Programmazione Funzionale

Esercitazione 13 – Alberi Binari

Ricordiamo la definizione del tipo dei alberi binari e del tipo delle direzioni che usiamo:

```
type 'a btree = Empty | Tr of 'a * 'a btree * 'a btree;;
type direction = Left | Right;;
```

Chiamiamo *indirizzo* o *posizione* una lista di tipo `direction list`.

Esercizio 1. Vogliamo definire diverse funzione di base su i alberi binari.

1. Definire `size : 'a btree -> int` una funzione che restituisce la quantità di elementi che contiene un albero binario.
2. Ricordiamo che una foglia e un albero della forma `Tr(a , Empty, Empty)`. Definire `isLeaf : 'a btree -> bool` una funzione che prende un albero binario e restituisce `true` se l'albero in entrata e una foglia, altrimenti la funzione restituisce `false`.

   Dare due versioni della funzione

   - Una senza usare `size`.
   - Una che usa la funzione `size`.
3. Definire `leaves : 'a btree -> 'a btree list` una funzione che prende un albero binario e restituisce la lista delle sue foglie.
4. Definire una funzione `mirror` che prende un albero binario `t` e inverte i figli destro e sinistro di ogni sotto albero di `t`.

   Ad esempio, avendo definito `let leaf a = Tr(a, Empty, Empty)` `mirror Tr(a , leaf(a) , Tr(b,leaf(a),Empty))` restituisce `Tr(a, Tr(b , Empty, leaf (a)) , leaf(a))`.

Esercizio 2. Dato un albero binario vogliamo fare diverse cose;

1. Definire la funzione `delete: 'a btree -> 'a -> 'a btree` che prende un albero e un elemento x e sostituisce tutti i sotto alberi di radice x trovati con l'albero vuoto `Empty`.
2. Definire `deleteFirst: 'a btree -> 'a -> 'a btree` un funzione definita con il backtracking che prende un albero un elemento x e sostituisce il primo sotto albero di radice x trovato con l'albero vuoto `Empty`.
3. Usando liste di elementi di tipo `direction` possiamo trovare dei sottoalberi senza ambiguita.

   Definire `find: direction list -> 'a btree -> 'a` che prende una lista di direzioni e un albero binario e restituisce il sottoalbero che si trova in quella posizione. Se la posizione non e raggiungibile la funzione sollevera un eccezione.
4. Pendendo spunto dalla funzione `find` definire `deleteAtpos: direction list -> 'a btree -> 'a btree` una funzione che prende una posizione e un albero binario `t` e cancella il sottoalbero di `t` che si trova in quella posizione.

Esercizio 3. Vogliamo implementare una funzione che dato un albero binario e una lista di direzione (e.g. posizione) restituisce l'elemento trovato in quella positione nel albero. Vogliamo poi generalizzare questa funzione per funzione su una lista di posizioni.

1. Implementare una funzione `find: direction list -> 'a btree -> 'a` che prende una lista di direzioni e un albero binario e restituisce l'elemento che si trova in quella posizione.
2. Vogliamo generalizzare la funzione precedente; definire una funzione `findList : (direction list) list -> 'a btree -> 'a` che prende una lista di posizione e un albero binario e restituisce la lista degli elementi trovati in quelle posizione.

   - Definire una versione senza usare `List.map`.
   - Definire una versione mediante `List.map`

Esercizio 4. Vogliamo definire una funzione che trova le posizione in un albero binario in cui occorono l'elemento dato `x`.

1. Definire una funzione `findFirst : 'a -> 'a btree -> direction list` che dato un elemento `x` e un albero `t` restituisce l'indirizzo della prima occorenza trovata di `x` in `t`.

   Se nessuna occorenza e trovata la funzione sollevera un eccezione.

2. Vogliamo modificare `findFirst` con accumulatore per definire una funzione `findAll : 'a -> 'a btree -> direction list list` che prende un elemento `x` e un albero `t` e restituisce la lista delle posizione in cui occore `x` in `t`.

   `findAll` non deve mai sollevare un eccezione.

3. Definire `substi : direction list -> 'a -> 'a btree -> 'a btree` che prende una posizione `dl : direction list` un elemento `x` e un albero binario `t` e restituisce `t` in cui l'elemento in posizione `dl` e stato sostituito con `x`.

4. Definire una versione generalizzata della funzione precedenete. Definire `substi_list : (direction list) list -> 'a -> 'a btree -> 'a btree` che prende una lista di posizione `plist` un elemento `x` e un albero binario `t` e sostituisce tutti i elementi trovati in una posizione di `plist` con l'elemento `x`.

5. Definire una funzione di sostituzione `substitution : 'a -> 'a -> 'a btree -> 'a btree` che prende un elemento `a`, un elemento `x` e un albero `t` e sostituisce tutte le occorenze di `a` in `t` con `x`.

   Usare le funzione `findAll` e `substi_list`.

Esercizio 5. Consideriamo di aver dichiarato;

```
type espr = I of Int | Plus of espr * espr | Mult of espr * espr;;
```

1. Definire una funzione `translate : espr -> string btree` che prende un espressione e restituisce l'albero binario corrispondente.

   Ad esempio, `translate I 3` restituisce `Tr ( "3" , Empty, Empty)`. `translate Plus(I 3, I 2)` restituisce `Tr ("Plus" , Tr("3" , Empty, Empty) , Tr("2" , Empty,Empty))`.

2. Definire la funzione inversa, `cotranslate : string btree -> espr` che prende un albero di stringhe e restituisce un espressione se e possibile. Se non e possibile la funzione sollevera un eccezione.