Correction TP de programmation fonctionnelle n°7

Licence d'informatique

—Caml—

▸ Exercice 1. ***Tri par insertion***

La fonction suivante effectue l'insertion à la bonne place d'un entier dans une liste triée :

```
# let rec inserer element = function
    | tete::suite when element > tete -> tete::(inserer element suite)
    | liste -> element::liste
val inserer : 'a -> 'a list -> 'a list = <fun>

# inserer 3 [1;5;9];;
- : int list = [1; 3; 5; 9]
```

1. *Programmer la fonction* `tri_par_insertion` *en utilisant la fonction précédente.*

   ```
   # let rec tri_par_insertion = function
       | [] -> []
       | x::y -> (inserer x (tri_par_insertion y));;
   val tri_par_insertion : 'a list -> 'a list = <fun>
   ```

2. *Une fonction de tri peut trier une liste d'éléments de n'importe quel type (entiers, mais aussi chaînes de caractères, par ordre lexicographique) si on lui passe en paramètre la fonction de comparaison pour ce type (*`compare:'a -> 'a -> int`*, où* `compare a b <= 0` *signifie que* `a` *est avant* `b` *dans l'ordre).*
   Modifier les fonctions `inserer` *et* `tri_par_insertion` *pour qu'elles remplissent cette condition.*

```
# let rec inserer2 element compare = function
    | tete::suite when compare element tete < 0 -> tete::(inserer2 element compare suite);;
    | liste -> element::liste
val inserer2 : 'a -> ('a -> 'a -> int) -> 'a list -> 'a list = <fun>

# let rec tri_par_insertion2 compare = function
    | [] -> []
    | x::y -> (inserer2 x compare (tri_par_insertion2 compare y));;
val tri_par_insertion2 : ('a -> 'a -> int) -> 'a list -> 'a list = <fun>
```

▸ **Exercice 2.** ***Manipulation d'une suite de cartes***

On veut manipuler un jeu de 52 cartes. Pour cela on représente les valeurs et les couleurs par des entiers selon le principe suivant :

As $\leftrightarrow 1$, *2* $\leftrightarrow 2$, $\ldots$, *10* $\leftrightarrow 10$, *Valet* $\leftrightarrow 11$, *Dame* $\leftrightarrow 12$, *Roi* $\leftrightarrow 13$

Trèfle $\leftrightarrow 1$, *Carreau* $\leftrightarrow 2$, *Coeur* $\leftrightarrow 3$, *Pique* $\leftrightarrow 4$

On représente alors une carte par un couple (valeur,couleur).
Exemple : Valet de pique $\leftrightarrow (11, 4)$

1. *Utiliser les fonctions de l'exercice 2 pour trier une liste de cartes par ordre de valeur puis de couleur.*

✁ -

```
# let comparer_valeur_puis_couleur (v1,c1) (v2,c2) =
    if v1=v2 then (c1-c2) else (v1-v2);;
val comparer_valeur_puis_couleur : int * int -> int * int -> int = <fun>

# tri_par_insertion2 [(1,2);(7,4);(12,1);(4,3)] comparer_valeur_puis_couleur;;
- : (int * int) list = [1, 2; 4, 3; 7, 4; 12, 1]
```

- ✃

2. *Même question en triant par couleur puis par valeur.*

✁ -

```
# let comparer_couleur_puis_valeur (v1,c1) (v2,c2) =
    if c1=c2 then (v1-v2) else (c1-c2);;
val comparer_couleur_puis_valeur : int * int -> int * int -> int = <fun>

# tri_par_insertion2 [(1,2);(7,4);(12,1);(4,3)] comparer_couleur_puis_valeur;;
- : (int * int) list = [12, 1; 1, 2; 4, 3; 7, 4]
```

- ✃

3. *Écrire une fonction permettant de couper une suite de cartes à la position n (on suppose que n est inférieur au nombre d'éléments de la liste). Exemple :*

```
# couper 3 [(1,2);(7,4);(12,1);(4,3)];;
- : (int * int) list = [4, 3; 1, 2; 7, 4; 12, 1]
```

✁ -

La méthode la plus propre n'utilise pas la concaténation (coûteuse) `@` à chaque fois, mais utilise une fois à la fin deux fonctions `mirror`, qui renverse une liste, et `rev_append`, qui concatène deux listes en renversant la seconde.

```
# let rec rev_append g = function
    | [] -> g
    | h::t -> rev_append (h::g) t;;
val rev_append : 'a list -> 'a list -> 'a list = <fun>
# let mirror = rev_append [];;
val mirror : 'a list -> 'a list = <fun>
# let couper =
    let rec aux g = function
      | 0 -> fun d -> rev_append (mirror g) (mirror d)
      | n -> function
               | [] -> failwith "Liste plus courte que n!"
               | h::t -> aux (h::g) (n-1) t in
    aux [];;
val couper : int -> 'a list -> 'a list = <fun>
```

4. *Écrire une fonction* `generer_couleur` *générant la liste triée des cartes de la couleur passée en paramètre.*

```
# let generer_couleur c =
    let rec aux l = function
      | 0 -> l
      | v -> aux ((v,c)::l) (v-1) in
    aux [] 13;;
val generer_couleur : 'a -> (int * 'a) list = <fun>
```

5. *En déduire une fonction* `generer_jeu` *générant un jeu de 52 cartes trié.*

```
# let generer_jeu =
          (generer_couleur 1)@(generer_couleur 2)
         @(generer_couleur 3)@(generer_couleur 4);;
val generer_jeu : (int * int) list =
  [1, 1; 2, 1; 3, 1; 4, 1; 5, 1; 6, 1; 7, 1; 8, 1; 9, 1;
   12, 1; 13, 1; 1, 2; 2, 2; 3, 2; 4, 2; 5, 2; 6, 2; 7,
   10, 2; 11, 2; 12, 2; 13, 2; 1, 3; 2, 3; 3, 3; 4, 3; 5
   8, 3; 9, 3; 10, 3; 11, 3; 12, 3; 13, 3; 1, 4; 2, 4; 3
   6, 4; 7, 4; 8, 4; 9, 4; 10, 4; 11, 4; 12, 4; 13, 4]
```

6. *Grâce aux fonctions* `Random.self_init` *(initialisation du générateur de nombres aléatoires) et* `Random.int` *(*`Random.int n` *renvoie un entier compris entre 0 et n), écrire une fonction* `battre` *prenant un jeu complet et le mélangeant aléatoirement. Essayer de trouver une méthode qui mélange bien le jeu (adaptée de votre expérience de joueur).*

✂ -

Une bonne méthode de mélange consiste à appliquer plusieurs fois (7 fois donne un bon mélange) la procédure suivante :

– séparer le jeu en 2 paquets égaux ;
– tant que les deux tas sont non vides, tirer une carte dans l'un ou l'autre aléatoirement, et ajouter cette carte au nouveau jeu.

```
# let faire_deux_tas =
    let rec aux = function
      | 1 -> fun l -> ([],l)
      | n -> function
               | [] -> failwith "Le paquet n'était pas plein!"
               | h::t -> let (a, b) = aux (n-1) t in (h::a, b) in
    aux 27;;
val faire_deux_tas : 'a list -> 'a list * 'a list = <fun>

# let rec melanger = function
    | ([], l)
    | (l, []) -> l
    | (x1::y1, b) when (Random.int 2)=0 -> x1::(melanger (y1, b))
    | (a, x2::y2) -> x2::(melanger (a, y2));;
val melanger : 'a list * 'a list -> 'a list = <fun>

# let battre_jeu l =
    let rec aux l = function
      | 0 -> l
      | n -> aux (melanger (faire_deux_tas l)) (n-1) in
    Random.self_init ();
    aux l 7;;
val battre_jeu : 'a list -> 'a list = <fun>

# battre_jeu generer_jeu;;
- : (int * int) list =
[12, 4; 12, 3; 5, 4; 3, 1; 1, 4; 7, 2; 8, 3; 2, 3;
 5, 2; 1, 1; 13, 3; 3, 4; 12, 1; 9, 3; 9, 1; 6, 3;
 6, 2; 9, 4; 7, 3; 7, 1; 12, 2; 4, 2; 5, 3; 8, 1; 1
 2, 4; 13, 2; 10, 2; 8, 4; 10, 4; 4, 1; 2, 1; 1, 3;
 5, 1; 4, 3; 11, 4; 7, 4; 6, 1; 9, 2; 11, 2; 4, 4]
```

- ✂

▸ **Exercice 3.** ***Non aux tricheurs***

1. *Écrire la fonction* `suite_de_cartes_valide` *qui prend une liste de cartes en paramètres et qui renvoie* `true` *si tous ses éléments ont bien une valeur comprise entre* 1 *et* 13 *et une couleur comprise entre* 1 *et* 4*, et* `false` *sinon.*

✂ -

```
# let rec suite_de_cartes_valide = function
    | [] -> true
    | (v,c)::y -> v>=1 && v<=13 && c>=1 && c<=4 && suite_de_cartes_valide(y);;
val suite_de_cartes_valide : (int * int) list -> bool = <fun>
```

2. *Si on tire honnêtement deux listes de cartes d'un même jeu, elles ne doivent avoir aucune carte commune. Programmez la fonction* `tricherie` *qui prend deux listes de cartes en paramètres et qui vérifie si leur intersection est bien vide.*

```
# let rec tricherie a = function
    | [] -> false
    | x::y -> let rec appartient e = function
                | [] -> false
                | h::t -> ((comparer_valeur_puis_couleur h e)=0)
                          || (appartient e t) in
              (appartient x a) || (tricherie y a);;
val tricherie : ('a * 'b) list -> ('a * 'b) list -> bool = <fun>
```