Esercizi proposti – 7

1. Sia dato il programma seguente:

```
type direzione = Su | Giu | Destra | Sinistra
type posizione = int * int * direzione
type azione =  Gira | Avanti of int
(* gira : direzione -> direzione *)
let gira = function
    Su -> Destra
  | Giu -> Sinistra
  | Destra -> Giu
  | Sinistra -> Su
(* avanti : posizione -> int -> posizione *)
let avanti (x,y,dir) n =
  match dir with
    Su -> (x,y+n,dir)
  | Giu -> (x,y-n,dir)
  | Destra -> (x+n,y,dir)
  | Sinistra -> (x-n,y,dir)
(* sposta : posizione -> azione -> posizione *)
let sposta (x,y,dir) act =
  match act with
    Gira -> (x,y,gira dir)
              (* le coordinate non cambiano,
                 la direzione gira di 90 gradi in senso orario *)
  | Avanti n -> avanti (x,y,dir) n
```

   Utilizzando la funzione `sposta`, definire una funzione

```
esegui:  posizione -> azione list -> posizione
```

   che, applicata a una posizione e una lista di azioni $[a_1, a_2, \ldots, a_n]$ riporti la posizione in cui si trova l'oggetto che, trovandosi inizialmente nella posizione data, esegue in sequenza (e in quest'ordine) le azioni $a_1, a_2, \ldots, a_n$.

   Ad esempio si deve avere:

```
# let p = (0,0,Su);;
val p : int * int * direzione = (0, 0, Su)

# esegui p [Avanti 3; Gira];;
- : int * int * direzione = (0, 3, Destra)

# esegui p [Avanti 3; Gira; Avanti 2];;
- : int * int * direzione = (2, 3, Destra)

# esegui p [Avanti 2; Gira; Avanti 3];;
- : int * int * direzione = (3, 2, Destra)

# esegui p [Avanti 2; Gira; Gira; Gira; Avanti 3];;
- : int * int * direzione = (-3, 2, Sinistra)
```

2. Definire il prodotto sul tipo `nat` così definito

   ```
   type nat = Zero | Succ of nat
   ```

   usando la funzione `somma` definita a lezione:

   ```
   (* somma : nat -> nat -> nat *)
   let rec somma n m =
     match n with
        Zero -> m
      | Succ k -> Succ(somma k m)
   ```

3. (Dal compito d'esame di giugno 2010) Le cassaforti della marca VeryHard hanno un sistema di apertura alquanto singolare: ciascuna di esse ha un numero N di chiavi, disposte in sequenza, ciascuna delle quali può essere in posizione aperta o chiusa. Solo quando tutte le chiavi sono aperte, la cassaforte si apre. Le chiavi devono essere girate una alla volta (passando così dalla posizione aperta a chiusa o viceversa), ma data una certa configurazione delle chiavi, non è possibile girare una chiave qualsiasi, ma soltanto **la prima chiave** (iniziando da sinistra), oppure **la chiave che segue immediatamente la prima chiave chiusa** (iniziando sempre da sinistra). Ad esempio, data la configurazione di chiavi seguente (dove C indica che la chiave è chiusa, A che è aperta): A A A C C A C è possibile girare la prima chiave, passando alla configurazione C A A C C A C, oppure la quinta (infatti la prima chiave chiusa è la quarta), passando a A A A C A A C. Se la prima chiave chiusa della configurazione è l'ultima (come ad esempio in A A C), si può soltanto girare la prima chiave.

   Date le seguenti dichiarazioni di tipo

   ```
   type chiave = Aperta | Chiusa
   type cassaforte = chiave list
   ```

   (cassaforte rappresenta una configurazione delle chiavi), definire le seguenti funzioni:

   (a) `giraPrima: cassaforte -> cassaforte`, che riporta la configurazione che si ottiene girando la prima chiave;

   (b) `giraDopoChiusa: cassaforte -> cassaforte`, che riporta la configurazione che si ottiene girando la chiave che segue la prima chiusa (e solleva un'eccezione se non è possibile eseguire l'operazione).

   (c) `successori: cassaforte -> cassaforte list`, che, applicata a una configurazione `clist`, riporta la lista con le configurazioni (una o due) che si possono ottentere da `clist` con una delle due operazioni (`giraPrima` e `giraDopoChiusa`; se `giraDopoChiusa` non è applicabile, la lista conterrà un solo elemento).

4. Si consideri il problema dei missionari e cannibali: tre missionari e tre cannibali sono sulla riva di un fiume e devono attraversarlo. Per farlo devono utilizzare una barca che non può trasportare più di due persone alla volta. Durante i trasferimenti, su nessuna delle due rive i cannibali devono essere in numero maggiore dei missionari (altrimenti ...).

   Per rappresentare le possibili situazioni dichiariamo i tipi seguenti:

```
type obj = Miss | Cann | Barca
type situazione = obj list * obj list
```

Una situazione è rappresentata da due liste di oggetti: quelli che si trovano sulla riva sinistra e quelli che si trovano sulla riva destra. Se inizialmente i tre missionari, i tre cannibali e la barca si trovano sulla riva sinistra, possiamo dichiarare:

```
let initial = ([Miss;Miss;Miss;Cann;Cann;Cann;Barca], [])
```

Le azioni consistono nello spostamento della barca con al massimo due persone da una delle due rive all'altra. Quindi possiamo dichiarare:

```
type azione =
    From_left of obj list
  | From_right of obj list
```

Stabiliamo che la lista di oggetti cui si applicano i costruttori non include la barca (che per forza si deve spostare), ma soltanto gli uomini che si spostano.

(a) Definire una funzione `safe: situazione -> bool`, che determina se una situazione e' sicura (nessun missionario viene mangiato).

(b) Definire una funzione `applica: azione -> situazione -> situazione`, che, applicata a un'azione `act` e una situazione `sit`, riporta la situazione che si ottiene applicando l'azione `act` a `sit`. La funzione deve sollevare un'eccezione se l'azione non è applicabile (ad esempio perché la barca non si trova sulla riva giusta, o se la riva "sorgente" dello spostamento non contiene il numero sufficiente di missionari o cannibali che si dovrebbero spostare), oppure se la situazione risultante non è sicura.

(c) Si definisca come segue la lista di tutte le azioni possibili:

```
let actions =
  let elems =
    [[Miss];[Cann];[Miss;Cann];[Miss;Miss];[Cann;Cann]]
  in (List.map (function x -> From_left x) elems)
      @ (List.map (function x -> From_right x) elems)
```

Definire una funzione `from_sit: situazione -> situazione list`, che, applicata a una situazione `sit` generi tutte le situazioni che si possono ottenere applicando un'azione possibile a `sit`.

5. (dall'esame di Febbraio 2011) Sia data la seguente dichiarazione di tipo:

```
type 'a pattern = Jolly | Val of 'a
```

Se `x` è un valore di tipo `'a` e `y` è un `'a pattern`, diciamo che `x` è conforme a `y` se `y = Val x` oppure `y = Jolly`. Analogamente, una lista $[x_1;\ldots;x_n]$ (con gli elementi di tipo `'a`) è conforme a una `'a pattern list` $[y_1;\ldots;y_n]$ se, per ogni $i = 1, ..., n$, x_i è conforme a y_i.

Scrivere una funzione:

```
most_general_match: 'a list -> 'a list -> 'a pattern list
```

che, date due liste della stessa lunghezza, determini la lista di pattern con il minor numero possibile di `Jolly` alla quale siano conformi entrambe le liste date. La funzione solleverà un'eccezione se le due liste hanno lunghezze diverse.