Esercizi di Programmazione Haskell

Marco Comini

19 ottobre 2009

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!

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!.
2. Si scriva la funzione $\binom{n}{k}$, combinazioni di k elementi su n.
3. Si scriva una funzione che calcoli una lista con tutte le combinazioni su n elementi. Si usi opportunamente

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

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

3. Scrivere il QuickSort (polimorfo).

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)

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.

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 una fold è difficile

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). Difficile

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).

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

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`$\left(\begin{pmatrix} a_{11} & \dots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nm} \end{pmatrix}\right) = \begin{pmatrix} s_1 & \dots & s_m \end{pmatrix}$

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.

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`.

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

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

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`.

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

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

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.

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 a (BST a) (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.

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

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

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

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

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

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

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.

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]

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.

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.

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).
    ```

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).

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

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

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

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

17. Sia $\max_n t$ il più grande dei valori strettamente minori di n di un BST t (se ne esistono), si scriva una funzione Haskell `boundedMaximum` che dato un numero n e una lista l di BST determina la lista dei $\max_n t$ che esistono, al variare di t in l.

18. 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.

19. 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.

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

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

22. 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.

23. 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).

24. 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.

25. 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.

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
```

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.

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

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.

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

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

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

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

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

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

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

12. 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.

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)

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

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.

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.

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).

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).

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).

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).

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
```

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

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

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

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

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

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

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.

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.

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.

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

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

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.

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`.

23. 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.

24. 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.

25. 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).