum square 3 5: **int**

sum square 3 5 $= \sum_{k=3}^{5} k^2$

```
(* sumsquare : int -> int -> int *)
let sumsquare = sum square
```

Funzioni di ordine superiore sulle liste

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

Abbiamo già incontrato
List.sort: ('a -> 'a -> int) -> 'a list -> 'a list
List.iter: ('a -> unit) -> 'a list -> unit

Iterators
List.map: ('a -> 'b) -> 'a list -> 'b list

List scanning
List.for_all: ('a -> bool) -> 'a list -> bool
List.exists: ('a -> bool) -> 'a list -> bool

List searching
List.find : ('a -> bool) -> 'a list -> 'a
List.filter: ('a -> bool) -> 'a list -> 'a list

Un esempio di applicazione di map

inits: **'a list -> 'a list list**

inits lst = lista con tutti i segmenti iniziali di **lst**

Esempio: inits [1;2;3;4] = [[1];[1;2];[1;2;3];[1;2;3;4]]

Casi semplici: **inits [ ]** =
inits [x] =

Un esempio di applicazione di map

inits: **'a list -> 'a list list**

inits lst = lista con tutti i segmenti iniziali di **lst**

Esempio: inits [1;2;3;4] = [[1];[1;2];[1;2;3];[1;2;3;4]]

Casi semplici: **inits [ ] = [ ]**
inits [x] = [[x]]

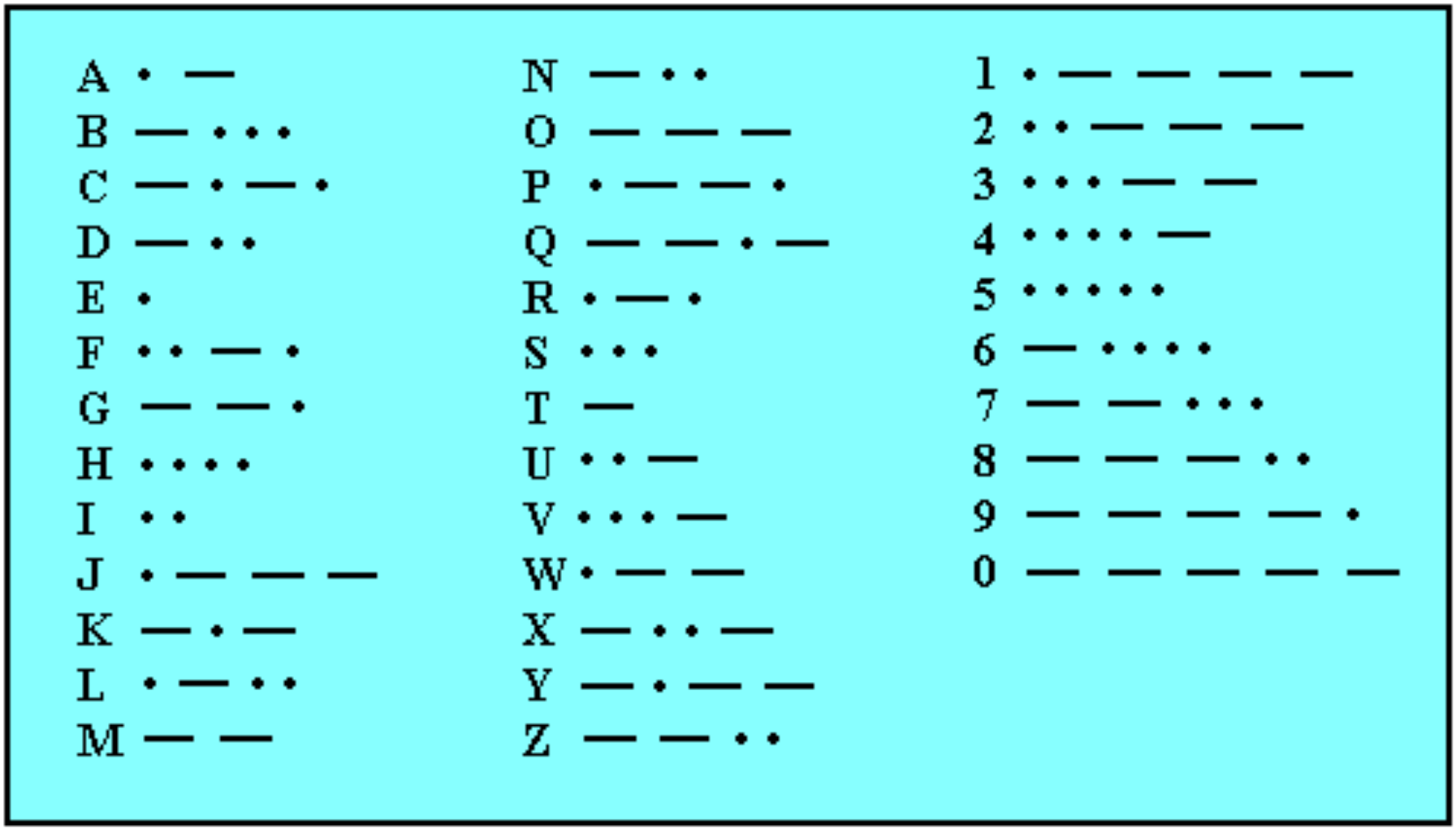
Caso generale: iniziamo a ragionare su un esempio:

```
inits [1;2;3;4]:
    sia L = inits [2;3;4] = [[2];[2;3];[2;3;4]]
    si deve applicare l'operazione "inserire 1 in testa a"
       a tutti gli elementi di L, e aggiungere [1]

inits (x::rest) =
     se L = inits rest = [s1;...;sk]
     ==>  [x] :: [x::s1; ....;x::sk]
```

Vai al codice

Il codice morse



Rappresentazione di insiemi finiti: insieme delle parti

powerset: 'a list -> 'a list list
applicata a una lista L rappresentante un insieme S, riporta una lista con tutti i sottoinsiemi di S.

- Caso base: **L = [ ]** $\Rightarrow$ **powerset L** =

Rappresentazione di insiemi finiti: insieme delle parti

powerset: 'a list -> 'a list list
applicata a una lista *L* rappresentante un insieme *S*, riporta una lista con tutti i sottoinsiemi di *S*.

- Caso base: **L = [ ]** $\Rightarrow$ **powerset L** =**[[ ]]**
- Lista non vuota, ragioniamo su un esempio: **L = [1;2;3]**.

  Per ipotesi (della ricorsione) si sa calcolare

  powerset [2;3] = [[ ];[2];[3];[2;3]].

  powerset [1;2;3] = [[ ];[2];[3];[2;3]] @ [[1];[1;2];[1;3];[1;2;3]]
  = [[ ];[2];[3];[2;3]] @ [1::[ ]; 1::[2]; 1::[3]; 1::[2;3]]
- Caso generale: **L = x::rest**

  se **[S1;...;Sk]** sono tutti i sottoinsiemi di **rest**:

  powerset L = **[S1;...;Sk] @ [x::S1;...;x::Sk]**

Soluzione

Sottoproblema: operazione di inserimento di un elemento in testa a tutti gli elementi di una lista di liste:

```
(* cons : 'a -> 'a list -> 'a list *)
let cons x rest = x::rest;;

List.map (cons x): 'a list list -> 'a list list

List.map (cons x) [lst1;lst2;...;lstn]
    = [x::lst1; x::lst2;...; x::lstn]
```

Problema principale:

powerset [ ]	$\Rightarrow$	[[ ]]
powerset (x::rest)	$\Rightarrow$	(powerset rest)
		@ (List.map (cons x) (powerset rest))

Vai al codice

Prodotto cartesiano di due insiemi

cartprod: 'a list -> 'b list -> ('a * 'b) list
applicata a due insiemi (liste) **setA** e **setB**, riporta la lista di tutte le coppie **(x,y)**, con **x** $\in$ **setA** e **y** $\in$ **setB**.

Ricorsione sulla prima lista

- Caso base: **cartprod [ ] setB** =

Prodotto cartesiano di due insiemi

cartprod: 'a list -> 'b list -> ('a * 'b) list
applicata a due insiemi (liste) **setA** e **setB**, riporta la lista di tutte le coppie **(x,y)**, con **x** $\in$ **setA** e **y** $\in$ **setB**.

Ricorsione sulla prima lista

- Caso base: **cartprod [ ] setB** = **[ ]**
- Caso generale: **setA = x::rest**

  Per ipotesi (della ricorsione), si sa calcolare **cartprod rest setB**
  Si devono aggiungere a questa lista tutte le coppie **(x,y)**, con **y** $\in$ **setB**.

  Sottoproblema: dato **x** e una lista **lst=[y1;...;yn]**, costruire la lista **[(x,y1);...;(x,yn)]**.

```
let pair x y = (x,y)
   pair: 'a -> 'b -> ('a * 'b)
   pair x:      'b -> ('a * 'b)
   List.map (pair x): 'b list -> ('a * 'b) list
```

cartprod(x::rest,set) $\Rightarrow$ (List.map (pair x) set) @ cartprod(rest,set)

Vai al codice

Altre utili funzioni del modulo List

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

List.rev_map : ('a -> 'b) -> 'a list -> 'b list

List.rev_map f lst = List.rev (List.map f lst)

List.find: ('a -> bool) -> 'a list -> 'a

List.partition: ('a -> bool) -> 'a list -> 'a list * 'a list

List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

List.fold_left f a [b1; ...; bn] = f (... (f (f a b1) b2) ...) bn.

```
(* sumof : int list -> int *)
(* sumof [x1;...;xk] = x1+...+xk *)
let sumof = List.fold_left (+) 0
```

List.fold_right : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

List.fold_right f [a1; ...; an] b = f a1 (f a2 (... (f an b) ...)).