Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

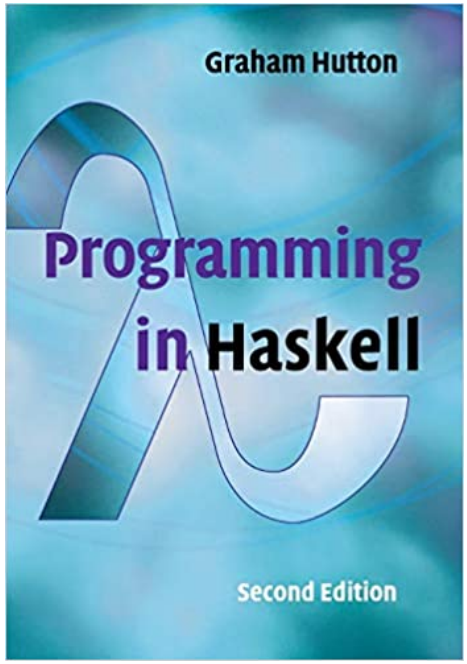
Ideologia Funzionale in Azione: Comporre, Comporre, Comporre

Corso di Laurea in **Informatica**, III anno

SAPIENZA
UNIVERSITÀ DI ROMA

Lezione **6**, 11 marzo 2022

tratta da:



Lezione 6a:
Voting Sistems

Voting System: problema e analisi

Abbiamo una lista **`votes`** che raccoglie tutte le **preferenze**.
Vogliamo determinare il **candidato** ha avuto **più preferenze**.

Idea Funzionale:

1. **comporre trasformazioni** fino a ottenere una lista **`results`** contenente **coppie nella forma** (*np*, *c*), dove *np* è il numero di preferenze e *c* è il candidato.
2. **ordinare** la lista: la relazione d'ordine sulle tuple è l'ordine lessicografico, quindi ordinerà per numero di preferenze (e per questo l'abbiamo messo nella prima componente)
3. si prende l'**ultimo** elemento (usiamo sort standard cresc.)
4. si **estrae** la seconda componente, cioè il candidato.

```
votes :: [String]
votes = [“Rossi”, “Bianchi”, “Verdi”, “Bianchi”,
“Bianchi”, “Rossi”]

results = [(2,”Rossi”),(3, “Bianchi”),(1,”Verdi”)]

winner :: Ord a => [a] -> a
winner = snd . last . sort . results
```

Voting System: results

Ora occorre produrre la lista **`results`.** Seguiamo un approccio simile e pensiamo a sequenze di trasformazioni di **`votes`**.

Idea Funzionale:

1. ottenere la **la lista `cdts` dei candidati** rimuovendo i duplicati;
2. conteggiare i voti di ciascun candidato (un po' inefficiente ma si può fare semplicemente componendo **`filter`** e **`length`**)
3. accoppiare le due liste.

Chiudiamo mostrando **`rmvDups`**.

```
results :: Eq a => [a] -> [(Int, a)]
results xs =
     zip
        (map (\x -> count x xs) cdts) cdts where
             count x = length . filter (==x)
             cdts = rmvDups xs

rmvDups :: Eq a => [a] -> [a]
rmvDups (x:xs) = x : (filter (/=x) (rmvDups xs))
rmvDups   []   = []
```

Morale della fiaba

È chiaramente una forma di metodologia **top-down**.

Porta a un **rapid-prototyping**, pensando alle **trasformazioni** di una struttura **dati nel suo complesso**, che generano un flusso, che passa diverse funzioni per trasformare i dati nel risultato.

All'occorrenza, può essere necessario **raffinare** qualche funzione (ad esempio, vedremo in una successiva applicazione un modo più efficiente di contare le occorrenze).

Infine, è sempre istruttivo vedere delle alternative. Questa è la versione di Graham Hutton, in cui l'accoppiamento viene fatto per **list-comprehension.**

```
results :: Ord a => [a] -> [(Int, a)]
results vs = sort [(count v vs, v) | v <- rmvDups vs] where
             count x = length . filter (==x)
```

Un grande classico della propaganda

Questo è un programma da **libro di scuola**, che mostra quanto sia facile programmare **quicksort** in Haskell.

Ma si tratterà di **verità** o **propaganda**?

Qual è il punto di forza di Quicksort (rispetto a Mergesort)?

Non allocare memoria e l'**efficienza** della procedura **partiziona** (che poi permette di `ricombinare' i risultati in $\Theta(1)$).

Ma qui? qual è la **complessità di ++**?

Inoltre: **quanta memoria** viene allocata per eseguire questa funzione?

```
 -- in Haskell quickSort si scrive in 4 righe…
qSort   []   = []
qSort (x:xs) = qSort smaller ++ [x] ++ qSort larger
   where
      smaller   = [a | a <- xs, a <=x ]
      larger = [b | b <- xs, b>x ]
```

Voting System v2 (1)

Problema: Consideriamo un sistema elettorale più complesso.

La lista di voti è composta non più da singole preferenza, ma da **liste di preferenze** in cui ciascuno **mette in ordine** i candidati.

Il vincitore si ottiene **eliminando iterativamente quelli che ricevono meno prime scelte**.

Nel nostro esempio, il primo a essere eliminato è `"Rossi"` e poi viene rimosso… finché non resta un solo candidato.

```
ballots :: [[String]]
votes = [[“Rossi”, “Verdi”], [“Bianchi”],
             [“Verdi”, “Rossi”, “Bianchi”],
             [“Bianchi”, “Verdi”, “Rossi”], [“Verdi”]]
-- si elimina Rossi
votes’ = [[“Verdi”], [“Bianchi”], [“Verdi”, “Bianchi”],
             [“Bianchi”, “Verdi”], [“Verdi”]]
-- si elimina Bianchi
votes’’=[[“Verdi”],[],[“Verdi”], [“Verdi”],[“Verdi”]]
```

Voting System v2 (2)

Possiamo sfruttare parte del lavoro precedente per scrivere una funzione **rank** che ordina i candidati.

È sufficiente **riusare** la funzione **results** scritta prima, sulle **prime preferenze** facilmente ottenibili con **map head**.

Eliminiamo il candidato ed eventuali ballot rimasti vuoti (ricordate che dobbiamo mappare **head**).

E infine ricomporre i pezzi.

```
rank :: Ord a => [[a]] -> [a]
rank = map snd . results . map head

elim :: Eq a => a -> [[a]] -> [[a]]
elim x = map (filter (/=x))

rmvEmpty = filter (/=[])

winnerB :: Ord a => [[a]] -> a
winnerB bs = let (c:cs) = rank (rmvEmpty bs)
   in if cs == [] then c -- ho finito
                  else winnerB (elim c bs)
```

my Voting System (1)

Si può essere anche più composizionali, scoprendo che in Haskell si può fare **iterazione** (e questo **è** un processo iterativo).

Idea: si itera l'eliminazione dai ballots finché non resta un solo candidato. In Haskell posso **mimare l'iterazione** con opportuni funzionali.

Ad esempio è predefinito il funzionale **`until p f x`**, che calcola la sequenza $x, f\,x, f\,(f\,x)$, … fino a che non vale $p\,(f\,(f\,(\ldots(f\,x)\ldots)))$ nel qual caso ritorna appunto il valore $f\,(f\,(\ldots(f\,x)\ldots))$.

A questo punto possiamo già scrivere il programma ☺

```
until :: (a -> Bool) -> (a -> a) -> a -> a
until p f a = if p a then a else until p f (f a)

-- il programma è più esplicativo della
-- sua stessa descrizione a parole ☺
winnerB' = until justOne elimLast
```

my Voting System (2)

Verificare che i ballots contengano solo un candidato, implica verificare che tutte le liste di preferenze rimaste contengono lo stesso candidato. Il che può essere computazionalmente gravoso (almeno per rispondere `True`) e anche un po' noioso da programmare (**Esercizio**).

Ho preferito scrivere una funzione **noOne** che si limita a verificare se la lista rimasta sia vuota o meno. Ciò però provoca un altro problema: mi serve ricordare l'**ultimo candidato eliminato**.

Posso sempre tenere informazione su una coppia e faccio un'iterazione della trasformazione $(e, b) \rightarrow (e', b')$ dove e traccia l'ultimo candidato eliminato e b i ballots rimasti.

Ma chi è il **primo candidato eliminato**, prima di cominciare? ☺

```
-- il programma è più esplicativo della
-- sua stessa descrizione a parole ☺
winnerB'' = fst . until noOne voteRound

winnerS = \xs -> winnerB'' (undefined, xs)
```

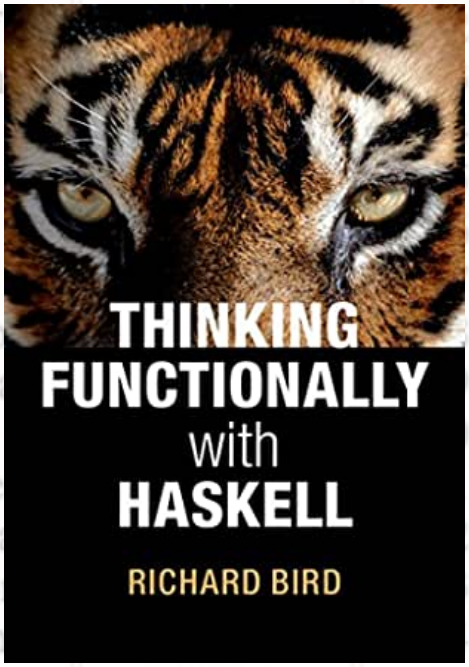
my Voting System (3)

Ovviamente tutte le funzioni si esprimono facilmente in funzione di quelle già scritte. Vediamo.

Molto istruttivo il tipo di **voteRound**: in particolare quel t tutto soletto…

```
-- piccole complicazioni dovute alle coppie
looserRound = snd . head . results . map head

-- occorre putroppo accoppiare/disaccoppiare
-- informazione
voteRound =
  \(x, xs) -> let e = looserRound xs
                  in (e, rmvEmpty (elim e xs))

voteRound :: (t, [[String]]) -> (String, [[String]])

-- basta verificare lista dei ballots vuota
noOne = null . snd

winnerB'' = fst . until noOne voteRound

-- occorre mantenere l'interfaccia richiesta
winnerS = \xs -> winnerB' (undefined, xs)
```

tratta da:



Lezione 6b:
Common Words

commonWords: problema & specifiche

Prendendo un testo, l'obiettivo è produrre la **stampa ordinata** (per frequenza) delle ***n* parole** che occorrono **più spesso**, insieme alle loro stesse frequenze.

Dobbiamo scrivere una funzione `commonWords` (nel riquadro il **tipo** e l'**output** atteso).

Tale output si ottiene caricando una stringa nella forma (**'\n'** è il carattere **return**):

```
" the : 321\n a : 122\n in : 81\n of : 50\n and : 37"
```

Ci sono molte vie, cerchiamone una **genuinamente funzionale**! Prima di cominciare **definiamo tipi** per **schiarire le specifiche**!

```
 -- specifica con tipi
commonWords :: Int -> [Char] -> [Char]
```

```
 -- I sinonimi aiutano a vivere meglio
type Text = [Char]
Type Word = [Char]
commonWords :: Int -> Text -> String
```

```
-- output
the : 321
a : 122
in : 81
of : 50
and : 37
```

Analisi del problema

Procederemo, come prima cercando di individuare le macro-operazioni da comporre per ottenere il risultato finale.

1. Il primo problema è dato il testo, **individuare le parole**.
2. Poi occorre **contarle**, magari producendo una lista di coppie nella forma (*f*, *p*) dove *p* è la parola ed *f* la sua frequenza…
3. … in modo da poterle facilmente **ordinare** per frequenza
4. **prendiamo le prime** *n* parole della lista ordinata…
5. e infine produciamo un **output adeguato** alle specifiche.

```
-- specifica Haskell di commonWords:
commonWords :: Int -> Text -> String

commonWords n =    -- Text -> String
   makeOutput .    -- [(Int, Word)] -> String
      take n .     -- [(Int, Word)] -> [(Int, Word)]
         sort .    -- [(Int, Word)] -> [(Int, Word)]
            countFreqs . -- [Word] -> [(Int, Word)]
              findwords          -- Text -> [Word]
```

Trovare le parole

Cos'è una parola? Noi considereremo una parola come una **sequenza massimale** di caratteri **alfabetici** circondati da spazi.

C'è la funzione di Prelude **`words`** che segmenta una stringa in parole circondate da spazi. Putroppo mantiene 'attaccati' ad esempio i **segni di punteggiatura**.

Infine uno vorrebbe essere case insensitive in questo problema e considerare "**un**" e "**Un**" come essere la stessa parola.

Ci sarebbero altre funzioni di libreria che ci aiutano in un modulo **`Data.Text`**. Ma noi faremo da soli.

```
-- Nel prelude è definita words:
words :: String -> [String]

> words "Un mattino, al risveglio da sogni inquieti,
Gregor Samsa si trovò trasformato in un enorme insetto."
["Un","mattino,","al","risveglio","da","sogni","inquieti,"
,"Gregor","Samsa","si","trovò","trasformato","in","un",
"enorme","insetto."]
```

Sistemiamo le parole

Vediamo come isolare le parole solo con lettere minuscole: lo facciamo in **un'unica passata** per **efficienza**, ma avremo potuto scrivere due funzioni: **`onlyAlpha`** e **`toLower`** e comporle.

Idea: costruiamo una tabella `ts=[(a, A),(b, B),…,(z,Z)]`.

Prendiamo un carattere alla volta e vediamo se è uguale alla prima o seconda componente di della coppia `(lower, upper)`…

e in caso affermativo restituiamo la **minuscola**.

Questa sarà il primo elemento della lista risultato e poi si continua ricorsivamente **fino alla parola vuota**.

Di conseguenza, **`findWords = toLowerAlpha . words`**

```
--tabella traduz. lower/upper, sfrutto le enumerazioni
ts = zip ['a'..'z'] ['A'..'Z']

toLowerAlpha :: Word -> Word
toLowerAlpha (c:cs) =
   if null ws then rs else fst (head ws):rs where
      ws = filter (\(l,u)-> l == c || u == c) ts
      rs = toLowerAlpha cs          -- chiamata ricorsiva
toLowerAlpha [] =[]
```

Conteggio

Abbiamo visto **Voting System** un facile one-liner funzione **`results`** per il conteggio dei voti. Guarda caso i tipi tornano (**`results`** è generica sugli oggetti che conta, richiede solo **`Eq a`**).

Chiaramente si tratta di una procedura di complessità $\theta(n^2)$. Non è una buona idea se si vuole contare le parole più frequenti nella Bibbia o nell'Ulysses di Joyce… ma tant'è.

Ma è **perfetta per testing**… quindi:

```
countFreqs = results
```

```
p = "Sopra la panca, la capra canta, sotto la panca la capra crepa"
w = words p
["Sopra","la","panca","la","capra","crepa,","sotto","la","panca",
"la","capra","canta."]
wl = map toLowerAlpha w
> wl
["sopra","la","panca","la","capra","crepa","sotto","la","panca",
"la","capra","canta"]
f = countFreqs wl
> f
[(1,"canta"),(2,"capra"),(1,"crepa"),(4,"la"),(2,"panca"),
(1,"sopra"),(1,"sotto")]
```

Ordinamento

Ora abbiamo la tabella di frequenze: **`quickSort`** ordina in ordine ascendente (sulle frequenze, prima componente).

Si può ovviamente scrivere un'altra funzione, che usa la relazione di ≥ invece di ≤, tuttavia…

… al prezzo di applicare un'altra funzione lineare (vedremo, la versione scritta 2 lezioni fa è quadratica), è sufficiente porre:

```
sort = reverse . quickSort
```

```
s = sort f
> s
[(4,"la"),(2,"panca"),(2,"capra"),(1,"sotto"),(1,"sopra"),
(1,"crepa"),(1,"canta")]
```

Stampa

Per la stampa, continuiamo a fare ragionamenti **modulari**: trasformiamo **una coppia in una stringa** nel formato desiderato, tanto poi c'è `map` che **estende alle liste di coppie**!

Cominicamo a scrivere `showFreqs`.

L'ultimo problema è che `map showFreqs` ha tipo `[String]` è quindi una **lista di stringhe**, ma noi vogliamo **una singola stringa**… no problem, abbiamo il funzionale giusto… Quindi:

```
show = concat . map showFreqs
```

```
-- produce una stringa da una coppia (freq, word)
showFreqs (n, w) = w ++ " : "++ show n ++ ['\n']
```

Raffinamenti: Conteggio

Tutte le funzioni scritte sono lineari, tranne l'ordinamento che è $\theta(n \log n)$ e il conteggio che è $\theta(n^2)$.

Forse sarebbe il caso di eliminare l'operazione quadratica.

Una delle idee principali degli algoritmi è l'**ordinamento** (anche a livello **concettuale** dell'esplorazione di uno spazio di ricerca, o nella generazione di risultati – vedi parte c).

Se ordino (per parola) trovo le occorrezze delle stesse parole in sequenza… vediamo **`countSeqs`**.

Quindi: **`countFreqs = countSeqs . quickSort`**

```
-- span divide una lista tra il prefisso su cui vale
-- un predicato e il suffisso successivo (come splitAt)
span :: (a -> Bool) -> [a] -> ([a], [a])
span p   []   = []
span p (x:xs) = if px then (x:ys, zs) else ([], x:xs) where
   (ys, zs) = span p xs

countSeqs [] = []
countSeqs(x:xs) = (1+length fs, x):countSeqs ls where
      (fs, ls) = span (==x) xs
```

Il sugo di tutta la storia

Abbiamo scritto la **specifica** formale **direttamente** in **Haskell**, descrivendo la soluzione come **composizione di funzioni**.

L'**implementazione** è consistita nello **specificare** tali funzioni.

Possiamo inizialmente usare **funzioni inefficienti** (ad esempio la **`count`** del voting system) salvo poi sostituirla facilmente con l'implementazione più furba vista ora (**raffinamento**).

Il programma risultante può sembrare inefficiente (e probabilmente lo è), ma ha **complessità** $\theta(n \log n)$ in quanto l'operazione più gravosa è l'ordinamento, e le altre sono tutte lineari e vengono eseguite in sequenza (le complessità semplicemente si sommano).

Nulla vieta, anche programmando in un linguaggio **non funzionale** di **pensare** le **specifiche funzionali** come visto in commonWords!

Think Functionally! (certo beware to side-effects & co.)

Addendum

Il problema di Common Words ha una simpatica storia: appa

L'**implementazione** è consistita nello **specificare** tali funzioni.

Possiamo inizialmente usare **funzioni inefficienti** (ad esempio la **count** del voting system) salvo poi sostituirla facilmente con l'implementazione più furba vista ora (**raffinamento**).

Il programma risultante può sembrare inefficiente (e probabilmente lo è), ma ha **complessità** $\theta(n \log n)$ in quanto l'operazione più gravosa è l'ordinamento, e le altre sono tutte lineari e vengono eseguite in sequenza (le complessità semplicemente si sommano).

Nulla vieta, anche programmando in un linguaggio **non funzionale** di **pensare** le **specifiche funzionali** come visto in commonWords!

Think Functionally! (certo beware to side-effects & co.)

Lezione 6a:

Alcuni problmi combinatòri

Powerset

Problema: *Data una lista, calcolare una **lista di liste** con tutte le sue sottoliste (come fosse il powerset, anche se queste sono, in effetti **liste** e **non insiemi**).*

Occorre ragionare **induttivamente**.

Chi sono tutti i sottoinsiemi di *x*:*xs*? Sono i sottoinsiemi che **non contengono** *x*, cioè i sottoinsiemi di *xs*, unito **tutti i sottoinsiemi che contengono *x***, ma questi sono di nuovo i sottoinsiemi di *xs* a cui aggiungo *x*.

Infine chi sono i **sottoinsiemi dell'insieme vuoto**? È l'insieme che **contiene l'insieme vuoto**. Attenzione! Ricordate che la cardinalità di $|\mathcal{P}(X)| = 2^{|X|}$

```
powerset :: [a] -> [[a]]
powerset (x:xs) = map (x:) ts ++ ts where
    ts = powerset xs
powerset [] = [[]]

> powerset [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

Combinazioni

Problema: *Data una lista, calcolare una* ***lista di liste*** *con tutte le sue sottoliste di cardinalità k.*

C'è ovviamente una **soluzione banale**, **da evitare**? Perché?

Occorre di nuovo ragionare **induttivamente**.

Le **combinazioni** di **cardinalità *k* di *x*:*xs*** (lunga *n*) sono le **combinazioni** di ***k*-1 elementi di *xs*** a cui **aggiungo *x***, unito le combinazioni che non contengono *x*, cioè **le combinazioni di *k* elementi di *xs*** (che è lunga *n*-1).

Casi base: *k*=*n* e quindi posso fare **l'unica combinazione** *xs*, oppure *k*=0 e ho l'unica combinazione vuota!

```
combinations xs k = cmbs xs (length xs) k
cmbs xs@(x:txs) n k
    | n==k      = [xs]
    | k==0      = [[]]
    | otherwise = map (x:) (cmbs txs (n-1) (k-1))
                    ++ (cmbs txs (n-1) k)
> combinations [1,2,3,4,5] 3
[[1,2,3],[1,2,4],[1,2,5],[1,3,4],[1,3,5],[1,4,5],
[2,3,4],[2,3,5],[2,4,5],[3,4,5]]
```

```
combinations xs k =
   filter (==k) powerset xs
```

Anagrammi: analisi

Problema: *Data una lista, calcolare una **lista di liste** con tutte le sue permutazioni.*

Ragioniamo ancora **induttivamente**: chi sono gli anagrammi di $xs = [x_1, \ldots, x_n]$? Sono tutte le liste che iniziano con x_1 e poi hanno tutte le permutazioni di $[x_2, \ldots, x_n]$, più tutte quelle che iniziano con x_2 e poi hanno tutte le permutazioni di $[x_1, x_3 \ldots, x_n]$ etc.

Se chiamiamo xs_{-i} la lista a cui ho tolto xi, sono:

$$\bigcup_{i \in [1..n]} x_i \bullet perm(xs_{-i})$$

dove • significa mettere in testa.

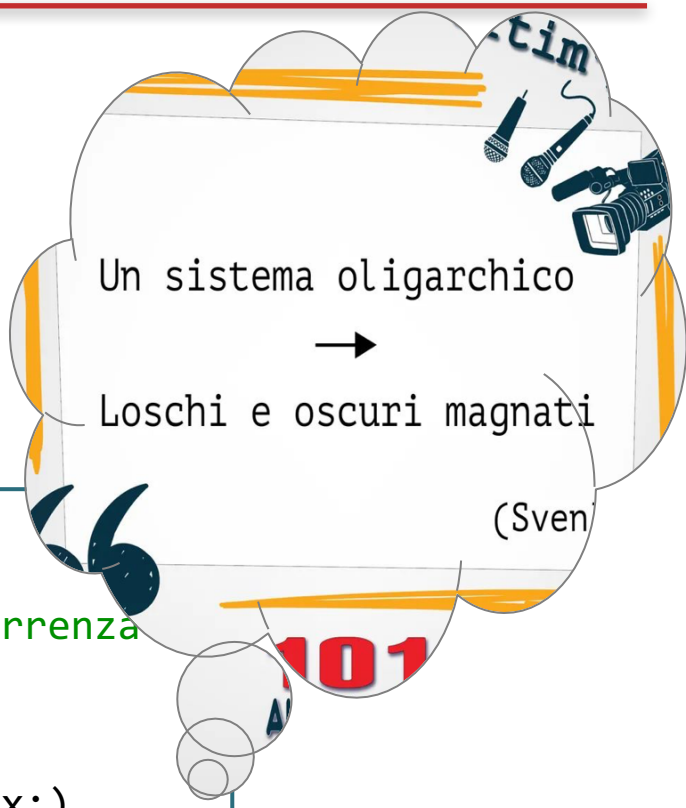
Andando verso il codice:

1. • è chiaramente la funzione `\x xss -> map (x:) xss`
2. ∪ è chiaramente un `foldr (++)`, quindi `concat`.
3. rimane da scrivere una funzione che data *xs*, generi tutte le liste xs_{-i} a cui manca un elemento. La faremo con un `map` su *xs* di una funzione `delete x xs` che rimuove *x* da *xs*.

Anagrammi: codice Haskell

Al solito si finisce
semplicemente ammirati ☺

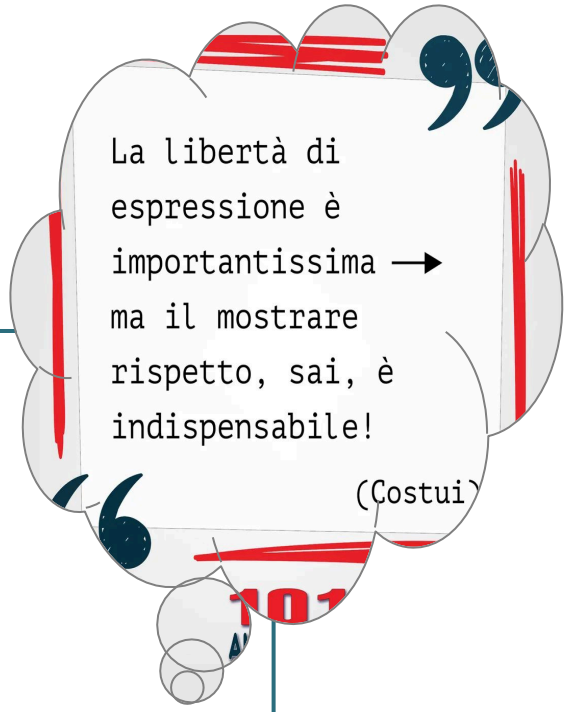
Esercizio: modificare **`permutations`** in modo che non produca ripetizioni quando *xs* ha ripetizioni.




```
delete :: Eq a => [a] -> [a]
delete x (y:ys)
| x==y       = ys  -- assume un'unica occorrenza
| otherwise = y:delete x ys

permutations :: Eq a => [a] -> [[a]]
permutations xs = concat (map (\x-> map (x:)
    (permutations (delete x xs))) xs)
permutations [] = [[]]

> permutations "roma"
["roma","roam","rmoa","rmao","raom","ramo","orma",
"oram","omra","omar","oarm","oamr","mroa","mrao",
"mora","moar","maro","maor","arom","armo","aorm",
"aomr","amro","amor"]
```

Altri Anagrammi

Un altro modo, forse più elegante.




```
xsMenoI :: [a] -> [[a]]
xsMenoI   [] = [[]]
xsMenoI (x:xs) = xs : map (x:) (xsMenoI xs)

> xsMenoI [1,2,3,4]
[[2,3,4],[1,3,4],[1,2,4],[1,2,3]]

-- notate:
> let xs = [1,2,3,4] in zip xs (xsMenoI xs)
[(1,[2,3,4]),(2,[1,3,4]),(3,[1,2,4]),(4,[1,2,3])]

-- ma quindi…
anagrammi [] = [[]]
anagrammi xs =
    concat
        (zipWith (\x -> map (x:))
            xs          -- con zipWith non serve accoppiare
            (map anagrammi (xsMenoI xs))
        )
```

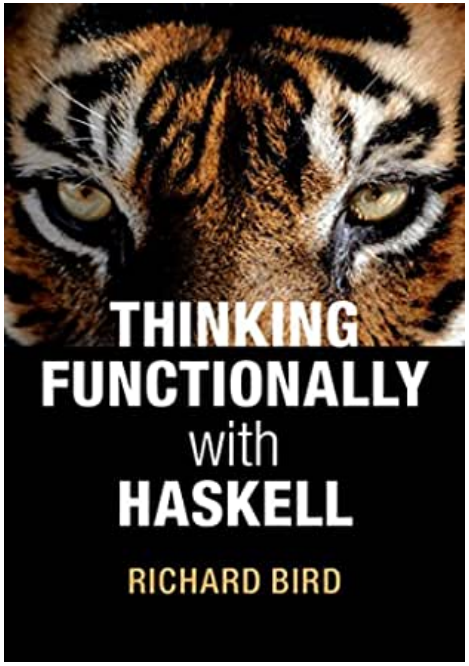
Lezione 6

That's all Folks…

Grazie per l'attenzione…

…Domande?

tratta da:



Lezione 6c:
Numeri in Parole