Programmazione Funzionale

SIMULAZIONE ESONERO 2 – SOLUZIONI

Esercizio 1. Vogliamo implementare una funzione che aggiunge dei figli a un nodo di un albero n-ario.

1. Definire il tipo `'a ntree` degli alberi n-ari.

   Soluzione: Ricordiamoci della definizione del tipo dei alberi n–ari:

   ```
   type 'a ntree = Tr of 'a * ('a ntree list);;
   ```

2. Definire una funzione `apply: ('a -> 'b) -> 'a ntree -> 'b ntree` che prende una funzione `f` e la applica a tutti i nodi di un albero `t`.

   Ad esempio `apply (function x -> x+1) Tr(1,[Tr(2,[]) ; Tr(5,[Tr(1,[])])])` restituisce `Tr(2,[Tr(3,[]) ; Tr(6,[Tr(2,[])])])`.

   Soluzione: La funzione `apply` dovra attraversare l'albero e propagarsi nei i suoi figli durante la ricorsione. Ecco un modo di definire `apply` usando `List.map` per propagare la funzione sulla lista dei sottoalberi:

   ```
   let rec apply f = function
       Tr(a,[]) -> Tr(f(a),[]) |
       Tr(a,tl) -> Tr(f(a), List.map (apply f) tl);;
   ```

   In realtà dato che `List.map (apply f) []` restituisce la lista vuota `[]`. La funzione `apply` puo essere definità con un caso solo.

   ```
   let rec apply f = function Tr(a,tl) -> Tr(f(a), List.map (apply f) tl);;
   ```

3. Definire una funzione `applysubtree: ('a ntree-> 'a ntree) -> 'a ntree -> 'a ntree` che prende una funzione `f:'a ntree -> 'a ntree` e un albero n-ario `t` e applica `f` a tutti i sottoalberi di `t`.

   Ad esempio `applysubtree (function Tr(a,tl) -> Tr(a , Tr(1,[])::tl) ) Tr(1,[Tr(2,[]) ; Tr(5,[Tr(1,[])])])` restituisce `Tr(1,[Tr(1,[]); Tr(2,[Tr(1,[])]) ; Tr(5,[Tr(1,[]) ; Tr(1,[Tr(1,[])])])])`, cioè corrisponde all'albero in entrata in cui è stato aggiunto il figlio `Tr(1,[])` a tutti i suoi sotto–alberi.

   Soluzione: Questa domanda assomiglia alla precedente, solo che la funzione `f` viene applicata a dei alberi invece che a un nodo. Ecco un modo di definire `applysubtree` usando `List.map`:

   ```
   let rec applysubtree f = function
       Tr(a,tl) -> f( Tr(a, List.map (applysubtree f) tl));;
   ```

4. Definire un funzione `addsonsat : 'a -> 'a ntree -> 'a ntree -> 'a ntree` che prende un elemento `x` un albero `t0` e un albero `t` e aggiunge a tutti i sotto–alberi di `t` che hanno come radice `x` l'albero `t0` come nuovo figlio.

   Ad esempio `addsonsat 5 t0 Tr(1,[Tr(2,[]) ; Tr(5,[Tr(1,[])])])` restituisce `Tr(1,[Tr(2,[]) ; Tr(5,[t0 ; Tr(1,[])])])`.

   Soluzione: Usando la funzione precedente sara facile definire `addsonsat`. Prima definiamo una funzione che aggiunge un figlio `t0` a un albero `t` se e solo se la sua radice e `x`:

   ```
   let addsoncnd x t0 t = match t with
       Tr(a,tl) when (x=a) -> Tr(a, t0::tl) |
       Tr(a,tl) -> Tr(a,tl);;
   ```

   Per concludere basta applicare la funzione `addsoncnd` a tutto l'albero, usando per esempio `applysubtree`.

   ```
   let addsonsat x t0 t = applysubtree (addsoncnd x t0) t;;
   ```

Esercizio 2. Vogliamo implementare diverse funzione di ricerca di un cammino in un albero binario.

1. Definire il tipo `'a btree` degli alberi binari.

   Soluzione: Ricordiamo la definizione del tipo dei alberi binari:

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

2. Definire una funzione `search: 'a -> 'a btree -> 'a list` che prende un elemento `x` e un albero binario `t` e restituisce un cammino dalla radice di `t` a `x` sotto forma di `'a list`.

   Definando `let leaf a = Tr(a,Empty, Empty)` Ad esempio `search 3 Tr(2, leaf 1 , Tr(5,leaf 3 ,Empty))` restituisce `[2;5;3]`.

   Soluzione: Per trovare un cammino usiamo il backtracking. Lo implemento come fatto nel corso ma esistono altri modo (per esempio con una mutua ricorsione):

```
    exception Reject;;

let search target t =
    let aux target t acc = match (t,acc) with
        (Tr(a,l,r),acc) when (a=target) -> acc@[a] |
        (Empty,acc) -> raise Reject |
        (Tr(a,l,r),acc) -> try aux target l (acc@[a]) with Reject -> aux target r (acc@[a]);;
    in aux target t [];;
```

3. Modificare la funzione precedente per definire una funzione `searchcnd: int -> int -> int btree -> int list` che prende un elemento `x` un intero `n` e un albero binario `t` e restituisce un cammino dalla radice di `t` a `x` sotto forma di `int list` tale che la somma degli elementi del cammino valga `n`.

   Ad esempio `searchcnd 3 12 Tr(2, leaf 1 , Tr(5, leaf 3 ,Tr(2,leaf 3, Empty)))` restituisce `[2;5;2;3]`.

   Soluzione: Per modificare la funzione precedente aggiungiamo un caso in cui la funzione solleva l'eccezione `Reject`. Definiamo una funzione che somma i elementi di una lista e poi una funzione `reject`.

```
let rec sumlist = function [] -> 0 | x::l -> x + (sumlist l);;

let reject n = function [] -> false | l -> (sumlist l > n);;
```

   Ora usiamo la funzione reject per aggiungere un caso alla funzione precedente e definire `searchcnd`:

```
let searchcnd target n t =
    let aux target n t acc = match (t,acc) with
        (Tr(a,l,r),acc) when (a=target && (sumlist (acc@[a]) = n ) )-> acc@[a] |
        (Empty,acc) -> raise Reject |
        (t,acc) when (reject n acc) -> raise Reject |
       (Tr(a,l,r),acc) -> try aux target n l (acc@[a]) with Reject -> aux target n r (acc@[a])
    in aux target n t [];;
```

4. Assumiamo di aver dichiarato `type direction = Left | Right`. Modificare la funzione precedente per definire una funzione `searchdir: int -> int -> int btree -> direction list` che prende un elemento `x` un intero `n` e un albero binario `t` e restituisce un cammino dalla radice di `t` a `x` sotto forma di `direction list` tale che la somma degli elementi del cammino valga `n`.

   Ad esempio `searchdir 3 12 Tr(2, leaf 1 , Tr(5, leaf 3 ,Tr(2,leaf 3, Empty)))` restituisce `[Right;Right;Left]`.

   Soluzione: Ora vogliamo modificare la funzione per restituisce il cammino sotto la forma di una lista di direzioni cioè di elementi che sono o il costruttore `Left` o il costruttore `Right`. Ci possiamo inspirare della funzione precendente ma non possiamo usare `sumlist` come prima quindi dobbiamo modificare il parametro `n` nelle chiamate ricorsive.

```
let searchdir target n t =
    let aux target n t acc = match (t,acc) with
        (Tr(a,l,r),acc) when (a=target && (n-a)=0 )-> acc |
        (Empty,acc) -> raise Reject |
        (t,acc) when (n < 0) -> raise Reject |
      (Tr(a,l,r),acc) -> try aux target (n-a) l (acc@[Left]) with Reject -> aux target (n-a) r (acc@[Right])
    in aux target n t [];;
```

Esercizio 3. Denotiamo $\mathbf{K}_3$ un grafo con 3 nodi tutti connessi l'un l'altro. Ad esempio se i nodi sono $\{1, 2, 3\}$ allore l'insieme degli archi è $\{(1, 2), (2, 1), (1, 3), (3, 1), (2, 3), (3, 2)\}$. Vogliamo definire una funzione che verifica se un grafo corrisponde a un grafo $\mathbf{K}_3$.

1. Definire il tipo dei grafi `'a graph`.

   Soluzione: Ricordiamo che un grafo e definito come una lista di archi cioè elementi di tipo `'a * 'a`.

   ```
   type 'a graph = ('a * 'a) list;;
   ```

2. Definire una funzione `nodes : 'a graph -> 'a list` che prende un grafo e restituisce la lista dei suoi nodi.

   Soluzione: Possiamo fare in diversi modi qua usa la versione con `list_to_set` e `List.map`.

   ```
   let rec list_to_set lst acc = match lst with
       [] -> acc |
       x::l when (List.mem x acc) -> list_to_set l acc |
       x::l -> list_to_set l (acc@[x]) ;;

   let nodes g =
   list_to_set ((List.map (function (a,b)-> a) g) @ (List.map (function (a,b)-> b) g) );;
   ```

3. Definire un funzione `nodeseq3 : 'a graph -> bool` che verificà che un grafo contiene esattamente 3 nodi.

   Soluzione: E abbastanza facile basta testare la lunghezza della lista `nodes g`. Per questo possiamo definire `length` che calcola la lunghezza di una lista.

   ```
   let rec length = function [] -> 0 | x::l -> 1 + (length l);;

   let nodeseq3 g = (length (nodes g)) = 3;;
   ```

4. Definire una funzione `noloop : 'a graph -> bool` che verificà che un grafo non contiene archi della forma (x, x).

   Soluzione: Basta leggere il grafo come una lista di archi. Ad esempio possiamo definire `noloop` nel modo seguente:

   ```
   let rec noloop = function
       [] -> false |
       (a,b)::l when (a=b) -> true |
       (a,b)::l -> noloop l;;
   ```

   Un altro modo è di usare un filtro `f : ('a * 'a) -> bool` e di usare `List.exists`.

   ```
   let noloop g = List.exists (function (a,b)-> a=b) g;;
   ```

5. Definire una funzione `remove : 'a -> 'a list -> a' list` che prende un elemento `x` una lista `lst` e rimuove la prima occorenza di `x` in `lst`.

   Soluzione: La funzione `remove` e definita leggendo semplicemente la lista in entrata, una volta tolta la prima occorenza di `x` trovata la funzione si ferma. Ecco un modo di implementare `remove`:

```
let remove x = function
    [] -> [] |
    h::t when (h=x) -> t |
    h::t -> h:: (remove x t);;
```

6. Definire un funzione `testing : ('a -> bool) list -> 'a list -> bool` che prende una lista di funzioni `fl: ('a -> bool) list` e una lista `lst` e restituisce `true` se e solo se per tutte le funzione `f` della lista `fl` esiste un elemento `x` di `lst` tale che `f(x)` restituisce `true`.

   Soluzione: Dato una lista di funzioni `f : ('a -> bool)` vogliamo verificare che una lista di tipo `'a list` contiene un elemento che restituisce `true` per `f` per tutte le funzioni della lista. Possiamo sfruttare l'effetto della funzione `List.exists` per costruire una congiunzione:

```
let rec testing fl lst = match fl with
    [] -> true |
    f::rest -> (List.exists f lst) && (testing rest lst);;
```

7. Definire un funzione `containsedges: 'a -> 'a graph -> bool` che prende un elemento `x` e un grafo `g` e restituisce `true` se il grafo `g` contiene tutti i archi `(x,u)` dove `u` e un nodo del grafo distinto da `x`.

   Soluzione: Per risolvere questa domanda possiamo sfruttare la domanda precedente. Dato un nodo `x` vogliamo verificare se il grafo contiene i archi `(x,y)` per ogni altro no `y`. Cioè `List.exists (function (a,b)-> (a,b)=(x,y)) grafo` deve restituire `true`.

   Possiamo usare `List.map` per creare tutti i filtri della forma `(function (a,b)-> (a,b)=(x,y))` e poi usare `testing`.

```
let filters x g = List.map (function y -> function (a,b)-> (a,b)=(x,y)) (remove x (nodes g));;

let containsedges x g = testing (filters x g) g;;
```

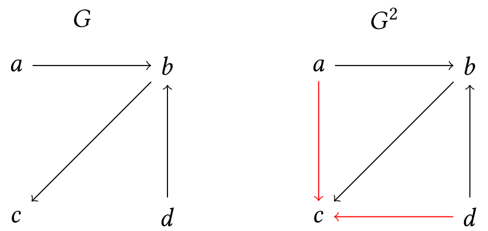
8. Usando le funzioni precedenti, definire una funzione `k3:'a graph -> bool` che prende un grafo e verifica se è un grafo $\mathbf{K}_3$.

   Soluzione: Possiamo ora concludere, per verificare che un grafo e $mathbfK_3$ dobbiamo assicurarci che non contiene nessun loop, che contiene 3 nodi e che tutti i suoi nodi sono connessi a i altri cioè che vale `containsedges x g` per tutti nodi `x`. Per definire quest'ultima funzione possiamo usare `List.forall`.

```
let connected g = List.forall (function a -> containsedges a g) (nodes g);;

let k3 g = (nodeseq3 g) && (noloop g) && (connected g);;
```

Esercizio 4. La *lunghezza* di un cammino in un grafo G è il numero di archi che attraversa. Dato un grafo G vogliamo definire il suo grafo quadrato G^2. G^2 ha gli stessi nodi di G e due nodi x e y di G^2 sono collegati con un arco se e solo se esiste un cammino in G da x a y di lunghezza minore o uguale a 2.



1. Definire una funzione `sons:'a -> 'a graph -> 'a list` che prende un nodo `x` e restituisce la lista dei suoi figli cioè i nodi `y` tali che esiste un arco `(x,y)`.

   Soluzione: Per definire la funzione `sons` basta leggere la lista dei archi e salvare i nodi che sono colleghati con un arco a `x`.

```
let rec sons x = function
    [] -> [] |
    (a,b)::l when (a=x) -> b :: (sons x l) |
    (a,b)::l -> sons x l;;
```

Un alternativà e di filtrare la lista dei archi e poi prendere i elementi a destra dei archi rimasti.

```
let sons x g = List.map (function (a,b)-> b) (List.filter (function (a,b)-> a=x) g);;
```

2. Definire una funzione `list_to_set : 'a list -> 'a list` che toglie le ripetizioni da una lista.

   Soluzione: L'abbiamo definità in un esercizio precedente, la definiamo nello stesso modo cui.

```
let rec list_to_set lst acc = match lst with
    [] -> acc |
    x::l when (List.mem x acc) -> list_to_set l acc |
    x::l -> list_to_set l (acc@[x]) ;;
```

3. Definire una funzione `sons2 : 'a -> 'a graph -> 'a list` che prende un elemento `x` e un grafo `g` e restituisce la lista dei figli dei figli di `x`.

   Soluzione: Per risolvere questo basta applicare la funzione `sons` su i figli di `x` cioè quello che restituisce `sons x g`. Se usiamo semplicemente `List.map` otteneremo una lista di liste, per cui abbiamo bisogna di concatenare queste liste. Per questo motivo definiamo `flatten` che prende una lista di liste e restituisce la loro concatenazione.

```
let rec flatten = function [] -> [] | l::rest -> l @ (flatten rest);;

let sons2 x g = list_to_set(flatten (List.map (function a -> sons a g) (sons x g)));;
```

4. Definire una funzione `newedge : 'a -> 'a graph -> 'a graph` che prende un elemento `x` e un grafo `g` e restituisce una lista di archi della forma `(x,y)` quando `y` appartiene a `sons2 x g`.

   Soluzione: I nuovi archi che dobbiamo aggiunge sono della forma `(x,y)` quando `y` appartiene a `sons2 x g`. Usando `List.map` e `sons2` possiamo facilmente creare questi archi:

```
let newedge x g = List.map (function y -> (x,y)) (sons2 x g);;
```

5. Definire una funzione `nodes: 'a graph -> 'a list` che restituisce la lista dei nodi di un grafo.

   Soluzione: Anche questa domanda e stata fatta prima, possiamo rispondere nello stesso modo.

```
let nodes g =
list_to_set ((List.map (function (a,b)-> a) g) @ (List.map (function (a,b)-> b) g) );;
```

6. Usando le funzioni precedenti definire una funzione `square : 'a graph -> 'a graph` che prende un grafo `g` e restituisce il grafo quadrato di `g`.

   Soluzione: Finalmente possiamo definire il grafo quadrato, basta aggiungere i nuovi archi al grafo, e togliere le eventuali ripetizioini con `list_to_set`:

```
let allnewedge g = flatten (List.map (function x -> newedge x g) (nodes g));;

let square g = list_to_set (g @ (allnewedge g));;
```