Esercizi proposti – 8

INDICAZIONI DI CARATTERE GENERALE:

- Quando scrivete una funzione che ne utilizza una ausiliaria, chiedetevi sempre se l'ausiliaria è necessaria, ricordando che è utile quando: (a) servono dei parametri in più, oppure (b) serve a risparmiare parametri (strutture dati grosse che verrebbero copiate ad ogni chiamata ricorsiva).
- Nella rappresentazione degli alberi binari si decide di introdurre l'albero vuoto per comodità. Ma una volta che c'è, va considerato sempre, nel senso che se si definisce una funzione sugli alberi binari, deve – normalmente – essere definita anche per `Empty`. A meno che non si voglia davvero che sia indefinita per l'albero vuoto. Ad esempio una funzione che riporti la radice di un albero deve essere indefinita per `Empty`, come `List.hd` è indefinita per la lista vuota.
- Leggere con attenzione il tipo delle funzioni richieste e prestare attenzione a quando si parla di "foglie" (si intende proprio "foglie" e non nodi qualsiasi).

---

1. Risolvere i problemi seguenti su espressioni rappresentate come alberi binari, mediante la dichiarazione di tipo

```
type expr =
    Int of int
  | Var of string
  | Sum of expr * expr
  | Diff of expr * expr
  | Mult of expr * expr
  | Div of expr * expr
```

   (a) Scrivere una funzione `subexpr: expr -> expr -> bool` che, date due espressioni aritmetiche E_1 e E_2 determini se E_2 è una sottoespressione di E_1.
   Ad esempio, le sottoespressioni di $5 + (3 \times 8)$ sono: $5 + (3 \times 8)$ stessa (qualsiasi espressione è una sottoespressione di se stessa), 5, 3×8, 3 e 8.

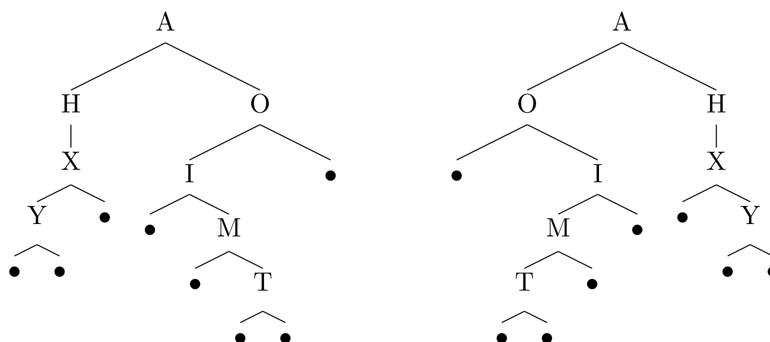
   (b) Scrivere una funzione `subst_in_expr: expr -> string -> expr -> expr` che, data un'espressione E, il nome di una variabile x e un'espressione E', riporti l'espressione che si ottiene da E sostituendo ogni occorrenza di x con E'.

2. Data la dichiarazione di tipo per la rappresentazione di alberi binari:

```
type 'a tree = Empty | Tr of 'a * 'a tree * 'a tree
```

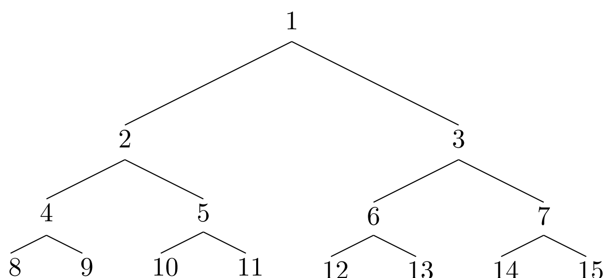
   definire le funzioni seguenti:

   (a) `reflect : 'a tree -> 'a tree`. Applicata a un albero binario, ne costruisce l'immagine riflessa. Ad esempio, i due alberi sotto rappresentati sono uno l'immagine riflessa dell'altro (• rappresenta l'albero vuoto).



(b) `fulltree` : `int -> int tree`. La funzione, applicata a un intero n, riporta un albero binario completo di altezza n, con i nodi etichettati da interi come segue: la radice è etichettata da 1, i figli di un nodo etichettato da k sono etichettati da $2k$ e $2k + 1$.

Ad esempio, `fulltree 4` costruisce l'albero seguente:



(Un albero binario è completo se ogni nodo interno ha esattamente due figli; in un albero binario completo di altezza n, ogni ramo ha esattamente n nodi)

(c) `balanced`: `'a tree -> bool`, determina se un albero è bilanciato (un albero è bilanciato se per ogni nodo n, le altezze dei sottoalberi sinistro e destro di n differiscono al massimo di 1).

(d) `preorder`, `postorder`, `inorder`, tutte di tipo `'a tree -> 'a list`. Dato un albero t, le funzioni riportano la lista dei nodi di t, nell'ordine in cui sarebbero visitate secondo gli algoritmi di visita, rispettivamente, in preordine, postordine e simmetrica.

(e) `balpreorder` e `balinorder`, entrambe di tipo `'a list -> 'a tree`. Data una lista lst, costruiscono un albero bilanciato con nodi etichettati da elementi di lst, in modo tale che

```
preorder (balpreorder lst) = lst
inorder  (balinorder lst)  = lst
```

(utilizzare `take` e `drop`). Ad esempio:

```
balpreorder [1;2;3;4;5] =

      Tr (1, Tr (2, Empty,
                    Tr (3, Empty, Empty)),
             Tr (4, Empty,
```

```
                    Tr (5, Empty, Empty)))

balinorder [1;2;3;4;5] =

        Tr (3, Tr (2, Tr (1, Empty, Empty),
                      Empty),
               Tr (5, Tr (4, Empty, Empty),
                      Empty))
```

3. `foglie_in_lista: 'a list -> 'a tree -> bool`, che, data una list `lst` e un albero binario `t`, determini se ogni *foglia* di `t` appartiene a `lst`. (Una foglia è rappresentata da un valore della forma `Tr(x,Empty,Empty)`).

4. `num_foglie: 'a tree -> int` che, applicata a un albero binario, riporti il numero di foglie dell'albero.

5. Una lista di booleani L può determinare un sottoalbero di un albero binario: quello che si ottiene, a partire dalla radice, scendendo al figlio sinistro per ogni `true` nella lista, al figlio destro per ogni `false`. Se la lista è più lunga del ramo che si ottiene da essa, allora il sottoalbero determinato da L è l'albero vuoto.

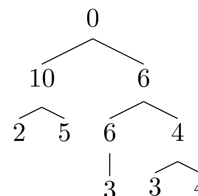
   Si consideri ad esempio l'albero completo rappresentato sopra per l'esercizio 2b. La lista `[true;false;false]` determina il sottoalbero che ha radice 11. La lista `[false;true]` determina il sottoalbero con radice 6. Liste con più di 3 elementi determinano l'albero vuoto.

   Scrivere una funzione `segui_bool: bool list -> 'a tree -> 'a` che, data una lista L di booleani e un albero binario T, riporti la radice del sottoalbero di T determinato da L, se questo non è vuoto, un errore altrimenti.

   Ad esempio, la funzione, applicata alla lista `[true;false;false]` e all'albero rappresentato per l'esercizio 2b riporterà 11. Applicata alla lista `[false;true]` e allo stesso albero, riporterà 6. Riporterà un errore se la lista ha più di 3 elementi.

6. Se T e' un albero binario etichettato da numeri interi, il costo di una foglia N di T è la somma di tutti i nodi che si trovano sul ramo che va dalla radice di T a N. Scrivere una funzione `foglia_costo: int tree -> (int * int)` che, dato un albero binario di interi, restuisca l'etichetta e il costo di una delle foglie più costosa dell'albero.

   Ad esempio, se l'albero è quello rappresentato a fianco, la funzione potrà riportare, indifferentemente, la coppia `(5,15)` oppure `(3,15)`.

7. Definire una funzione `foglie_costi: int tree -> (int * int) list` che, applicata a un albero binario `T` etichettato da interi, riporti una lista di coppie, ciascuna delle quali ha la forma `(f,n)`, dove `f` è l'etichetta di una foglia in `T` e `n` il costo di tale foglia (dove il costo di una foglia è definito come nell'esercizio precedente).

Ad esempio, se l'albero è quello rappresentato nell'esercizio precedente, la funzione riporterà la lista `[(2,12);(5,15);(3,15);(3,13);(4,14)]` (o una sua permutazione).

8. Si consideri la seguente estensione del tipo di dati `expr` per la rappresentazione di espressioni aritmetiche:

```
type expr =
    Jolly
  | Int of int
  | Var of string
  | Sum of expr * expr
  | Diff of expr * expr
  | Mult of expr * expr
  | Div of expr * expr
```

Il "jolly" è un carattere speciale @, usato come 'metavariabile' (come la variabile muta), che può comparire in una espressione. Chiamiamo "espressione" una `expr` che non contenga alcun jolly, mentre le `expr` che contengono uno o più jolly vengono chiamate "modelli" (o "pattern"). Ad esempio, $x \times @$ è un pattern. Una espressione E e un modello M si confrontano positivamente se hanno la stessa struttura, cioe' se E si ottiene da M sostuendo i caratteri @ da opportune sottoespressioni (non necessariamente dalla stessa sottoespressione). Il carattere @ si comporta cioè come una variabile "muta". Ad esempio: l'espressione $a + ((b * c) - d)$ si confronta positivamente con i seguenti modelli:

$$@$$
$$a + @$$
$$@ + (@ - d)$$
$$a + (@ - @)$$

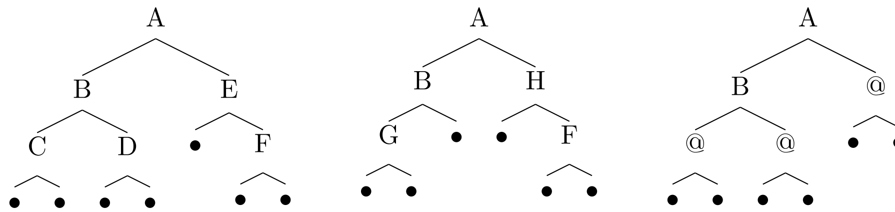
Non si confronta positivamente con:

$$a + (@ - b)$$
$$@ + (@ \times @)$$

Scrivere una funzione `pattern_matching: expr -> expr -> bool` che, data un'espressione E e un modello M, determini se E e M si confrontano positivamente oppure no.

9. Definire una funzione `max_common_subtree: string tree -> string tree -> string tree`, che, dati due alberi binari A e B, i cui nodi sono etichettati da stringhe, costruisca il massimo sottoalbero comune a A e B, partendo dalla radice: i nodi di tale sottoalbero avranno la stessa etichetta che hanno i nodi corrispondenti in A e in B, se essi sono uguali; altrimenti, se il nodo x di A è diverso dal corrispondente nodo di B (o se uno dei due nodi non c'è), il nodo corrispondente a x nel massimo sottoalbero comune di A e B sarà una foglia etichettata da `"@"`.

Ad esempio, il massimo sottoalbero comune dei due alberi rappresentati a sinistra e al centro qui sotto è quello rappresentato a destra (sono qui omesse le virgolette per delimitare le stringhe e • rappresenta l'albero vuoto).

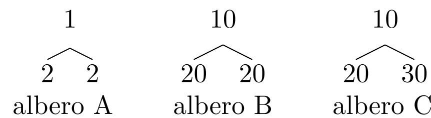


10. (Dal compito d'esame di settembre 2011)

    (a) Scrivere un predicato `stessa_struttura: 'a tree -> 'a tree -> bool` che determini se due alberi binari hanno la stessa struttura (cioè se essi sono uguali quando si ignorano le rispettive etichette; ad esempio i tre alberi rappresentati per l'esercizio successivo hanno tutti la stessa struttura).

    (b) Una funzione f è un `mapping` da un albero binario t_1 a un albero t_2 se l'applicazione di f alle etichette di t_1 trasforma t_1 in t_2. In particolare, perché possa esistere un mapping da t_1 a t_2, i due alberi devono avere la stessa struttura, ma questa non è una condizione sufficiente.

       Si considerino ad esempio gli alberi sotto rappresentati:

       Esiste un mapping dall'albero A all'albero B (la funzione rappresentata dalla lista `[(1,10);(2,20)]`, ma non dall'albero A all'albero C (la "trasformazione" `[(1,10);(2,20);(2,30)]` non è una funzione). Dall'albero C esiste un mapping all'albero A (`[(10,1);(20,2);(30,2)]`) e un mapping all'albero B (`[(10,10);(20,20);(30,20)]`).

       Scrivere un predicato `esiste_mapping: 'a tree -> 'a tree -> bool` che, applicato a due alberi binari t_1 e t_2 determini se esiste un mapping da t_1 a t_2. La funzione non deve mai sollevare eccezioni, ma deve riportare sempre un booleano.

       (Suggerimento: costruire prima la lista di coppie che dovrebbe trasformare il primo albero nel secondo, e poi verificare se essa rappresenta una funzione).

11. (Dal compito d'esame di giugno 2011) Definire una funzione

    ```
    path: ('a -> bool) -> 'a tree -> 'a list,
    ```

    che, applicata a un predicato `p: 'a -> bool` e a un albero `t: 'a tree`, riporti, se esiste, un cammino dalla radice a una *foglia* di `t` che non contenga alcun nodo che soddisfa `p`. La funzione solleverà un'eccezione se un tale cammino non esiste.

12. (Riformulazione di un esercizio del compito d'esame di febbraio 2011) Sia data la seguente dichiarazione di tipo:

    ```
    type 'a sostituzione: ('a * 'a tree) list
    ```

Una `sostituzione` è una lista associativa che associa alberi a valori di tipo `'a`. L'applicazione di una sostituzione `subst` a un albero `t` è l'albero che si ottiene da `t` sostituendo ogni *foglia* etichettata da `x` con l'albero `xt` associato a `x` in `subst`, se `subst` associa a `x` un albero, e lasciandola immutata altrimenti.

Scrivere una funzione `applica: 'a sostituzione -> 'a tree -> 'a tree` che, applicata a una sostituzione `subst` e un albero `t` riporti l'albero che si ottiene applicando la sostituzione `subst` a `t`.

13. (Dal compito d'esame di febbraio 2010). Si definisca un tipo di dati per la rappresentazione di alberi binari e scrivere un programma con una funzione `path_coprente: 'a tree -> 'a list -> 'a list` che, dato un albero A e una lista di elementi dello stesso tipo dei nodi di A, restituisca, se esiste, un ramo dell'albero dalla radice a una *foglia* che contenga tutti i nodi di L (in qualsiasi ordine) ed eventualmente anche altri nodi. Se un tale cammino non esiste, il programma solleverà un'eccezione. Si assuma che la lista L sia senza ripetizioni.

    Ad esempio, se l'albero è quello rappresentato per l'esercizio 6 e la lista è `[3;6]`, la funzione può restituire il ramo `[0;6;6;3]` oppure `0;6;4;3]`. Se la lista è `[10]`, la funzione può restituire il ramo `[0;10;2]` oppure `0;10;5]`.

14. (Dal compito d'esame di luglio 2009).

    (a) Siano date la seguenti dichiarazioni di tipo

    ```
    type col = Rosso | Giallo | Verde | Blu
    type 'a col_assoc = (col * 'a list) list
    ```

    Una lista di tipo `'a col_assoc` rappresenta un'*associazione di colori* a liste di elementi di tipo `'a`: ogni colore è associato alla lista di elementi di quel colore.

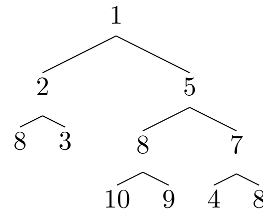
    Scrivere una funzione `colore: 'a -> 'a col_assoc -> col`, che, dato un valore `x` di tipo `'a` e una lista che rappresenta un'associazione di colori, riporti il colore di `x`, se tale colore è definito, sollevi un'eccezione altrimenti.

    Ad esempio, se `lst = [(Rosso,[1;2;4;7;10]); (Giallo,[3;8;11]); (Verde,[0;5;6;13]); (Blu,[9;12;14;15])]`, il valore di `colore 6 lst` è `Verde`, mentre `colore 100 lst` solleva un'eccezione.

    (b) Un ramo di un albero è detto a colori alterni se, nella sequenza di nodi $[x_1;\ldots;x_n]$ che lo rappresenta, due valori adiacenti hanno sempre colore diverso (secondo una data associazione di colori).

    Dichiarare un tipo di dati per rappresentare alberi binari e scrivere una funzione `path_to: 'a -> 'a col_assoc -> 'a tree -> 'a list`, che, dato un valore `x: 'a`, un'associazione di colori e un albero binario, riporti – se esiste – un ramo a colori alterni, dalla radice dell'albero a una *foglia* etichettata da `x`. Se un tale ramo non esiste, solleverà un'eccezione.

    Ad esempio, se `lst` è l'associazione di colori data al punto 1 e `t` è l'albero rappresentato qui sotto, il valore di `path_to 8 colori t` è la lista `[1;5;7;8]`.



15. Un albero binario di ricerca (ABR) è un albero binario etichettato da coppie (k, v) dove i primi elementi di ciascuna coppia (le chiavi) appartengono a un insieme ordinato e i secondi elementi sono detti valori. La proprietà fondamentale di un ABR è la seguente: per ogni nodo N dell'albero, le chiavi di tutti i nodi del sottoalbero sinistro di N sono minori della chiave di N e le chiavi di tutti i nodi del sottoalbero destro sono maggiori della chiave del nodo N.

    (a) Definire un predicato `abr_check: ('a * 'b) tree -> bool` che controlli se un albero è un ABR.

    (b) Definire una funzione `abr_search: ('a * 'b) tree -> 'a -> 'b` che, dato un ABR e una chiave k riporti, se esiste, il valore v associato a k nell'albero, un errore altrimenti.

    (c) Definire una funzione `abr_update: ('a * 'b) tree -> ('a * 'b) -> 'a tree` che, dato un ABR T e una coppia (k, v), riporti l'ABR che si ottiene da T aggiungendo l'elemento (k, v). Se T già contiene un elemento con chiave k, il suo valore verrà sostituito con v.

    (d) Definire una funzione `abr_delmin: 'a tree -> 'a * 'a tree` che, dato un ABR T riporti la coppia $(label, T')$ dove *label* (una coppia chiave-valore) è l'etichetta di T con chiave minima e T' è l'albero che si ottiene da T eliminando il nodo etichettato da *label*. Ovviamente, se l'albero T è vuoto, si solleverà un'eccezione.

        Si ricordi che in un ABR il nodo con chiave minima è quello "più a sinistra", cioè si trova scendendo sempre al sottoalbero sinistro, fino a che non si trova un nodo senza sottoalbero sinistro. E l'eliminazione della radice in un albero senza sottoalbero sinistro non è altro che il suo sottoalbero destro.

    (e) Definire una funzione `abr_delete: ('a * 'b) tree -> 'a -> ('a * 'b) tree` che implementi l'algoritmo di cancellazione di un elemento da un ABR: dato un ABR T e una chiave k, costruire l'ABR che si ottiene da T cancellando l'elemento con chiave k (se esiste, e lasciando T immutato altrimenti).

        Per chi non ricordasse l'algoritmo per la cancellazione di un elemento da un ABR, mantenendo la proprietà degli ABR:

        - sia N il nodo da eliminare e t il sottoalbero di cui è radice;
        - se t ha un solo sottoalbero t', allora t viene sostituito da t';
        - se t ha due sottoalberi, *sin* e dx, allora N viene sostituito dal nodo N' con chiave minima di dx, e dx viene sostituito dall'albero che si ottiene da dx cancellando N'.

    (f) Il risultato della visita simmetrica di un ABR è una lista ordinata secondo chiavi crescenti. L'algoritmo di ordinamento chiamato *tree sort* costruisce

un ABR i cui nodi sono etichettati dagli elementi della sequenza da ordinare, e poi riporta la sequenza risultante dalla visita simmetrica di tale albero.

Dato che le sequenze possono contenere ripetizioni, si deve però consentire che il sottoalbero sinistro di un nodo N contenga elementi uguali (e non solo minori) dell'etichetta di N. Inoltre le etichette possono essere di un tipo ordinato qualsiasi.

Scrivere una funzione `abr_insert: 'a tree -> 'a -> 'a tree` per l'inserimento di un elemento in un ABR generale di questo tipo (simile a `abr_update`), ed implementare il `tree_sort: 'a list -> 'a list`.