Tecniche di Programmazione Funzionale e Imperativa

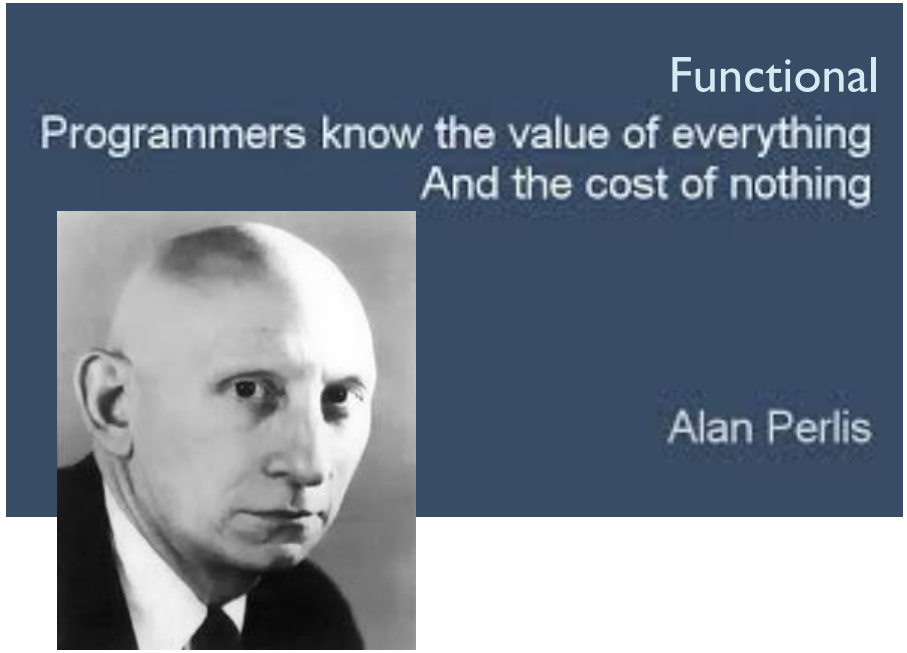
Ivano Salvo

Funzionali sì, ma efficienti!
Spazio e Tempo

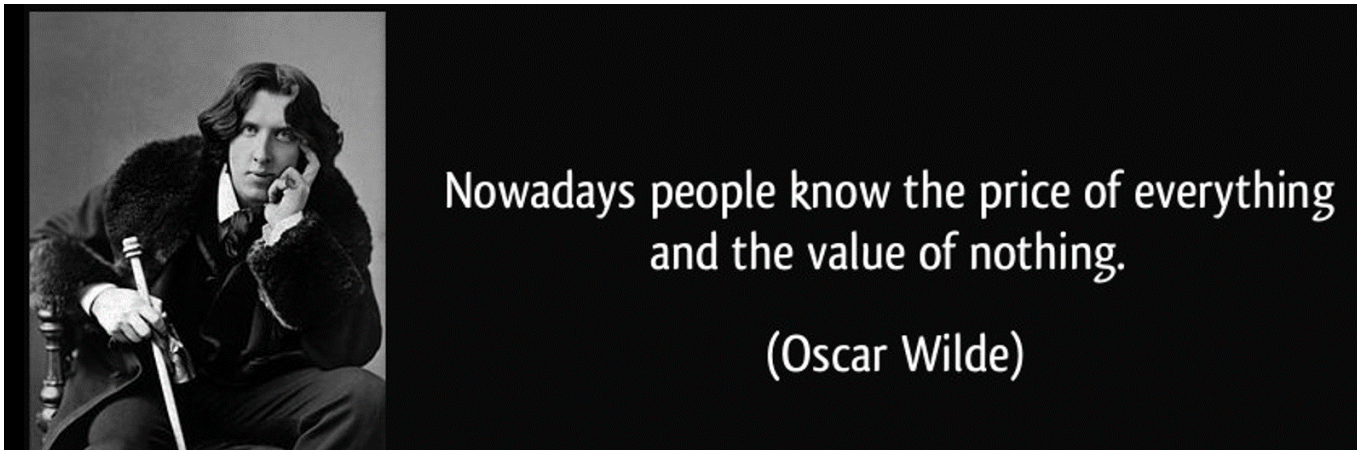
Corso di Laurea in **Informatica**, III anno

SAPIENZA
UNIVERSITÀ DI ROMA

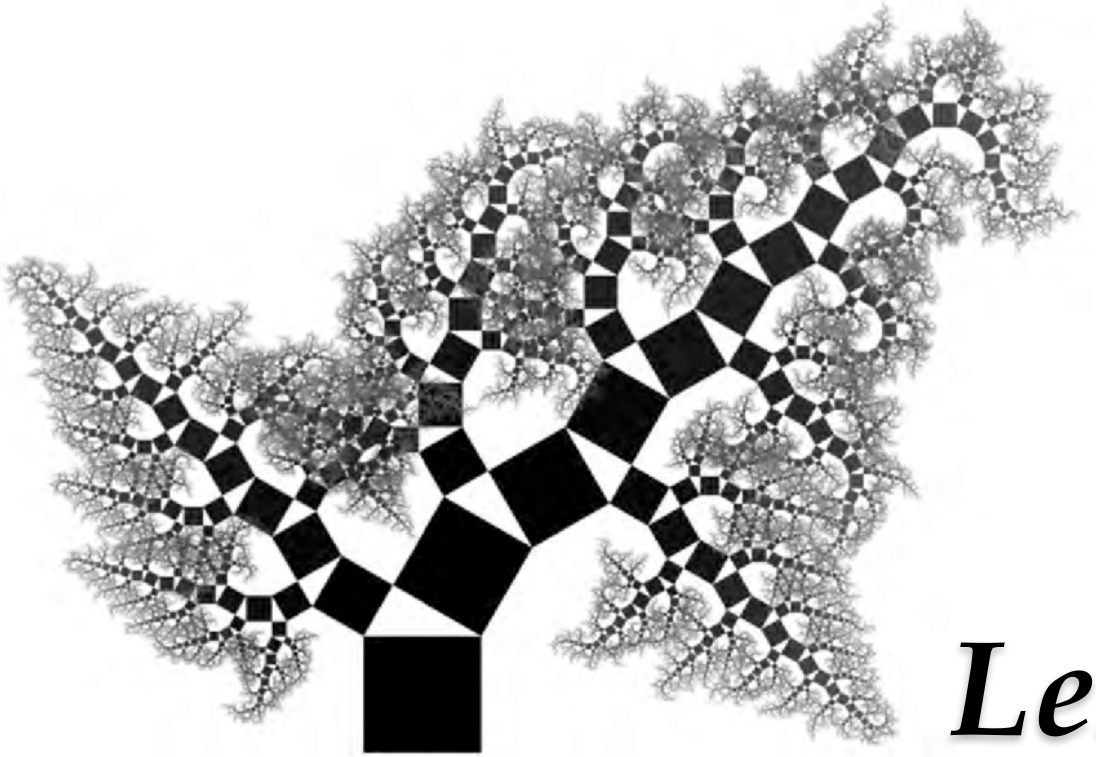
Lezione **9**, 22 marzo 2022



Alan Perlis
Epigrams on Computing
SIGPLAN Notices Vol. 17, No. 9, September 1982



40. There are two ways to write error-free programs; only the third one works.



Lezione 9a:
Altre tecniche di ottimizzazione: Tupling

Qualche utile consiglio "costante"

Qualche suggerimento per migliorare l'efficienza di qualche **costante moltiplicativa** (ma **significativa**!):

- come nel caso della programmazione imperativa, l'aiuto di **profiler**, debugger, e altri strumenti.
- **funzioni di libreria** possono essere più efficienti di quelle home-made (anche se il linea di principio le loro implementazioni sono quelle semplici che abbiamo visto in Haskell stesso).
- **tipi**: il tipo inferito dal type-checker è il **tipo più generale** (e questa è un'informazione utile): tuttavia, conoscere un **tipo specifico** può aiutare il compilatore a fare delle **ottimizzazioni**.

Esempio canonico: `Integer` sono gli interi illimitati, ma `Int` sono gli interi della macchina: se sappiamo che i numeri non crescono molto, è molto più efficiente usare l'aritmetica finita della macchina legata al tipo `Int`.

La strada maestra, ovviamente, è sempre trovare **algoritmi migliori** (in termini di complessità asintotica)

Fibonacci Efficiente con Tupling

Rivediamo come scrivere la funzione efficiente di **Fibonacci** **usando le coppie** piuttosto che i parametri per memorizzare gli ultimi due numeri di Fibonacci.

Un approccio "più generale" definendo un funzionale **for**.

```
-- l'idea è iterare la trasformazione di fibonacci
 fib = fst . fibAux where
   fibAux 0 = (0,1)
   fibAux n = (f1+f2, f1) where
          (f1, f2) = fibAux (n-1)

-- oppure possiamo definire un funzionale for che
-- compone n volte una funzione
for f 0 = \x -> x
for f n = f . (for f (n-1))

>:t for
for :: (Num a, Eq a) => (b -> b) -> a -> b -> b

-- e lo applichiamo alla trasf. di fibonacci
fib n = (fst . (for g n)) (0,1)
          where g (x, y) = (x+y, x)
```

```
 -- for con foldr
for' f n =
   foldr (\x->x) (f.) [0..n]
```

MergeSort

Vediamo un grande classico: **mergeSort**.

Ricordiamo le funzioni predefinite:

`take :: Int->[a]->[a]` e `drop :: Int->[a]->[a]`

che rispettivamente **prendono** e **scartano** i primi *n* elementi da una lista e voilà.

Ma scorro **xs 3 volte** per dividerla in 2!

Ricordiamo anche la funzione **merge** che fonde liste ordinate.

Esercizio: scrivere MergeSort in modo che scomponga la lista in un'unica passata (attenzione che **length** scorre la lista)

```
merge [] ys = ys
merge xs [] = xs
merge xs@(x:txs) ys@(y:tys)
 |   x < y   = x:merge txs ys
 | otherwise = y:merge xs tys
```

```
mergeSort []  = []
mergeSort [x] = [x]
mergeSort xs =
    merge (mergeSort (take n xs))
          (mergeSort (drop n xs))
   where n = length xs `div` 2
```

MergeSort: Raffinamento 1

Un primo miglioramento, si può ottenere usando **splitAt**, che con **un'unica scansione** della lista soddisfa le specifiche:

```
splitAt n xs = (take n xs, drop n xs)
```

Tuttavia ci sono ancora due passate di **xs** per dividerla in due: (ora ne abbiamo 2, una per `length` e una per `splitAt`)?

Aiutandoci con un parametro, possiamo fare **1 sola passata** per calcolare **length** una sola volta…

```
mergeSort xs = merge (mergeSort ls)(mergeSort rs)
        where n = length xs `div` 2
              (ls, rs)  = splitAt n xs
```

splitAt* è una specie di versione tupled di *take* e *drop

```
splitAt n   []   = ([], [])
splitAt 0   xs   = ([], xs)
splitAt n (x:xs) = (x:ls, rs)
  where (ls, rs) = splitAt (n-1) xs
```

```
mergeSort' xs = msAux xs (length xs) where
   mSAux' xs n = merge (mSAux' ls m) (mSAux' rs (n-m))
      where m = n `div` 2
            (ls, rs)  = splitAt m xs
```

MergeSort: Raffinamento 2

Avendo commesso una rapina e di dover dividere il bottino equamente? Contereste prima le banconote?

Usualmente si applica l'**algoritmo del malfattore**: una a me, una a te e così via, fino all'esaurimento delle banconote.

Qui la divisione avviene in un modo diverso, ma **`mergeSort`** richiede solo che **la lista sia divisa a metà**.

L'algoritmo si può scrivere anche in altri modi: **mutua ricorsione** o **leggendo i parametri a due a due**: io trovo efficace questa versione che scambia i ruoli delle liste a ogni chiamata.

```
dividi   []   = ([], [])
dividi (x:xs) = (x:ds, ps) where
       (ps, ds) = dividi xs

mergeSort'' xs =
   merge (mergeSort'' ls) (mergeSort'' rs)
      where (ds, ps) = dividi xs
```

Tupling: quickSort

Ovviamente anche la funzione **`partiziona`** di quickSort può essere fatta via tupling, **evitando la doppia passata** con list comprehension dell'elegante programma visto tempo fa.

Ovviamente, tutti questi programmi possono essere scritti con l'espressione **`let`**.

```
-- possiamo evitare le due passate anche in quickSort
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
   where
      (smaller, larger) = partiziona xs x
      partiziona (x:xs) p =
         if x<=p then (x:s, l) else (s, x:l)
                where (l,s) = partiziona xs
      partiziona [] p = ([], [])
```

Regole generali di tupling

Come già visto, **`foldr`** è un **principio generale di ricorsione** e (quasi) **tutte le funzioni** ricorsive su liste (e non, usando la lista immaginaria **`[0..n]`**) possono essere **definite** usando **`foldr`**.

È quindi utile considerare una regola generale di tupling:

```
(foldr f a xs, foldr g b xs) = foldr h (a, b) xs
```

a patto che `h x (y, z) = (f x y, g x z)`. Ecco la dimostrazione nel caso induttivo (il caso base è ancora più facile).

La proprietà vale per `foldl`, se `h x (y, z) = (f y x, g z y)`.

```
(foldr f a (x:xs), foldr g b (x:xs))
   = { def di foldr }
(f x (foldr f a xs), g x (foldr g b xs))
   = { proprietà di h }
h x (foldr f a xs, foldr g b xs)
   = { induzione }
h x (foldr h (a, b) xs)
   = { def. di foldr }
foldr h (a, b) (x:xs)
```

È più rilevante il guadagno in spazio che evitare la doppia scansione di xs

La cosa **non vale** per **liste parziali**, in quanto in Haskell **`undefined`** ≠ `(undefined, undefined)`!

La seconda, è una **whnf** del **tipo coppia**!

Esempio: per fare un albero...

Per fare un esempio di **tupling** e **accumulating parameters** usate **insieme** per rendere efficiente una computazione, approfittiamo per fare un paio di esercizi sugli **alberi**.

Consideriamo il tipo **`BinTree`** come definito sotto (labels solo sulle foglie) e ricostruiamo un albero bilanciato da una lista.

Anche usando il modo migliore per **dividere** la lista, **costa $\theta(n)$** e quindi la complessità di **`build`** soddisfa all'equazione:

$$T(n) = 2\,T(n/2) + \theta(n),$$

che è la stessa di mergeSort e quindi **`build`** è **$\theta(n \log n)$**: se avessi un vettore potrei fare la divisione in $\theta(1)$ e di conseguenza la complessità scenderebbe a $\theta(n)$.

```
BinTree = R (BinTree a) (BinTree a) | F a

build :: [a] -> BinTree a
build [x] = F x
build  xs = R (build ys) (build zs) where
   (ys, zs) = dividi xs
```

per fare un albero...

Scriviamo la specifica di una funzione **`buildPT`** che usa sia tupling che parametri ausiliari:

- **`m`** è la **lunghezza della lista** da **usare per ricostruire** l'albero (idea: mi basta sapere solo la sua lunghezza senza dividerla), mentre
- il secondo elemento della coppia risultato **è la lista rimasta** di lunghezza **`length xs - m`** (idea: suddivido la lista mentre ricostruisco l'albero)

Questa funzione deve soddisfare l'equazione:

```
build xs = fst (buildPT (length xs) xs)
```

e scriviamo un'equazione ricorsiva diretta per **`buildPT`** (quella sotto dipende da **`build`**)

```
buildPT :: Int -> [a] -> (BinTree a, [a])
buildPT 1 xs = F (head xs) (tail xs)
buildPT n xs = (build take m xs, drop m xs) where
   m = n `div` 2
```

per fare un albero...

Il caso base è evidente: se devo ricostruire un albero di dimensione 1 basta prendere la testa e il rimanente della lista è la coda, come già scritto prima.

Per il caso ricorsivo, partiamo da (ottenuta da **build**, usando **take** e **drop** per fare **dividi**):

```
buildPT n xs = (R (build (take m (take n xs)))
                  (build (drop m (take n xs))),
                drop n xs) where m = n `div` 2
```

A questo punto, per **m<=n** ho le equazioni:

```
take m . take n = take m
drop m . take n = take (n-m) . drop m
```

Questo porta a una nuova def. di **buildPT**:

```
buildPT n xs = (R u v, drop n xs) where
           (u, xs') = buildPT m xs
           (v, xs'') = buildPT (n-m) xs'
           m = n `div` 2
```

Ma posso **non calcolare drop n xs** perché:

xs'' = drop (n-m) xs' = drop (n-m) (drop m xs) = **drop n xs**.

per fare un albero...

Morale: posso scorticare la lista mentre ricostruisco l'albero, senza mai doverla decomporre.

Di fatto, ogni valore viene **tolto dalla lista** quando viene **caricato in una foglia** (il programma nel riquadro).

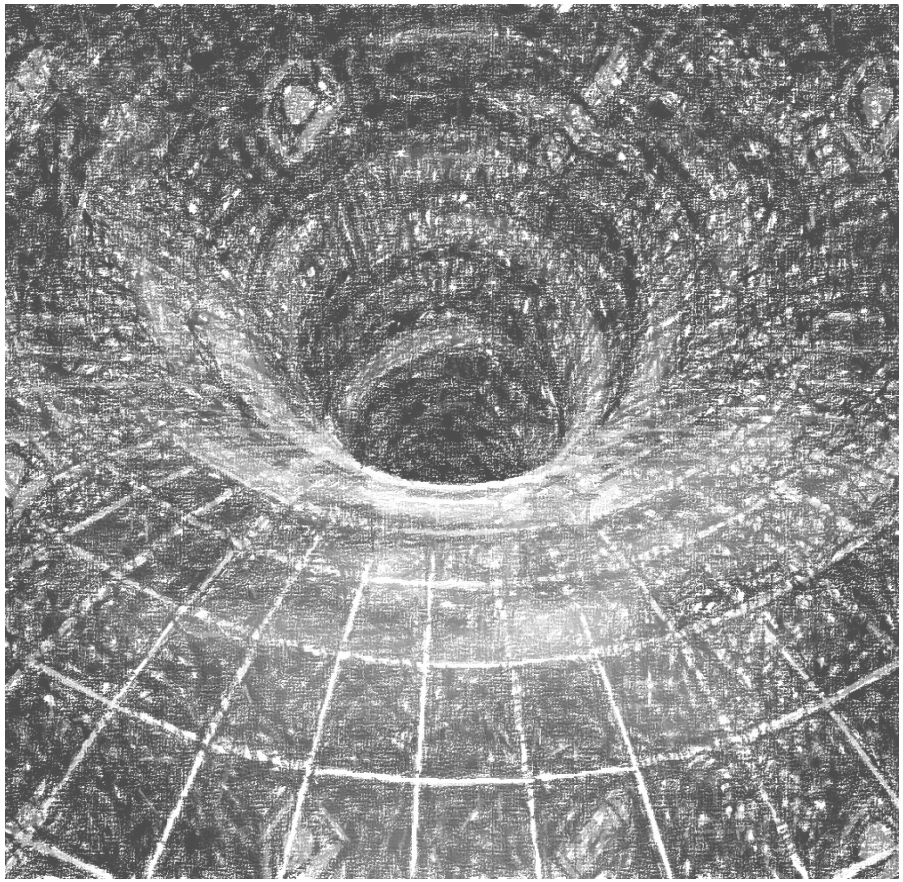
Ora la complessità di ogni chiamata ricorsiva è $\theta(1)$, da cui:

$$T(n) = 2\,T(n/2) + \theta(1),$$

di cui è ben nota la soluzione $T(n) = \theta(n)$.

```
buildPT 1 xs = (F (head xs), tail xs)
buildPT n xs = (R u v, xs'') where
       (u, xs') = buildPT m xs
       (v, xs'') = buildPT (n-m) xs'
       m = n `div` 2

build xs = fst (buildPT xs (length s))
```



Lezione 9b:
Spazio e Tempo

Difficoltà di analisi: lazy evaluation

Nei linguaggi **eager** la complessità dell'applicazione di funzione è semplicemente la complessità di **valutare i parametri** la complessità della **funzione esterna**.

$$T(f \cdot g)|x| = T(f)(|g(x)|) + T(g)|x|$$

dove |x| è la "misura" dell'input x.

Ma nei linguaggi lazy questo è **banalmente falso**, come rimarcato più volte durante il corso: il valore di f potrebbe dipendere solo da una **valutazione parziale** di g, oppure da **una parte** dei dati $g(x)$.

Queste difficoltà suggeriscono di calcolare la complessità sempre riferendosi a un **modello di valutazione eager**, che automaticamente fa da **limite superiore** alla valutazione lazy.

Manca una **teoria compiuta** della **complessità dei programmi lazy**: evidentemente è troppo difficile.

Esempio: minimo e ordinamento

Facciamo un esempio che potrebbe stuzzicare un programmatore lazy che va un po' oltre i casi banali visti finora.

In un mondo eager, la complessità di questa definizione di min sarebbe: $T(min) = T(head) + T(insertSort) = \theta(1)+\theta(n^2)$, ma **che succede in un mondo lazy?**

```
head (insSrt [3,4,1,2])                  = { def. di insSrt }
head (ins 3 (insSrt [4,1,2]))            = { def. di insSrt }
head (ins 3 (ins 4 (insSrt [1,2])))      = … = { def. di insSrt più volte}
head (ins 3 (ins 4 (ins 1 (ins 2 []))))  = { def. di ins }
head (ins 3 (ins 4 (ins 1 (2:[]))))      = { def. di ins }
head (ins 3 (ins 4 (1:[2])))             = { def. di ins }
head (ins 3 (1:(ins 4 [2])))             = { def. di ins }
head (1:(ins 3 (ins 4 [2])))             = { def. di head }
1
```

min è lineare, insSrt calcola Selection Sort!!!

```
-- inserimento ordinato
ins y (x:xs)
 | y <= x    = y:x:xs
 | otherwise = x : ins y xs
```

```
-- insertionSort
insSrt (x:xs) = ins x (insSrt xs)
insSrt   []   = []

min = head . insSrt
```

Efficienza di programmi funzionali

Nei linguaggi funzionali l'**allocazione/deallocazione** della memoria è **implicita**.

Anche peggio: per preservare la **trasparenza referenziale** (= **no side-effects**) i dati sono tutti **immutabili** e quindi ci sono **continue copiature** e quindi allocazioni/deallocazioni di strutture dati.

Haskell è molto efficiente ad allocare/deallocare anche **grandi quantità di memoria**, ma vedremo che come si scrivono i programmi possono aiutare/intralciare questa attività automatica.

Cominciamo con il programma già visto del `powerset`: la clausola `where` evita di ricalcolare i valori, ma la lista ps non può essere subito deallocata (e ovviamente queste liste **crescono esponenzialmente**)

```
 -- con unico calcolo ricorsivo
powerset (x:xs) = ps ++ map (x:) ps where
   ps = powerset xs
powerset [] = [[]]
```

Calcolo del powerset

Abbiamo in realtà scritto un'altra versione (sotto) con un'unica chiamata di powerset sulla lista con un elemento in meno.

Il programma precedente ha complessità $\theta(n2^n)$ in quanto ripete il calcolo del powerset su `xs` 2 volte e quindi la sua complessità è la soluzione dell'equazione $T(n) = 2 \cdot T(n\text{-}1) + \theta(2^n)$.

Questo invece ha complessità "ottima" $\theta(2^n)$, soluzione dell'equazione di ricorrenza $T(n) = T(n\text{-}1) + \theta(2^n)$, ma la lista **`ps`** va **tenuta in memoria** durante tutta la **discesa ricorsiva** (per **eseguire la parte destra**) dove si creano nuove liste.

Haskell potrebbe automaticamente ottimizzare il tempo (**common subformula elimination**), ma alla luce di queste considerazioni lascia al programmatore **libertà di scelta**.

```
 -- powerset con due calcoli ricorsivi
powerset (x:xs) = powerset xs ++ map (x:) powerset xs
powerset [] = [[]]
```

Spazio vs Tempo: memoization

Haskell automaticamente fa **memoization** di variabili che appaoiono **top-level**. Vediamo un primo generatore di primi.

`foo e fooMem` sono la stessa funzione, ma se si valutano i tempi (in **`GHCi`** con **`:set +s`**), si vede che **alla seconda esecuzione** **`fooMem`** ci mette molto meno di **`foo`**: la lista (infinita) **`primes`** rimane in memoria per la parte già valutata.

Le definizioni **locali**, ovviamente vengono **scaricate dalla memoria**

Memoization può **accelerare** i calcoli, ma **saturare** la memoria!

```
 -- generatore di primi
divisors n = [d | d<-[2,n], n `mod` d == 0]
primes = [n | n<-[0..]], divisors n == [n]
fooMem n = sum (take n primes)

foo n = sum (take n primes) where
   divisors n = [d | d<-[2,n], n `mod` d == 0]
   primes = [n | n<-[0..]], divisors n == [n]
```

Ancora: foldl & spazio

Ricordiamo il comportamento di **`foldl`**, che per sua natura **rimane la funzione esterna** da ridurre **finché** non si **esaurisce la lista** e solo dopo si cominciano a fare i conti.

In casi come questo, sarebbe meglio **risparmiare spazio**, valutando **prima** le espressioni. Del resto, **`foldl`** si sposa male con la laziness, in quanto **non estrae mai informazione** prima del tempo.

Haskell ha un funzionale molto particolare:

```
seq: a -> b -> b
```

che è **l'applicazione eager**. L'espressione **`seq x y`** prima valuta **`x`** e poi valuta **`y`** (e **torna** il valore della valutazione della **`y`**). È interessante se **`x`** occorre in **`y`**. **`seq`** **non si può scrivere** in Haskell.

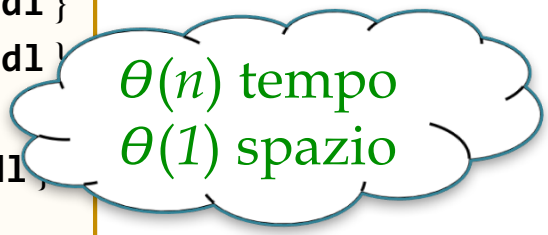
```
foldl (+) 0 [1..1000]                          = { def. di foldl }
foldl (+) 1 [2..1000]                          = { def. di foldl }
foldl (+) (1+2) [3..1000]                      = { def. di foldl }
foldl (+) ((1+2)+3) [4..1000]                  = { def. di foldl }
…
foldl (+) (((…(1+2)+3)+…)+999)+1000)[]  = { def. di foldl }
(((…(1+2)+3)+…)+999)+1000) = …                 = { si valutano i + }
500500
```

Il funzionale foldl'

Il funzionale **foldl'** è scritto in termine di **seq** e come tale forza la valutazione dell'espressione parametro. Vediamo come si usa **seq** e le conseguenze sulla valutazione.

Dalla valutazione, vediamo che **foldl'** forza la valutazione dell'espressione, con **grande vantaggio di spazio** (che diventa $\theta(1)$ invece che $\theta(n)$.

```
foldl' (+) 0 [1..1000] = { def. di foldl }
foldl' (+) 1 [2..1000] = { def. di foldl }
foldl' (+) 3 [3..1000] = { def. di foldl }
foldl' (+) 6 [4..1000] = { def. di foldl }
...
foldl' (+) 500500 []   = { def. di foldl }
500500
```




```
foldl' f v   []   = v
foldl' f v (x:xs) = y `seq` foldl f y xs where
      y = f v x
```

Zucchero Sintattico: $ e $!

Ci sono in Haskell predefiniti un paio di operatori che permettono di scrivere l'applicazione in modo infisso, evitando (a volte) scomode parentesizzazioni "esterne", sono **$** (applicazione **lazy**) e **$!** (applicazione **eager**).

Il vantaggio è che le scritture (**tra loro equivalenti**):

$f_1\ (f_2\ (f_3\ x))$ e $(f_1\ .\ f_2\ .\ f_3)\ x$

sono a loro volta **equivalenti** a: $f_1\ \$\ f_2\ \$\ f_3\ x$ che occasionalmente può essere **più comoda**.

La sintassi **$!** può anche essere **più intuitiva** di **seq**.

```
-- definizione di operatori (si dichiara la precedenza)
infix 0 $ $! -- hanno minima priorità
-- ecco i tipi:
($), ($!) :: (a -> b) -> a -> b
-- definizioni:
f $ x = f x
f $! x = x `seq` f x -- prima valuto il parametro
-- fibonacci con $ e for
fib n = fst $ for g n $ (0,1) where g (a, b) = (a+b, a)
```

Applicazione di foldl': length

La tecnica di accumulating parameters ovviamente soffre del problema di space leaks dovuta a laziness.

La funzione **`length`** è $\theta(n)$ definita usando sia **`foldr`** che **`foldl`**. Tuttavia può essere meglio definirla usando **`foldl'`**, visto che non c'è bisogno di non forzare la valutazione.

La correttezza si può provare come istanza di (vedi Lezione 7): **`foldl (@) e = foldr (#) e`** a patto che: **`(x # y) @ z = x # (y @ z)`** e **`e @ x = x # e`**.

Per **`reverse`** (dimostrato in Lezione 8) e **`length`** è il caso facile in cui **`# = @`**, è associativo ed **`e`** è l'elemento neutro.

$\theta(n)$ tempo
$\theta(1)$ spazio

```
length  = foldl' (\n x -> n+1) 0
reverse = foldl'(\xs x -> x:xs) []
```

Applicazione di foldl': media

Vediamo come ottimizzare il **calcolo della media**. Osserviamo che per motivi di tipo occorre trasformare l'intero calcolato da **`length`** in un Float con **`fromIntegral`**.

Il caso lista vuota viene trattato dal caso **`[]`**. Tuttavia, scritta così:

- **tempo**: si fanno **due scansioni** della lista;
- **spazio**: **`xs`** viene valutata da **`sum`**, e poi **va tenuta in memoria** per il calcolo di **`length`**.

È meglio scandire un'unica volta la lista. A ben vedere occorre foldare (= iterare) la trasformazione $\lambda x.\ \lambda(s, l) \rightarrow (s + x, l + 1)$ per fare contemporaneamente somma e lunghezza.

Questa trasformazione **è corretta**, per le regole generali di tupling viste prima e quindi… tuttavia…

```
mean :: [Float] -> Float
mean [] = 0
mean xs = sum xs / fromIntegral (length xs)

-- mean efficiente: (siamo sicuri?)
mean = s/l where
   (s, l) = foldl' (\x (s, l) -> (s+x, l+1)) (0,0)
```

Insidie nascoste di laziness

Tuttavia le **espressioni** nella forma `(f x, g x)` sono già in **whnf**: non si va a ridurre dentro i costruttori delle strutture dati!

Quindi c'è una piccola differenza tra numeri e tipi strutturati: se voglio veramente ridurre un numero **devo forzare la valutazione degli elementi della coppia** (a ogni applicazione).

Di conseguenza, per ottenere spazio costante, occorre essere ancora più scrupolosi.

Morale: il **linguaggio rimane lazy**, `seq` va spinto fino alla profondità necessaria o desiderata.

```
-- mean efficiente in spazio:
mean = foldl' f (0,0) where
   f (s, l) x = s `seq` l `seq` (s+x, l+1)
```

Sottile differenza tra foldl e foldl'

foldl e **foldl'** sono veramente **equivalenti**?

Ricordiamo che **foldl** **non estrae nessuna informazione** prima di finire la sua ricorsione sulla lista. Sulle **liste parziali** quindi è sempre **undefined**, esattamente come la gemellina eager.

C'è tuttavia una **sottile differenza patologica** quando la funzione **foldata non è stretta**.

Prendete `f n x = if x==0 then undefined else 0`.
Allora avremo che:

`foldl f 0 [0,2]` = `0` ≠ `undefined` = `foldl' f 0 [0,2]`

perché `f (f 0 0) 2 = 0` (lazy, si riduce la `f` esterna), invece **seq** forza la valutazione di `f 0 0` che è **undefined**, e viene propagato da **seq**: infatti vedere `g` sotto.

```
-- seq propaga undefined:
g x y = x `seq` y
>g undefined 3
*** Exception: Prelude.undefined
```

Lezione 9

That's all Folks…

Grazie per l'attenzione…

…Domande?