Programmazione Funzionale

Esercitazione 14 – Alberi n–ary

In questi esercizi, albero significa albero n–ario e assumiamo di aver dichiarato;

```
let 'a tree = Tr of 'a * ('a tree list);;
```

In questo contesto, una *posizione* o *indirizzo* e una sequenza di interi. Un *foglia* e un albero della forma `Tr(a , [])`.

Esercizio 1. Definiamo delle funzioni di base sui i alberi.

1. Definire una funzione `size : 'a tree -> int` che prende un albero e restituisce la quantità di elementi che contiene.
2. Definire una funzione `sumTree : int tree -> int` che prende un albero di interi e restituisce la somma dei suoi elementi.
3. Definire `leaf : 'a -> 'a tree` che prende une elemento `x` e restituisce l'albero fatto di solo l'elemento `x` (quest'albero è una foglia).
4. Definire `isLeaf: 'a -> 'a tree` una funzione che prende un albero `t` e restituisce `true` se `t` e una foglia, altrimenti la funzione restituisce `false`.

Esercizio 2. Vogliamo implementare algoritmi di ricerca di un cammino in un albero, anche cerchando cammini con una certa proprietà.

1. Mediante backtracking definire una funzione `find : 'a -> 'a tree -> 'a list` che prende un elemento `a` e un albero `t` e restituisce un cammino sotto forma di lista di tipo `'a list` della radice di `t` a l'elemento `a`. Se nessun cammino esiste la funzione restituisce un eccezione.

   Ad esempio `find 1 Tr ( 2 , [Tr (3 , [leaf(1)]) ; leaf(3) ; Tr (1 , [leaf(1) ; leaf(2)] )])` restituisce `[2;3;1]`.

2. La funzione precedente restituisce una lista `'a list` per identificare il cammino, questo crea ambiguita. Vogliamo modificare la funzione precedente tale che restituisce una lista `int list` di interi.

   Definire `findPos : 'a -> 'a tree -> int list` che prende un elemento `a` e un albero `t` e restituisce un cammino sotto forma di lista di tipo `int list` della radice di `t` a l'elemento `a`. Se nessun cammino esiste la funzione restituisce un eccezione.

   Ad esempio `findPos 1 Tr ( 2 , [Tr (3 , [leaf(1)]) ; leaf(3) ; Tr (1 , [leaf(1) ; leaf(2)] )])` restituisce `[1;1]`.

3. Definire una funzione `findPosd : 'a -> 'a tree -> (int * 'a) list` che prende un elemento `a` e un albero `t` e restituisce un cammino sotto forma di lista di tipo `(int * 'a) list` della radice di `t` a l'elemento `a`. Se nessun cammino esiste la funzione restituisce un eccezione.

   Ad esempio `findPos 1 Tr ( 2 , [Tr (3 , [leaf(1)]) ; leaf(3) ; Tr (2 , [leaf(1) ; leaf(2)] )])` restituisce `[(1,3);(1,1)]`.

4. Vogliamo ora definire una funzione che trova un cammino sotto forma di lista `(int * 'a) list` tale che la somma dei elementi attraversati vale 5.

   Adattendo la funzione `findPosd` definire una funzione `findCond : 'a -> 'a tree -> (int * 'a) list` che prende un elemento `a` e un albero `t` e restituisce un cammino sotto forma di lista di tipo `(int * 'a) list` della radice di `t` a l'elemento `a` tale che la somma dei elementi attraversati vale 5.

   - Definire una funzione `accept : (int * 'a list) -> bool`.
   - Definire una funzione `reject : (int * 'a list) -> bool`.
   - Implementare `findCond` usando il backtracking.

   Ad esempio `findCond 1 Tr ( 2 , [Tr (3 , [leaf(1)]) ; leaf(3) ; Tr (2 , [leaf(1) ; leaf(2)] )])` restituisce `[(3,2);(1,1)]`.

Esercizio 3. Vogliamo implementare un algoritmo che ricerca un elemento e restituisce i possibili cammini.

1. Definire `findPos : 'a -> 'a tree -> int list` che prende un elemento `a` e un albero `t` e restituisce un cammino sotto forma di lista di tipo `int list` della radice di `t` a l'elemento `a`. Se nessun cammino esiste la funzione restituisce un eccezione.

   Ad esempio `findPos 1 Tr ( 2 , [Tr (3 , [leaf(1)]) ; leaf(3) ; Tr (1 , [leaf(1) ; leaf(2)] )])` restituisce `[1;1]`.

2. Definire `findAll : 'a -> 'a tree -> int list list` che prende un elemento `a` e un albero `t` e restituisce un lista di cammini `int list list` della radice di `t` a l'elemento `a`. La funzione non solleva mai eccezione.

   Ad esempio `findAll 1 Tr ( 2 , [Tr (3 , [leaf(1)]) ; leaf(3) ; Tr (1 , [leaf(1) ; leaf(2)] )])` restituisce `[[1;1] ; [3;1]]`.

Esercizio 4. Vogliamo definire una funzione che aggiunge a un sotto–albero dei figli, poi vogliamo definire un operazione di sostituzione sui alberi.

1. Definire una funzione `find : int list -> 'a tree -> 'a tree` che prende una posizione `p: int list` e un albero `t` e restituisce il sotto albero in posizione `p` di `t`. Se non esiste la funzione sollevera un eccezione.
2. Definire una funzione `add : int list -> 'a tree list -> 'a tree -> 'a tree` che prende una posizione `p` un lista di alberi `tlist` e un albero `t` e restituisce l'albero `t` in cui il sottoalbero in position `p` di `t` sono venuti aggiunti come figli i alberi di `tlist`.

   Ad esempio `add [2;1] [leaf(1) ; leaf(3)] Tr(1 , [leaf(3) ; Tr (1 , [ leaf(8)] ) ] )` restituisce `Tr(1 , [leaf(3) ; Tr (1 , [ Tr (8 , [leaf(1) ; leaf(3)])])])` .
3. Definire una funzione `substi : int list -> 'a tree -> 'a tree` che prende una posizione `p` una albero `subs` e un albero `t` e restituisce l'albero `t` in cui il sottoalbero di `t` in posizione `p` e stato sostituito con l'albero `subs`.