Esercizi di Programmazione Haskell

Marco Comini

28 marzo 2015, 00:34:27

Draft ON

Versione con soluzioni

In questo documento ho raccolto diversi esercizi per aiutare ad imparare a programmare in Haskell, imparando progressivamente a sfruttare le caratteristiche tipiche dei linguaggi funzionali (quali l'higher-order) che quelle dei linguaggi funzionali lazy. Gli esercizi sono raggruppati per "argomento" e ordinati per difficoltà crescente all'interno delle varie sezioni. Non vi è ordinamento per difficoltà invece fra esercizi di diverse sezioni. Consiglio in ogni caso di procedere in ordine strettamente sequenziale visto che spesso per la soluzione di alcuni esercizi si riutilizza quanto fatto prima. Inoltre anche se non si dovesse riutilizzare in senso stretto quanto fatto in precedenza l'esperienza che si viene a costruire progressivamente aiuta per le nuove soluzioni.

I primi esercizi delle nuove sezioni sembreranno particolarmente facili rispetto a quanto appena terminato, ma si andrà rapidamente a crescere!

Non consiglierei di tentare le soluzioni in modo rigidamente sequenziale fra le varie sezioni, ma di procedere in interleaving, facendo i primi esercizi di ogni sezione e poi aggiungendone un po' per ogni sezione man mano.

Indicativamente una suddivisione in blocchi potrebbe essere

Base Con un minimo di nozioni si possono fare:

Numeri	tutti
Liste	Da 1 a 4
Matrici	1
Alberi Binari di Ricerca	Da 1 a 12
Quad Trees	Da 1 a 8
Matrici mediante Quad Trees	Da 1 a 12

Fold for beginners Tutti questi esercizi presuppongono di utilizzare una funzione di fold in forma "base".

Liste	5
Matrici	Da 2 in poi
Alberi Binari di Ricerca	Da 14 a 20
Alberi Generici	Da 1 a 14
Quad Trees	Da 9 in poi
Matrici mediante Quad Trees	Da 13 a 14

Fold & co Questi esercizi presuppongono di aver imparato ad utilizzare (e/o scrivere) funzioni di fold usando la tecnica del "function level".

Liste	Da 6 in poi
Alberi Binari di Ricerca	Da 21 in poi
Alberi Generici	Da 15 in poi

Raccomandazione

Si scrivano i programmi con variabili anonime ove possibile.

[Si ricorda che in questo contesto un predicato è una funzione con risultato booleano.]

1 Numeri

Si ricordi che si dispone di varie funzioni aritmetiche polimorfe nel Prelude, come

```
(+), (*) :: (Num a) => a -> a -> a
(div) :: (Integral a) => a -> a -> a
```

e quindi si cerchi di scrivere i programmi nel modo più generico possibile in modo da poter usare l'aritmetica a precisione illimitata.

1. Si scriva la funzione fattoriale. Si verifichi il funzionamento calcolando 10000!.

```
fact 0 = 1
fact n = n * fact (n-1)
```

2. Si scriva la funzione $\binom{n}{k}$, combinazioni di k elementi su n.

```
comb n k = div (fact n) (fact (n-k)*fact k)
```

3. Si scriva una funzione che calcoli una lista con tutte le combinazioni su n elementi. Si usi opportunamente

```
map :: (a -> b) -> [a] -> [b]
```

```
allcomb n = map (comb n) [1..n]
```

2 Liste

Si ricordi che si dispone di varie funzioni del Prelude, come

```
foldr :: (a->b->b) -> b -> [a] -> b
```

che accumula, a partire da un opportuno elemento neutro, tutti gli elementi di una lista applicando un operatore binario da destra a sinistra

```
foldr f z [x1,x2,...,xn] = (x1 `f` (x2 `f` ... (xn `f` z)...))
```

1. Scrivere una funzione che data una lista ne costruisce una rimuovendo gli elementi di posizione pari (si conti partendo da 1).

```
eveninlist (x:_:xs) = x:eveninlist xs
eveninlist [] = []
eveninlist [x] = [x]
```

2. Scrivere una funzione che calcola la somma degli elementi di posizione dispari di una lista.

```
sumeveninlist = foldr (+) 0 . eveninlist
```

oppure

```
sumeveninlist (x:_:xs) = x+sumeveninlist xs
sumeveninlist [] = 0
sumeveninlist [x] = x
```

3. Scrivere il QuickSort (polimorfo).

Versione dichiarativa (come da tutorial)

```
quicksort [] = []
quicksort (x:xs) =
  quicksort [ y | y <- xs, y<x ] ++
  (x:quicksort [ y | y <- xs, y>=x ])
```

Versione ottimizzata

```
quicksort xs = qs xs []
  where
    qs [] xs = xs
    qs (x:xs) ys =
      qs smallers (x:qs biggers ys)
        where
          (smallers,biggers) = partition x xs

    partition _ [] = ([],[])
    partition y (x:xs)
      | x > y     = (ls,x:bs)
      | otherwise = (x:ls,bs)
      where (ls,bs) = partition y xs
```

4. Scrivere una funzione che calcola i 2 minori elementi dispari di una lista (se esistono). Ad esempio `minOdd([2,3,4,6,8,7,5])` riduce a (3,5)

Versione dichiarativa

```
minodd = (\(x:y:_)->(x,y)) . sort . (filter odd)
```

Versione ottimizzata

```
minodd (x:xs)
  | odd x     = minodd2 x xs
  | otherwise = minodd xs
 where
    minodd2 y (x:xs)
      | odd x     = if x <= y then minodd3 x y xs
                              else minodd3 y x xs
      | otherwise = minodd2 y xs

    minodd3 x y [] = (x,y)
    minodd3 x y (z:xs)
      | odd z && z < y = if z >= x then minodd3 x z xs
                                   else minodd3 z x xs
      | otherwise = minodd3 x y xs
```

Versione elegante e efficiente

```
minodd xs = foldr updatemins (y1,y2) zs
  where
    (x1:x2:zs) = filter odd xs
    (y1,y2) = if x1>x2 then (x2,x1) else (x1,x2)
    updatemins z (x,y)
      | z>=y      = (x,y)
      | z >= x    = (x,z)
      | otherwise = (z,x)
```

5. Scrivere una funzione che costruisce, a partire da una lista di numeri interi, una lista di coppie in cui
   (a) il primo elemento di ogni coppia è uguale all'elemento di corrispondente posizione nella lista originale e
   (b) il secondo elemento di ogni coppia è uguale alla somma di tutti gli elementi conseguenti della lista originale.

```
annotateAfter = foldr annot []
  where
    annot x []            = [(x,0)]
    annot x xs@((y,z):_) = (x,y+z):xs
```

oppure (meglio)

```
annotateAfter = fst . foldr ( \x (l,c) -> ((x,c):l,x+c) ) ([],0)
```

6. Scrivere una funzione che costruisce, a partire da una lista di numeri interi (provate poi a generalizzare), una lista di coppie in cui
   (a) il primo elemento di ogni coppia è uguale all'elemento di corrispondente posizione nella lista originale e
   (b) il secondo elemento di ogni coppia è uguale alla somma di tutti gli elementi antecedenti della lista originale.

Farlo con foldr o foldl è difficile.

```
annotateBefore xs = annot xs 0
  where
    annot [] _ = []
    annot (x:xs) y = ( (x,y) : annot xs (x+y) )
```

oppure (high order)

```
annotateBefore xs =
  (fst . foldl (\(f,c) x->(f . ((x,c):),x+c)) (id,0)) xs []
```

oppure (laziness + mutua ricorsione)

```
annotateBefore xs = ys
  where
    (sumxs,ys) = foldr acc (0,[]) xs
    acc x (sum,zs) = ( x+sum, (x,sumxs-sum-x):zs )
```

7. Si scriva una funzione Haskell `shiftToZero` che data una lista costruisce un nuova lista che contiene gli elementi diminuiti del valore minimo.

   A titolo di esempio, `shiftToZero [5,4,2,6]` $\Longrightarrow$ `[3,2,0,4]`.

   La funzione **non deve** visitare gli elementi della lista più di una volta (si sfrutti la laziness).

   Farlo con foldr o foldl è difficile.

```
shiftToZero xs = nl
  where
    (mnm,nl) = foldr aggr (head xs,[]) xs
      where
        aggr z (y,nl) = (min z y, (z-mnm):nl)
```

3 Matrici

Le matrici si implementano come liste di liste, per righe o per colonne a seconda delle preferenze.

1. Si scriva una funzione `matrix_dim` che data una matrice ne calcola le dimensioni, se la matrice è ben formata, altrimenti restituisce (-1,-1).

```
matrix_dim [] = (0,0)
matrix_dim [(_:xs)] = (1,1+length xs)
matrix_dim (xs:xxs)
  | m > 0 && length xs == m = (1+n,m)
  | otherwise               = (-1,-1)
      where (n,m) = matrix_dim xxs
```

10/07/06

2. Si scriva una funzione `colsums` che data una matrice calcola il vettore delle somme delle colonne.

```
colsums [] = []
colsums xxs = foldl1 (zipWith (+)) xxs
```

21/09/06

3. Si scriva una funzione `colaltsums` che, data una matrice implementata come liste di liste per righe, calcola il vettore delle somme a segni alternati delle colonne della matrice. Detto $s_j = \sum_{i=1}^{n}(-1)^{i+1}a_{ij}$, `colaltsums`$\left(\begin{pmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{pmatrix}\right) = \begin{pmatrix} s_1 & \dots & s_m \end{pmatrix}$

```
colaltsums [] = []
colaltsums xxs = foldr1 (zipWith (-)) xxs
```

07/09/06

4. Si scriva una funzione `colMinMax` che, data una matrice implementata come liste di liste per righe, calcola il vettore delle coppie (minimo,massimo) delle colonne della matrice.

```
colMinMax [] = []
colMinMax (xs:xxs) = foldl (zipWith mm) mi xxs
  where
    mm (a,b) x = (min a x, max b x)
    mi = zip xs xs
```

21/12/2005

5. Si scriva un predicato `lowertriangular` che determina se una matrice (quadrata) è triangolare inferiore.

   A titolo di esempio, `lowertriangular([[1,0,0],[2,-3,0],[4,5,6]])` restituisce `True`, mentre `lowertriangular([[0,0,1],[2,-3,0],[4,5,6]])` restituisce `False`.

```
lowertriangular = snd . foldl acc (0,True)
  where
    acc (n,b) (_:xs) = (n+1, b && all (0 ==) (drop n xs))
```

13/01/2006

6. Si scriva un predicato `uppertriangular` che determina se una matrice (quadrata) è triangolare superiore.

```
uppertriangular = snd . foldl uptonow (0,True)
  where
    uptonow (n,b) xs = (n+1,b && allzeroes n xs)

    allzeroes 0 xs     = True
    allzeroes n (0:xs) = allzeroes (n-1) xs
    allzeroes _ _      = False
```

27/03/2006

7. Si scriva un predicato `diagonal` che determina se una matrice (quadrata) è diagonale.

```
diagonal = snd . foldl uptonow (0,True)
  where
    uptonow (n,b) xs = (n+1,b && allzeroes n xs)

    allzeroes 0 (_:xs) = all (0 ==) xs
    allzeroes n (0:xs) = allzeroes (n-1) xs
    allzeroes _ _      = False
```

06/04/2006

8. Una matrice quadrata M di ordine n si dice *convergente* con raggio r se il modulo della somma degli elementi di ogni riga, escluso quello sulla diagonale, è inferiore a r.

   Si scriva un predicato `convergent m r` che determina se una matrice (quadrata) `m` è convergente con raggio `r`.

```
convergent m r = snd $ foldl uptonow (0,True) m
  where
    uptonow (n,b) xs = (n+1,b && abs (rowsum n xs) < r)
```

```
rowsum 0 (_:xs) = foldr (+) 0 xs
rowsum n (x:xs) = x+rowsum (n-1) xs
```

9. Si scriva una funzione che data una matrice di dimensioni $m \times n$ restituisce la corrispondente matrice trasposta (di dimensioni $n \times m$).

Soluzione dichiarativa

```
transp [] = []
transp mat@(_:_) = tr mat
  where
    tr ([]:_)      = []
    tr m@((_:_):_) = (map head m):transp (map tail m)
```

oppure (ottimizzata)

```
transp [] = []
transp m@(x:xs) = fr:transp b
  where
    (fr,b) = splitCol m

    splitCol [] = ([], [])
    splitCol ([x]:m) = (x:frm, [])
      where
        (frm, []) = splitCol m
    splitCol ((x:y:xs):m) = (x:frm, ((y:xs):bm))
      where
        (frm, bm) = splitCol m
```

oppure (più elegante anche se ha più problemi di Garbage Collection)

```
transp = foldr add_col []
  where
    add_col [] [] = []
    add_col (h:hs) [] = [h]:add_col hs []
    add_col (h:hs) (rs:rss) = (h:rs):add_col hs rss
```

oppure (la più elegante di tutte!) pensando bene a cosa sta facendo `add_col` e sfruttando la laziness e la definizione di zip

```
trasposta xss = foldr (zipWith (:)) (repeat []) xss
```

20/12/2007

10. Si scriva un predicato `isSymmetric` che, data una matrice quadrata, determina se è simmetrica.

```
isSymmetric m = m == transp m
```

MAI dato

11. Si scriva una funzione che data una matrice di dimensioni $n \times k$ ed una $k \times m$ restituisca la matrice prodotto corrispondente (di dimensioni $n \times m$). Si assuma di moltiplicare matrici con dimensioni compatibili e (se facesse comodo) matrici non degeneri.

```
matrix_multiply m1 m2 = map (flip vectmatrmult m2) m1
  where
    vectmatrmult [x] [ys] = map (x*) ys
    vectmatrmult (x:xs) (ys:yys) = zipWith ((+) . (x*)) ys (vectmatrmult xs yys)
```

4 Alberi Binari di Ricerca

Si definiscano gli Alberi Binari di Ricerca col seguente tipo di dato astratto (polimorfo)

```
data (Ord a, Show a, Read a) => BST a = Void | Node {
  val :: a,
  left,right :: BST a
  }
   deriving (Eq, Ord, Read, Show)
```

e si usi (per comodità) lo stesso tipo di dato anche per Alberi Binari normali.

1. Scrivere una funzione che calcola la somma dei valori di un albero a valori sommabili.

```
bstSum Void = 0
bstSum (Node x l r) = x + bstSum l + bstSum r
```

2. Scrivere una funzione che calcola la somma dei valori dispari di un albero a valori sommabili su cui sia utilizzabile la funzione `odd`.

```
bstSumOdd Void = 0
bstSumOdd (Node x l r)
  | odd x = x + bstSumOdd l + bstSumOdd r
  | otherwise = bstSumOdd l + bstSumOdd r
```

3. Si scriva un predicato `samesums` che presa una lista di alberi $[t_1, \dots, t_n]$ determina se le somme $s_1, \dots, s_n$ dei valori degli elementi di ogni t_i sono tutte uguali fra loro.

```
samesums = allsame . (map bstSum)
 where
  allsame xs = all (head xs==) xs
```

Notare che la laziness del linguaggio ci assicura di non calcolare inutilmente la somma di alberi dopo che si è scoperto che le somme non sono più uguali.

4. Scrivere un predicato `bstElem` (infisso magari) per determinare se un valore è presente in un BST.

```
bstElem x Void = False
bstElem x (Node y l r)
  | x==y      = True
  | x<y       = bstElem x l
  | otherwise = bstElem x r
```

5. Si scriva una funzione per eseguire l'inserimento di un dato x in un albero t.

```
insert x Void = Node x Void Void
insert x (Node y l r)
  | x<=y      = Node y (insert x l) r
  | otherwise = Node y l (insert x r)
```

6. Si scriva una funzione `bst2List` che calcola la lista ordinata degli elementi di un BST. Ci si assicuri di scrivere una funzione lineare.

```
bst2list t = bsttolist t []
 where
  bsttolist Void xs = xs
  bsttolist (Node x l r) xs = bsttolist l (x:bsttolist r xs)
```

7. Si scriva una (semplice) funzione di ordinamento di liste come combinazione di funzioni fatte nei precedenti esercizi.

```
bst2list = foldr insert Void

bstsort = bst2list . bst2list
```

21/12/2005

8. Si scriva una funzione `filtertree` p t che costruisce una lista (ordinata) di tutti gli elementi dell'albero t che soddisfano il predicato p.

```
filtertree p t = filt t []
 where
  filt Void xs = xs
  filt (Node x l r) xs
    | p x       = filt l (x:filt r xs)
    | otherwise = filt l (filt r xs)
```

Lezione?

9. Si scriva una funzione `annotate` che costruisca un nuovo BST che in ogni nodo contenga, al posto del valore originale, una coppia composta dal medesimo valore e dall'altezza del nodo stesso (la lunghezza del massimo cammino, cioè $1 + max(height(sx), height(dx))$. Si scelga di attribuire all'albero vuoto 0 o -1 a seconda delle preferenze.

   [Con una opportuna scelta dell'ordine di ricorsione si può fare in tempo lineare]

```
annotate = fst . annot
  where
    annot Void = (Void, -1)
    annot (Node x l r) = (Node (x,d) newL newR, h)
      where
        h = 1+max hL hR
        (newL,hL) = annot l
        (newR,hR) = annot r
```

Lezione?

10. Si scriva un predicato (funzione a valori booleani) `almostBalanced` per determinare se un albero binario ha la seguente proprietà: per ogni nodo le altezze dei figli destro e sinistro differiscono al massimo di 1.

```
almostBalanced = fst . almost
 where
  almost Void = (True, -1)
  almost (Node _ l r) = (balL && balR && bal,h)
   where
     (balL,hL) = almost l
     (balR,hR) = almost r
     bal = abs (hL-hR) <= 1
     h = 1+max hL hR
```

MAI dato

11. Data la seguente definizione del tipo di dato astratto (polimorfo) *Weighted Binary Search Tree* che consiste in un BST in cui in ogni nodo viene mantenuta l'altezza del nodo stesso.

```
data WBST a = Void | Node a Int (WBST a) (WBST a)
```

Si scriva una funzione `insert` che inserisce un nuovo valore in un WBST.

```
insert x Void = Node x 1 Void Void
insert x (Node y d l r)
  | x <= y    = Node y (max d (1+nld)) nl r
  | otherwise = Node y (max d (1+nrd)) l nr
    where
      nl = insert x l
      (Node _ nld _ _) = nl
      nr = insert x r
      (Node _ nrd _ _) = nr
```

10/04/2008 e 11/07/2008

12. Si scriva una funzione `diff2next` che, dato un albero binario di ricerca, costruisce un albero binario di ricerca (annotato) di coppie dove il primo elemento di ogni coppia è l'elemento dell'albero originale mentre il secondo elemento è `Just(`la differenza rispetto al valore successivo`)`, secondo l'ordinamento dei valori contenuti, oppure `Nothing` per il nodo di valore massimo. A titolo di esempio,

```
Node 4 Void (Node 7 (Node 5 Void Void) Void)
```

restituisce la soluzione

```
Node (4,Just 1) Void (Node (7,Nothing) (Node (5,Just 2) Void Void) Void).
```

```
diff2next :: (Num a,Ord a,Show a) => BST a -> BST (a,Maybe a)
diff2next n = fst $ d2n n Nothing
 where
  d2n Void xs = (Void,xs)
  d2n (Node x l r) mx = (Node (x,d) nl nr,mxl)
    where
      (nr,mxr) = d2n r mx
      (nl,mxl) = d2n l $ Just x
```

```
      d = case mxr of
           Nothing -> Nothing
           Just y  -> Just (y-x)
```

2012-02-21

13. Si scriva una funzione che dato un BST ne restituisce la lista degli elementi ottenuti visitando l'albero a livelli.

```
bst2bsf t = forest2list [t]
  where
    forest2list []                 = []
    forest2list (Void:ts)          = forest2list ts
    forest2list ((Node x l r):ts) = x:forest2list (ts++[l,r])
```

oppure

```
bst2bsf t = forest2bsf [t]
  where
    forest2bsf [] = []
    forest2bsf fs@(_:_) = lev1++forest2bsf f2s
      where (lev1,f2s) = forest2level fs

    forest2level [] = ([],[])
    forest2level ((Node x l r):fs) = ( (x:xs), (l:r:ts) )
      where (xs,ts) = forest2level fs
    forest2level (Void:fs) = forest2level fs
```

Si consideri d'ora in poi la seguente generalizzazione a BST della funzione foldr su liste:

```
fold :: (Ord a) => (a -> b -> b -> b) -> b -> BST a -> b
fold _ z Void = z
fold f z (Node x l r) = f x (fold f z l) (fold f z r)
```

Ci si assicuri di scrivere funzioni lineari (non ha senso scrivere soluzioni che usino "forzosamente" una fold).

14. Si scriva una funzione `treeheight` per calcolare l'altezza di un albero usando opportunamente `fold`.

```
treeheight t = fold acc -1 t
  where
    acc _ nl nr = 1 + max nl nr
```

15. Si riscriva la funzione `annotate` dell'Esercizio 9 usando opportunamente `fold`.

```
annotate = fst . fold annot (Void, -1)
  where
    annot x (newL,hL) (newR,hR) = (Node (x,d) newL newR, h)
      where
        h = 1+max hL hR
```

16. Si riscriva la funzione `almostBalanced` dell'Esercizio 10 usando opportunamente `fold`.

```
almostBalanced = fst . fold almost (True, -1)
 where
  almost _ (balL,hL) (balR,hR) = (balL && balR && bal,h)
   where
     bal = abs (hL-hR) <= 1
     h = 1+max hL hR
```

20/12/2007

17. Si scriva una funzione `maxDiameter` che data una lista l di BST determina il massimo dei diametri dei BST di l. Il diametro di un BST è la lunghezza del massimo cammino fra due nodi, indipendentemente dall'orientamento degli archi.

```
diameter t = fst $ fold aux (-1,-1) t
  where
    aux _ (d1,h1) (d2,h2) = (d1 `max` d2 `max` (2+h1+h2),1+max h1 h2)
```

13/01/2006

18. Si scriva un predicato `isBST`, usando opportunamente `fold`, che dato un albero verifica se i valori in esso contenuti soddisfano la proprietà strutturale dei Binary Search Trees.

```
isBST t = case fold aggr Nothing t of
            Nothing      -> True
            Just (b,_,_) -> b
 where
  aggr x l r = Just $ ac l r

  ac Nothing                Nothing                = (True,
x,   x)
  ac Nothing                (Just (br,minr,maxr)) = (x < minr && br,
x,   maxr)
  ac (Just (bl,minl,maxl)) Nothing                = (maxl <= x && bl,
minl,x)
  ac (Just (bl,minl,maxl)) (Just (br,minr,maxr)) = (maxl <= x && x < minr && bl && br,m
```

06/04/2006

19. Si scriva un predicato `isAVL` che dato un albero secondo la seguente definizione di tipo

```
data (Ord a) => ABST a = Void | Node Bal a (ABST a) (ABST a)
    deriving (Eq, Ord, Read, Show)
data Bal = Left | Bal | Right  deriving (Eq, Ord, Read, Show)
```

determina se è ben formato, cioè se

- la differenza fra le profondità dei sottoalberi destro e sinistro di un qualunque nodo è al massimo 1;
- le etichette Bal dei nodi sono consistenti con lo (s)bilanciamento.

```
foldAVL _ z AVoid = z
foldAVL f z (ANode b x l r) = f b x (foldAVL f z l) (foldAVL f z r)

isAVL  = fst . foldAVL almost (True, -1)
 where
  almost l _ (bL,hL) (bR,hR) = (bL && bR && bal && valid l,h)
   where
     bal = abs (hL-hR) <= 1
     h = 1+max hL hR

     valid Left  = hL > hR
     valid Right = hL < hR
     valid Bal   = hL == hR
```

10/07/2006 e 0

20. Si scriva un predicato isRBT che dato un albero secondo la seguente definizione di tipo

```
data (Ord a) => RBT a = Void | Node a Color (RBT a) (RBT a)
    deriving (Eq, Ord, Read, Show)
data Color = Red | Black  deriving (Eq, Ord, Read, Show)
```

determina se è ben formato, cioè se

- ogni nodo contiene un valore non minore dei valori del suo sottoalbero sinistro e minore dei valori del sottoalbero destro;
- tutti i cammini dalla radice a una foglia hanno lo stesso numero di nodi Black;
- i nodi Red devono avere genitore Black;
- la radice è Black.

```
foldRBT _ z CVoid = z
foldRBT f z (CNode x c l r) = f x c (foldRBT f z l) (foldRBT f z r)

isRBT :: (Ord a) => RBT a -> Bool
isRBT t = case foldRBT aggr Nothing t of
            Nothing          -> True
            Just (b,c,_,_,_) -> c == Black && b
  where
    aggr x c l r = Just (b,c,nb+nbs,min,max)
      where
        (b,nbs,min,max) = ac l r

        ac Nothing Nothing                                          = (True,
0,  x,   x)
        ac Nothing (Just (br,cr,nbr,minr,maxr))                     = (x < minr && br
nbr,x,   maxr)
        ac (Just (bl,cl,nbl,minl,maxl)) Nothing                     = (maxl <= x && bl
nbl,minl,x)
        ac (Just (bl,cl,nbl,minl,maxl)) (Just (br,cr,nbr,minr,maxr)) = (maxl <= x && x
                                                                       br && col2 cl c

        cBl = (c == Black)

        nb = if cBl then 1 else 0

        col y = impl (y == Red) cBl
        col2 y z = impl (y == Red || z == Red ) cBl

    impl a b = not a || b
```

21. Si riscriva la funzione `bst2List` dell'Esercizio 6 usando opportunamente `fold`.

Difficile quanto l'Esercizio 6 delle liste

```
bst2list t = fold aggr id t []
  where
    aggr x lacc racc = lacc . (x:) . racc
```

21/12/2005

22. Si riscriva la funzione `filtertree` dell'Esercizio 8 usando opportunamente `fold`.

```
filtertree p t = fold aggr id t []
  where
    aggr x lacc racc
      | p x = lacc . (x:) . racc
      | otherwise = lacc . racc
```

23. Si riscriva la funzione `diff2next` dell'Esercizio 12 usando opportunamente `fold`.

```
diff2next t = fst $ fold aggr init t Nothing
  where
    init x = (Void, x)
    aggr x lacc racc mx = (Node (x,d) nl nr,mxl)
      where
        (nr,mxr) = racc mx
        (nl,mxl) = lacc $ Just x
        d = case mxr of
              Nothing -> Nothing
              Just y  -> Just (y-x)
```

27/03/2006

24. Si scriva una funzione `limitedVisit` che dato un BST e due valori x, y costruisce la lista (ordinata) degli elementi dell'albero compresi nell'intervallo di valori da x a y.

Garantire che `let d=d in limitedVisit 3 7 (Node 7 (Node 2 (Node d Void Void) Void) (Node d Void Void))` termini non è immediato.

Riutilizzando quanto fatto in precedenza

```
limitedVisit n m  = filtertree (\x->n<=x && x<=m)
```

anche se si va ad eseguire una scansione su un sacco di elementi di un albero sicuramente inutili, quindi si può migliorare in

```
limitedVisit n m t
  | n<=m      = fold aggr id t []
  | otherwise = []
  where
    aggr x lacc racc
      | x < n     = racc
      | x > m     = lacc
      | x == m    = lacc . (x:)
      | otherwise = lacc . (x:) . racc
```

Tra l'altro in questo modo

```
let d=d in limitedVisit 3 7
  (Node 7 (Node 2 (Node d Void Void) Void) (Node d Void Void))
```

valuta a `[7]` senza divergere.

21/09/2006

25. Si scriva una funzione `shiftToZero` che dato un BST t costruisce un nuovo BST isomorfo che contiene gli elementi t diminuiti del valore minimo di t.

    La funzione **non deve** visitare un nodo dell'albero t più di una volta (si sfrutti laziness e scoping mutuamente ricorsivo).

    Difficile quanto l'Esercizio 7 delle liste

```
shiftToZero t = nt
  where
    (minim,nt) = fold aggr (Nothing,Void) t

    (Just mn) = minim

    aggr x (y,nl) (_,nr) = (f y, Node (x-mn) nl nr)
      where
        f Nothing = Just x
        f y = y
```

5 Alberi Generici

Si definiscano Alberi "generici" col seguente tipo di dato astratto (polimorfo)

```
data (Eq a,Show a) => Tree a = Void | Node a [Tree a]
    deriving (Eq,Show)
```

Con questo tipo di dato ci sono vari possibili modi per rappresentare una foglia: (`Node` x `[]`), (`Node` x `[Void]`), (`Node` x `[Void, Void]`), ..., (`Node` x `[Void, ...Void]`), ... Rinunciando all'albero vuoto si avrebbe una formulazione unica come

```
data (Eq a,Show a) => NonEmptyTree a = Node a [NonEmptyTree a]
  deriving (Eq,Show)
```

ma nel seguito abbiamo bisogno dell'albero vuoto e andremo a convivere con la rappresentazione non univoca.

1. Si scriva una generalizzazione della funzione `foldr` delle liste per Alberi Generici che abbia il seguente tipo:

```
treefold :: (Eq a,Show a) => (a->[b]->b) -> b -> Tree a -> b
```

```
treefold f z Void = z
treefold f z (Node x []) = f x [z]
treefold f z (Node x ts) = f x $ map (treefold f z) ts
```

2. Si scriva una funzione `height` per calcolare l'altezza di un albero usando opportunamente la `treefold` dell'Esercizio 1. Si attribuisca altezza -1 all'albero vuoto.

   Si colga l'occasione per verificare che `treefold` sia stata definita correttamente e quindi

```
height (Node 'a' $ replicate n Void)
```

restituisca sempre 0 al variare di n.

```
height t = treefold aggr (-1) t
  where
    aggr _ xs = 1 + maximum xs
```

3. Si scriva una funzione `simplify` per eliminare i figli `Void` ridondanti usando opportunamente la `treefold` dell'Esercizio 1.

```
simplify t = fold aggr Void t
  where
    aggr x ts = Node x $ filter (/=Void) ts
```

4. Si scrivano le generalizzazioni delle funzioni `foldr` e `foldl` delle liste per Alberi Generici aventi i seguenti tipi (abbiamo bisogno di due "zeri" corrispondenti all'albero vuoto e alla lista di alberi vuota):

```
treefoldr :: (Eq a,Show a) => (a->b->c)->c->(c->b->b)->b->Tree a->c
treefoldl :: (Eq a,Show a) => (b->a->c)->c->(c->b->b)->b->Tree a->c
```

Con queste fold non c'e bisogno di costruire la lista intermedia a cui applicare la funzione di "aggregazione" ma si esegue il lavoro man mano.

```
treefoldr f zf g zg Void = zf
treefoldr f zf g zg (Node x xs) = f x $ foldr (g . treefoldr f zf g zg) zg xs

treefoldl f zf g zg Void = zf
treefoldl f zf g zg (Node x xs) = f (foldl g' zg xs) x
  where g' b ta = g (treefoldl f zf g zg ta) b
```

5. Si riscriva la funzione `height` per calcolare l'altezza di un albero usando opportunamente la `treefoldr` dell'Esercizio 4.

```
height t = treefoldr node (-1) max (-1) t
  where
    node _ = (1+)
```

6. Si riscriva la funzione `simplify` per eliminare i figli `Void` ridondanti usando opportunamente la `treefoldr` dell'Esercizio 4.

```
simplify t = treefoldr Node Void aggr [] t
  where
    aggr Void xs = xs
    aggr x xs    = x:xs
```

11/07/2008 e 17/09/2008

7. Si scriva una funzione `degree` che restituisce il grado di un albero (il massimo del numero di figli per ogni nodo).

```
degree t = treefoldr aggr1 (-1) aggr2 ((-1),0) t
  where
    aggr1 _ (md,l) = max md l
    aggr2 d (md,l) = (max d md,l+1)
```

17/09/2008

8. Si scriva una funzione `transpose` che restituisce il trasposto di un albero (per ogni nodo i trasposti dei figli in ordine inverso).

```
transpose t = treefoldl (flip Node) Void (:) [] t
```

Mai dato?

9. Si scriva un predicato `issymm` che stabilisce se un albero ha una forma simmetrica (cioè è uguale, non considerando il contenuto, al suo trasposto).

```
skeleton t = treefoldr aggr1 Void (:) [] t
  where
    aggr1 _ ts = Node () ts

issymm t = s == transpose s
  where s = skeleton t
```

11/07/2011

10. Si scriva una funzione `normalize` che dato un albero con valori nella classe Integral costruisca un nuovo albero che in ogni nodo contenga, al posto del valore originale, tale valore moltiplicato per l'inverso dell'altezza. (Si presti attenzione nell'espressione della moltiplicazione in modo da avere tipi compatibili).

```
normalize :: (Integral a, Fractional b) => Tree a -> Tree b
normalize t = fst $ treefoldr aggr1 (Void, -1) aggr2 (0,[]) t
  where
    aggr1 x (h,ts) = (Node ((fromIntegral x) / (fromIntegral $ h+1)) ts,h+1)

    aggr2 (t,h) (hm,ts) = (max h hm, t:ts)
```

13/06/2011

11. Si scriva una funzione `annotate` che costruisca un nuovo albero che in ogni nodo contenga, al posto del valore originale, una coppia composta dal medesimo valore e dall'altezza del nodo stesso.

```
annotate t = fst $ treefoldr aggr1 (Void, -1) aggr2 (-1,[]) t
  where
    aggr1 x (h,ts) = (Node (x,h+1) ts,h+1)

    aggr2 (t,h) (hm,ts) = (max h hm, t:ts)
```

01/07/2009

12. Si scriva un predicato `iscorrect` che determina se un albero è un albero di parsing secondo le regole di una grammatica codificata mediante una funzione che, dato un simbolo, restituisce la lista delle possibili espansioni (stringhe di simboli) secondo le produzioni.

Assumiamo ovviamente di avere Tree ben formati in input

```
iscorrect rules t =
  case treefoldr aggr1 Nothing aggr2 (Just []) t of
    Nothing -> False
    _       -> True
  where
    aggr2 (Just x) (Just xs) = Just (x:xs)
    aggr2 _        _         = Nothing

    aggr1 x (Just xs) | any (xs==) (rules x) = Just x
    aggr1 _ _                                = Nothing
```

2009/02/02

13. Si scriva una funzione `diameter` che determina il diametro di un albero. Il diametro di un albero è la lunghezza del massimo cammino fra due nodi, indipendentemente dall'orientamento degli archi.

```
diameter t = snd $ treefoldr aggr1 ((-1),(-1)) aggr2 ((-1),(-1),(-1)) t
  where
    aggr1 _ (h1,h2,dm) = (1+h1, max dm (2+h1+h2))
    aggr2 (h,d) (h1,h2,dm)
      | h <= h2   = (h1,h2,ndm)
      | h <= h1   = (h1,h,ndm)
      | otherwise = (h,h1,ndm)
      where ndm = max d dm
```

14. Si scriva una funzione `maxPathWeight` che, dato un albero di valori numerici positivi, determina il massimo peso di tutti i cammini, indipendentemente dall'orientamento degli archi. Il peso di un cammino, è la somma dei valori dei nodi del cammino.

```
maxPathWeight :: (Ord a, Num a) => Tree a -> a
maxPathWeight t = snd $ treefoldr aggr1 (0,(-1)) aggr2 (0,0,(-1)) t
  where
    aggr1 x (whmx,whsmx,wmx) = (x+whmx, max wmx (x+whmx+whsmx))

    aggr2 (wh,w) (whmx,whsmx,wmx)
      | wh <= whsmx   = (whmx,whsmx,nwmx)
      | wh <= whmx    = (whmx,wh,nwmx)
      | otherwise = (wh,whmx,nwmx)
      where nwmx = max w wmx
```

10/04/2008

15. Si scriva una funzione `preorder` che restituisce la lista degli elementi di una visita in preordine.

```
preorder t =  treefoldr aggr id (.) id t []
  where
    aggr x acc = (x:) . acc
```

26/02/2009

16. Si scriva una funzione `frontier` che restituisce la frontiera di un albero (la lista degli elementi delle foglie).

Quadratica semplicissima

```
frontier t = treefoldr aggr1 [] (++) [] t
  where
    aggr1 a [] = [a]
    aggr1 _ xs = xs
```

oppure lineare per Tree ben formati

```
frontier t = treefoldr aggr1 id aggr2 Nothing t []
  where
    aggr1 x (Just acc) = acc
    aggr1 x Nothing    = (x:)

    aggr2 f (Just g) = Just $ f . g
    aggr2 f Nothing  = Just f
```

oppure lineare per Tree qualsiasi

```
frontier t =
  case treefoldr aggr1 Nothing aggr2 Nothing t of
    Nothing -> []
    Just f -> f []
  where
    aggr1 x (Just acc) = Just $ acc
    aggr1 x Nothing    = Just $ (x:)

    aggr2 (Just f) (Just g) = Just $ f . g
    aggr2 jg       Nothing  = jg
    aggr2 Nothing  jg       = jg
```

17. Si scriva una funzione `smallParents` che restituisce la lista dei (valori dei) nodi che son genitori *ma non nonni* di qualche altro nodo. La lista deve essere prodotta rispettando l'ordine di comparizione nell'albero.

```
smallParents t =
  case treefoldr aggr1 Nothing aggr2 Nothing t of
    Nothing -> []
    Just f -> f []
  where
    aggr1 x (Just (Just acc)) = Just acc
    aggr1 x (Just Nothing) = Just (x:)
    aggr1 x Nothing    = Nothing

    aggr2 x Nothing  = Just x
    aggr2 x (Just Nothing)  = Just x
    aggr2 Nothing x  = x
    aggr2 (Just f) (Just (Just g)) = Just $ Just $ f . g
```

18. Si scriva un predicato `arithmSmallParents` che determina se i (valori dei) nodi che son genitori *ma non nonni* di qualche altro nodo sono una progressione aritmetica ($\exists y, z : \forall i.\, x_i = y + i * z$).

```
arithmSmallParents = arithm . smallParents
  where
    arithm (y:x:xs) = fst $ foldl aggr (True,x) xs
      where
        delta = x-y
        aggr (b,v) z = (b && z-v == delta,z)
    arithm _ = True
```

oppure si può scrivere una soluzione diretta fondendo i due codici in un'unica funzione.

```
FARE
```

11/07/2011

19. Codificando un'espressione e in notazione polacca inversa mediante una lista di tipo `Num a =>[Either a (op,Int)]`, si scriva una funzione `rpn2tree` che data e costruisca un opportuno albero di sintassi astratta. La componente intera della coppia che identifica un operazione ne stabilisce l'arità. Ad esempio per x y z s/2 m/2 otteniamo `Node m [Node s [Node z [], Node y []], Node x []]`.

    Se servisse si assuma che le e in input siano ben formate (corrispondano ad una vera espressione).

SISTEMARE

```
normalize :: (Integral a, Fractional b) => Tree a -> Tree b
normalize t = fst $ treefoldr aggr1 (Void, -1) aggr2 (0,[]) t
  where
    aggr1 x (h,ts) = (Node ((fromIntegral x) / (fromIntegral $ h+1)) ts,h+1)

    aggr2 (t,h) (hm,ts) = (max h hm, t:ts)
```

6 Quad Trees

Molte tecniche sviluppate per la compressione di immagini si basano su una codifica ad albero chiamata "Quad Tree". Si codificano in questo modo immagini quadrate il cui lato sia una potenza di 2. Se l'immagine è omogenea (stesso colore) si codifica, indipendentemente dalle sue dimensioni, con una foglia contenente il colore. Se l'immagine è eterogenea allora si utilizza un nodo i cui figli contengono le codifiche dei quadranti superiore-sinistro, superiore-destro, inferiore-sinistro, inferiore-destro, rispettivamente.

Si definiscano i QuadTrees col seguente tipo di dato astratto (polimorfo)

```
data (Eq a,Show a) => QT a = C a | Q (QT a) (QT a) (QT a) (QT a)
  deriving (Eq,Show)
```

Con questa struttura si possono costruire termini che non corrispondono propriamente ad un QuadTree. Ad esempio

```
let u = C 1 in Q u u u u
```

non è la codifica di un'immagine, visto che dovrebbe essere semplicemente `C 1`. Chiamerò "termini di tipo QT" questi casi patologici, mentre QuadTrees quelli che corrispondono correttamente alla codifica di un'immagine. Possiamo subito notare dall'esempio di prima che partendo da 4 QuadTrees non si garantisce di costruire con il costruttore `Q` un QuadTree, ma solo un termine di tipo QT.

1. Si scriva una funzione `buildNSimplify` che dati 4 QuadTree costruisca un QuadTree la cui immagine codificata sia quella ottenuta dalle 4 immagini corrispondenti ai 4 QuadTree messe nei quadranti superiore-sinistro, superiore-destro, inferiore-sinistro, inferiore-destro, rispettivamente. (Attenzione che tutti sono e devono essere QuadTrees, non solo termini di tipo QT)

```
buildNSimplify (C z1) (C z2) (C z3) (C z4)
  | z1 == z2 && z1 == z3 && z1 == z4 = C z1
buildNSimplify q1 q2 q3 q4 = Q q1 q2 q3 q4
```

2. Si scriva una funzione `simplify` che dato un termine di tipo QT genera il QuadTree corrispondente.

```
simplify x@(C -)          = x
simplify (Q q1 q2 q3 q4) = buildNSimplify (simplify q1) (simplify q2)
                                          (simplify q3) (simplify q4)
```

18/12/2006

3. Si scriva una funzione `map` che data una funzione f e un QuadTree q determina il QuadTree che codifica l'immagine risultante dall'applicazione di f a tutti i pixel dell'immagine codificata da q.

```
map f = qm
  where
    qm (C x) = C $ f x
    qm (Q x1 x2 x3 x4) =
      buildNSimplify (qm x1) (qm x2) (qm x3) (qm x4)
```

4. Si scriva una funzione `howManyPixels` che dato un QuadTree determina il numero (minimo) di pixel di quell'immagine. Ad esempio

   `let z=C 0; u=C 1; q=Q z u u u in howManyPixels (Q q (C 0) (C 2) q)`

   restituisce 16.

```
howManyPixels (C _)           = 1
howManyPixels (Q q1 q2 q3 q4) = 4*(howManyPixels q1 ‘max‘
                                   howManyPixels q2 ‘max‘
                                   howManyPixels q3 ‘max‘
                                   howManyPixels q4)
```

13/04/2007

5. Si scriva una funzione `limitAll` che dato un colore c e una lista di QuadTrees costruisca la lista dei QuadTrees che codificano le immagini i cui pixels sono limitati al colore c (pixel originale se il colore è $< c$, c altrimenti).

```
limitAll n = Prelude.map (QT.map (min n))
```

18/12/2006

6. Si scriva una funzione `occurrencies` che dato un QuadTree ed un colore determina il numero (minimo) di pixel di quel colore. Ad esempio

   `let z=C 0; u=C 1; q=Q z u u u in occurrencies (Q q (C 0) (C 2) q) 0`

   restituisce 6 (visto che il QuadTree codifica almeno 16 pixel).

```
occurrencies q x = snd $ w q
  where
    w (C y) = (1,if x ==y then 1 else 0)
    w (Q q1 q2 q3 q4) = (4*n,k)
      where
        (n1,k1) = w q1
        (n2,k2) = w q2
        (n3,k3) = w q3
        (n4,k4) = w q4
        n = n1 `max` n2 `max` n3 `max` n4
        c m = ((n `div` m)*)
        k = c n1 k1 +
            c n2 k2 +
            c n3 k3 +
            c n4 k4
```

17/09/2007

7. Si scriva una funzione Haskell `difference` che dato un colore c ed un QuadTree q determina la differenza fra il numero di pixel dell'immagine codificata da q che hanno un colore maggiore di c e quelli minori di c. Ad esempio

```
let d = C 2; u = C 1; q = Q d u u u
in difference 1 (Q q (C 0) (C 3) q)
```

restituisce -4 (visto che il QuadTree codifica almeno 16 pixel).

```
difference x = snd . fold f g
  where
    g y = (1,if y>x then 1 else (-1))
    f (n1,k1) (n2,k2) (n3,k3) (n4,k4) = (4*n,k)
      where
        n = n1 `max` n2 `max` n3 `max` n4
        c m = ((n `div` m)*)
        k = c n1 k1 + c n2 k2 + c n3 k3 + c n4 k4
```

13/07/2007

8. Si scriva una funzione Haskell `overColor` che dato un colore c ed un QuadTree q determina il numero (minimo) di pixel dell'immagine codificata da q che hanno un colore maggiore di c. Ad esempio

```
let d = C 2; u = C 1; q = Q d u u u
in overColor 1 (Q q (C 0) (C 3) q)
```

restituisce 6 (visto che il QuadTree codifica almeno 16 pixel).

```
overColor q x = snd $ w q
  where
    w (C y) = (1,if y>x then 1 else 0)
    w (Q q1 q2 q3 q4) = (4*n,k)
      where
        (n1,k1) = w q1
        (n2,k2) = w q2
        (n3,k3) = w q3
        (n4,k4) = w q4
        n = n1 `max` n2 `max` n3 `max` n4
        c m = ((n `div` m)*)
        k = c n1 k1 + c n2 k2 + c n3 k3 + c n4 k4
```

9. Si scriva una generalizzazione della funzione `foldr` delle liste per i termini di tipo QT che abbia il seguente tipo:

   `fold :: (Eq a, Show a) => (b->b->b->b->b) -> (a->b) -> QT a -> b`

```
fold :: (Eq a, Show a) =>
        (b -> b -> b -> b -> b) -> (a -> b) -> QT a -> b
fold f g (C x)           = g x
fold f g (Q q1 q2 q3 q4) = f (fold f g q1) (fold f g q2)
                             (fold f g q3) (fold f g q4)
```

10. Si scriva una funzione `height` che dato un QuadTree ne determina l'altezza usando opportunamente `fold`.

```
height = fold f g
  where
    g _ = 0
    f x1 x2 x3 x4 = 1 + x1 `max` x2 `max` x3 `max` x4
```

11. Si scriva una funzione `length` che dato un QuadTree ne determina il numero di nodi usando opportunamente `fold`.

```
length = fold f g
  where
    g _ = 1
    f x1 x2 x3 x4 = 1 + x1 + x2 + x3 + x4
```

12. Si riscriva la funzione `simplify` dell'Esercizio 2 usando opportunamente `fold`.

```
simplify = fold buildNSimplify C
```

13. Si riscriva la funzione `map` dell'Esercizio 3 usando opportunamente `fold`.

```
map f = fold buildNSimplify (C . f)
```

12/01/2007

14. Si scrivano due funzioni `flipHorizontal`/`flipVertical` che costruiscono il QuadTree dell'immagine simmetrica rispetto all'asse orizzontale/verticale.

```
flipHorizontal = fold fh C
  where
    fh x1 x2 x3 x4 = Q x3 x4 x1 x2

flipVertical = fold fv C
  where
    fv x1 x2 x3 x4 = Q x2 x1 x4 x3
```

dentro 17/09/0

15. Si scrivano tre funzioni `rotate90Right`, `rotate90Left` e `rotate180` che costruiscono il QuadTree dell'immagine ruotata di $-\pi/2$, $+\pi/2$ e π.

```
rotate90Right = fold rr C
  where
    rr x1 x2 x3 x4 = Q x3 x1 x4 x2

rotate90Left = fold rl C
  where
    rl x1 x2 x3 x4 = Q x2 x4 x1 x3

rotate180 = fold rot C
  where
    rot x1 x2 x3 x4 = Q x4 x3 x2 x1
```

23/03/2007

16. Si scrivano tre predicati `isHorizontalSymmetric`, `isVerticalSymmetric` e `isCenterSymmetric` che determinano se un QuadTree codifica un'immagine simmetrica rispetto all'asse orizzontale, all'asse verticale o al centro.

```
isHorizontalSymmetric q = q == (flipHorizontal q)

isVerticalSymmetric q = q == (flipVertical q)

isCenterSymmetric q = q == rotate180 q
```

13/07/2007

17. Si scriva un predicato `elem_or_mele` che dati un QuadTree t e una lista di QuadTrees ts determina se t, o il QuadTree che codifica l'immagine di t ribaltata rispetto all'asse orizzontale, sono elementi della lista ts.

```
elem_or_mele x xs = any p xs
  where
    p y = y==x || y==flipHorizontal x
```

17/09/2007

18. Si scriva un predicato `isRotatedIn` che dati un QuadTree t e una lista di QuadTrees ts determina se uno dei QuadTrees che codificano l'immagine di t ruotata di 0, 90, 180 o 270 gradi è un elemento della lista ts.

```
isRotatedIn x xs = any p xs
  where
    p y = y==x || y==x1 || y==x2 || y==x3
    x1 = rotate90Right x
    x2 = rotate90Right x1
    x3 = rotate90Right x2
```

con `rotate90Right` soluzione dell'Esercizio apposito dell'eserciziario.

19. Si riscriva la funzione `howManyPixels` dell'Esercizio 4 usando opportunamente `fold`.

```
howManyPixels = fold f g
  where
    f a b c d = 4*(a `max` b `max` c `max` d)
    g _ = 1
```

18/12/2006

20. Si riscriva la funzione `occurrencies` dell'Esercizio 6 usando opportunamente `fold`.

```
occurrencies q x = snd $ fold f g q
  where
    g y = (1,if x ==y then 1 else 0)
    f (n1,k1) (n2,k2) (n3,k3) (n4,k4) = (4*n,k)
      where
        n = n1 `max` n2 `max` n3 `max` n4
        c m = ((n `div` m)*)
        k = c n1 k1 + c n2 k2 + c n3 k3 + c n4 k4
```

12/01/2007 e (

21. Si scriva una funzione `zipWith` per QuadTrees che, analogamente alla `zipWith` per le liste, data un'operazione binaria $\oplus$ e due QuadTrees q_1 e q_2 costruisce il QuadTree che codifica l'immagine risultante dall'applicazione di $\oplus$ a tutti i pixel della stessa posizione nelle immagini codificate da q_1 e q_2.

```
zipWith f = qop
  where
    qop (C x)          (C y)          = C (f x y)
    qop q1@(Q _ _ _ _) q2@(Q _ _ _ _) = aux q1 q2
    qop q1@(Q _ _ _ _) y@(C _)        = aux q1 (Q y y y y)
    qop x@(C _)        q2@(Q _ _ _ _) = aux (Q x x x x) q2

    aux (Q x1 x2 x3 x4) (Q y1 y2 y3 y4) =
      buildNSimplify (qop x1 y1) (qop x2 y2) (qop x3 y3) (qop x4 y4)
```

02/02/2009

22. Si scriva una funzione Haskell `insertPict` che dati i QuadTrees di due immagini q_t, q_f ed un QuadTree "maschera" a valori booleani, costruisce il QuadTree dell'immagine risultante mantenendo i pixel di q_t in corrispondenza del valore `True` (della maschera) oppure di q_f in corrispondenza del valore `False`.

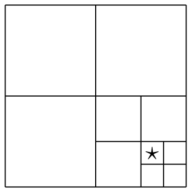
```
insertPict p1 p2 mask = zipWith aggr2 (zipWith aggr1 p1 mask) p2
  where
    aggr1 _ False = Nothing
    aggr1 x True  = Just x

    aggr2 Nothing  x = x
    aggr2 (Just x) _ = x
```

con `zipWith` soluzione dell'apposito esercizio dell'eserciziario.

01/07/2009

23. Si scriva una funzione Haskell `insertLogo` che dati i QuadTrees di due immagini q_l, q_p ed un QuadTree "maschera" a valori booleani, costruisce il QuadTree dell'immagine risultante inserendo la figura q_l all'interno del quadrante marcato $\star$ di q_p scegliendo i pixel di q_l o q_p in corrispondenza del valore True o False della maschera.



```
insertLogo logo mask (Q q1 q2 q3 (Q r1 r2 r3 (Q s1 s2 s3 s4)))
 = buildNSimplify q1 q2 q3 qq
   where
     qq = buildNSimplify r1 r2 r3 rr
     rr = buildNSimplify xx s2 s3 s4
     xx = insertPict logo mask s1
insertLogo logo mask (Q q1 q2 q3 (Q r1 r2 r3 s@(C _)))
 = insertLogo logo mask (Q q1 q2 q3 (Q r1 r2 r3 (Q s s s s)))
insertLogo logo mask (Q q1 q2 q3 r@(C _))
 = insertLogo logo mask (Q q1 q2 q3 (Q r r r (Q r r r r)))
insertLogo logo mask r@(C _)
 = insertLogo logo mask (Q r r r (Q r r r (Q r r r r)))
```

con `buildNSimplify` e `insertPict` soluzioni degli appositi esercizi dell'eserciziario.

26/02/2009

24. Si scriva una funzione Haskell `commonPoints` che data una lista non-vuota di QuadTrees l costruisce il QuadTree "maschera", a valori booleani, che ha "un pixel" a `True` se nella medesima posizione tutte le immagini di l hanno pixels uguali, `False` altrimenti.

```
commonPoints (q:qs) = map maybe2bool $ foldl aggr jq qs
  where
    maybe2bool = (Nothing/=)

    aggr = zipWith cmn
      where
        cmn j@(Just y) x | x==y = j
        cmn _ _                 = Nothing

    jq = map Just q
```

23/03/2007

25. Si scriva un predicato `framed` che dato un predicato sui colori p ed un QuadTree determina se il bordo esterno dell'immagine codificata è tutto composto da pixels che soddisfano p.

```
data Check = All | UL | U | UR | L | R | DL | D | DR

framed :: (Show t, Eq t) => (t -> Bool) -> QT t -> Bool
framed p = fr All
  where
    fr _ (C y) = p y
    fr w (Q x1 x2 x3 x4) =
      case w of
        All-> fr UL x1 && fr UR x2 && fr DL x3 && fr DR x4
        UL -> fr UL x1 && fr U  x2 && fr L  x3
        UR -> fr U  x1 && fr UR x2 && fr R  x4
        DL -> fr L  x1 && fr DL x3 && fr D  x4
```

```
          DR -> fr R  x2 && fr D  x3 && fr DR x4
          U  -> fr U  x1 && fr U  x2
          L  -> fr L  x1 && fr L  x3
          R  -> fr R  x2 && fr R  x4
          D  -> fr D  x3 && fr D  x4
```

13/04/2007

26. Si scriva una funzione `frame` che dato un QuadTree restituisca `Just c` se il bordo esterno dell'immagine codificata è tutto composto da pixels di colore `c` (`Nothing` altrimenti).

```
data Check = All | UL | U | UR | L | R | DL | D | DR

frame = fr All
  where
    a (Just x) (Just y)
      | x==y            = Just x
    a _        _        = Nothing

    fr _ (C y) = Just y
    fr w (Q x1 x2 x3 x4) =
      case w of
        All-> fr UL x1 `a` fr UR x2 `a` fr DL x3 `a` fr DR x4
        UL -> fr UL x1 `a` fr U  x2 `a` fr L  x3
        UR -> fr U  x1 `a` fr UR x2 `a` fr R  x4
        DL -> fr L  x1 `a` fr DL x3 `a` fr D  x4
        DR -> fr R  x2 `a` fr D  x3 `a` fr DR x4
        U  -> fr U  x1 `a` fr U  x2
        L  -> fr L  x1 `a` fr L  x3
        R  -> fr R  x2 `a` fr R  x4
        D  -> fr D  x3 `a` fr D  x4
```

7 Matrici mediante Quad Trees

Grazie ai Quad Trees introdotti nella sezione precedente si possono implementare certe operazioni matriciali, nel caso dei linguaggi funzionali puri ovviamente, in modo molto più efficiente.

Si implementino matrici $2^n \times 2^n$ utilizzando il seguente tipo di dato astratto (polimorfo)

```
data (Eq a,Num a,Show a) => Mat a = Mat {
  nexp :: Int,
  mat :: QT a
  }
  deriving (Eq,Show)
```

dove nel campo `mat` non metteremo mai solo "termini di tipo QT" ma QuadTrees "propri".

1. Si scriva un predicato `lowertriangular` che determina se una matrice è triangolare inferiore.

   Attenti a cosa devono restituire
   `lowertriangular $ Mat 0 (C 2)` e `lowertriangular $ Mat 1 (C 2)`.

```
lowertriangular m = lt (nexp m) (mat m)
  where
    lt k (C x) = k==0 || x==0
    lt k (Q q1 q2 _ q4) = q2 == C 0 && lt m q1 && lt m q4
      where m=k-1
```

2. Si scriva un predicato `uppertriangular` che determina se una matrice è triangolare superiore.

```
uppertriangular m = ut (nexp m) (mat m)
  where
    ut k (C x) = k==0 || x==0
    ut k (Q q1 _ q3 q4) = q3 == C 0 && ut m q1 && ut m q4
      where m=k-1
```

3. Si scriva un predicato `diagonal` che determina se una matrice è diagonale.

```
diagonal m = diag (nexp m) (mat m)
  where
    diag k (C x) = k==0 || x==0
    diag k (Q q1 q2 q3 q4) = q2 == C 0 && q3 == C 0 && diag m q1 && diag m q4
      where m=k-1
```

4. Si scriva una funzione `matSum` che date 2 matrici calcoli la matrice somma.

```
matSum (Mat n1 m1) (Mat n2 m2)
 | n1==n2 = zipWith (+) m1 m2
 | otherwise = error "matMul:␣incompatible␣matrices"
```

01/02/2010

5. Si scriva una funzione `matMul` che date 2 matrici calcoli la matrice prodotto.

```
matMul (Mat n1 m1) (Mat n2 m2)
 | n1==n2 = mul m1 m2
 | otherwise = error "matMul:␣incompatible␣matrices"
  where
    mul (C x) (C y) = C $ x*y
    mul (Q a11 a12 a21 a22) (Q b11 b12 b21 b22) =
     Q ((a11 `mul` b11) `sum` (a12 `mul` b21))
       ((a11 `mul` b12) `sum` (a12 `mul` b22))
       ((a21 `mul` b11) `sum` (a22 `mul` b21))
       ((a21 `mul` b12) `sum` (a22 `mul` b22))
    mul q@(Q _ _ _ _) x@(C _) = mul q (Q x x x x)
    mul x@(C _) q@(Q _ _ _ _) = mul (Q x x x x) q

    sum = zipWith (+)
```

6. Si scriva una funzione `zong` che, dati due valori x, y e una matrice A, calcola la matrice $xA - yI$ (dove I è la matrice unitaria della giusta dimensione).

```
zong x y m@(Mat n q) = Mat n $ zipWith (-) q1 q2
  where
    q1 = map (x*) q
```

```
    q2 = yId n

    yId 0 = C y
    yId m = Q (yId k) (C 0) (C 0) (yId k)
      where
        k = m-1
```

7. Si scriva una funzione `f` che, dati un vettore v e una matrice A, calcola lo scalare vAv^T. Si scelga la struttura dati per i vettori nel modo che si ritiene più opportuno.

> SISTEMARE
>
> la matrice $xvA - yv = v(xA - yI)$ (dove I è la matrice unitaria della giusta dimensione).
>
> `vectMatMultiply` che, dati un vettore ed una matrice di dimensioni compatibili, calcola il vettore prodotto.
>
> ```
> tong :: (Num a) => a -> a -> Vect a -> Mat a -> Vect a
> tong x y b m = vectZipWith f (vectMatMultiply b m) b
>   where
>     f a b = x*a-y*b
> ```
>
> dove ... SISTEMARE

8. Si scriva una funzione `colSums` che data una matrice calcola il vettore delle somme delle colonne della matrice.

   Ad esempio

   ```
   let z=C 0; u=C 1; d=C 2 in colsums $ Mat 3 $ Q (Q u d d u) z u d
   ```

   deve produrre [10,10,10,10,8,8,8,8].

9. Si scriva una funzione `rowSums` che data una matrice calcola il vettore delle somme delle righe della matrice.

10. Si scriva una funzione `colMinMax` che data una matrice calcola il vettore delle coppie (minimo,massimo) delle colonne della matrice.

11. Si scriva una funzione `colVar` che, data una matrice, calcola il vettore delle variazioni (= massimo - minimo) delle colonne della matrice.

> ```
> colVar :: (Ord a,Num a) => Mat a -> [a]
> colVar = map (\(mn,mx)->mx-mn) . colMinMax
> ```

12. Si scriva una funzione `colAltSums` che calcola il vettore delle somme a segni alternati delle colonne della matrice. Detto $s_j = \sum_{i=1}^{n}(-1)^{i+1}a_{ij}$, `colaltsums`$\left(\begin{pmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{pmatrix}\right) = \begin{pmatrix} s_1 & \dots & s_m \end{pmatrix}$

> Versione non-fold da fare

13. Si scriva una funzione `transpose` che calcola la matrice trasposta.

```
transpose = lift1 (fold tr C)
  where
    tr q1 q2 q3 q4 = Q q1 q3 q2 q4

lift1 f (Mat n m) = Mat n $ f m
```

14. Si scriva un predicato `isSymmetric` che determina se una matrice è simmetrica.

```
isSymmetric m = m == transpose m
```

15. Si scriva una funzione `foldMat` con tipo

```
foldMat :: (Num a) =>
  (Int -> b -> b -> b -> b -> b) ->
  (Int -> a -> b) -> Mat a -> b
```

```
foldMat f g (Mat n q) = fld n q
  where
    fld k (C x)          = g k x
    fld k (Q q1 q2 q3 q4) = f k (fld m q1) (fld m q2)
                                (fld m q3) (fld m q4)
      where m = k-1
```

begin DRAFT

16.

17. Si scriva una funzione `zipContWith`

```
type Cont a = [a] -> [a]

zipContWith :: (a -> b -> c) -> Cont a -> Cont b -> Cont c
```

che FINIRE

```
zipContWith aggr f g = h
  where
    h xs = fold2 aggr xs (f []) (g [])

    fold2 _ xs [] [] = xs
    fold2 f xs (y:ys) (z:zs) = f y z : fold2 f xs ys zs
```

18. Si scriva una funzione `base` con tipo

```
base :: (Num a,Num b) => (a -> b -> c) -> Int -> b -> Cont c
```

che FINIRE

```
base g n x = f n 1
  where
    f 0 m = (g m x:)
    f k m = h . h
      where h = f (k-1) (2*m)
```

19. Si riscrivano le funzioni degli Esercizi 8 e 9 usando `foldMat`, `zipContWith` e `base`.

```
colSums = ($[]) . foldMat aggr (base (*))
  where
    aggr _ f1 f2 f3 f4 = zipContWith (+) (f1 . f2) (f3 . f4)

rowSums = ($[]) . foldMat aggr (base (*))
  where
    aggr _ f1 f2 f3 f4 = zipContWith (+) (f1 . f3) (f2 . f4)
```

20. Si riscriva la funzione dell'Esercizio 10 usando `foldMat`, `zipContWith` e `base`.

```
colMinMax = ($[]) . foldMat aggr basePair
  where
    aggr _ f1 f2 f3 f4 = zipContWith minmax (f1 . f2) (f3 . f4)
    minmax (x1,y1) (x2,y2) = (min x1 x2,max y1 y2)


basePair n x = f n x
  where
    f 0 x = ((x,x):)
    f m x = h . h
      where h = f (m-1) x
```

21. Si riscriva la funzione dell'Esercizio 12 usando `foldMat`, `zipContWith` e `base`.

```
colAltSums = ($[]) . foldMat aggr (base (\_ _->0))
  where
    aggr 1 f1 f2 f3 f4 = zipContWith (-) (f1 . f2) (f3 . f4)
    aggr _ f1 f2 f3 f4 = zipContWith (+) (f1 . f2) (f3 . f4)
```

end
DRAFT