Correction TP de programmation fonctionnelle n°8

Licence d'informatique

—Caml—

Filtrage et Poker

Dans ce TP, nous allons modifier la structure du TP précédent pour manipuler un jeu de cartes : Une carte est représentée par un couple (valeur, couleur) où la couleur est définie par le type

```
type couleur = Trefle | Carreau | Coeur | Pique ;;
```

et la valeur par

```
type valeur = Petite of int | Valet | Dame | Roi | As ;;
```

Exemple : 10 de pique = `(Petite 10, Pique)`.

On considèrera l'ordre des cartes où la plus petite carte est le 2, la plus grande l'as, l'ordre des couleurs est Trèfle < Carreau < Coeur < Pique.

▸ Exercice 1. *Mise à jour des fonctions du TP précédent*

Adapter depuis le TP précédent les fonctions de comparaison de cartes, et de génération de la liste des cartes. Remarquer que toutes les autres fonctions marchent encore.

✂ -

Parler de la possibilité de forcer les types avec la syntaxe `(expr:type)`.

```
# let comparer_valeurs v1 v2 = match (v1,v2) with
     (Petite(x),Petite(y)) -> x-y
   | (Petite(x),_) -> -1
   | (_,Petite(y)) -> 1
   | (a,b) -> if (a<b) then -1
              else if (a=b) then 0
              else 1;;
val comparer_valeurs : valeur -> valeur -> int = <fun>

# let comparer_valeur_puis_couleur (v1,(c1:couleur)) (v2,(c2:couleur)) =
           let a = comparer_valeurs v1 v2
           in if a<>0 then a
           else if c1<c2 then -1
           else if c1=c2 then 0
```

```
          else 1 ;;
val comparer_valeur_puis_couleur :
  valeur * couleur -> valeur * couleur -> int = <fun>

# let comparer_couleur_puis_valeur (v1,(c1:couleur)) (v2,(c2:couleur)) =
          if c1=c2 then (comparer_valeurs v1 v2)
          else if c1<c2 then -1
          else if c1=c2 then 0
          else 1 ;;
val comparer_couleur_puis_valeur :
  valeur * couleur -> valeur * couleur -> int = <fun>

# let generer_couleur c =
         let rec gen v c =
            if v=10 then [(Petite(10),c)]
            else (Petite(v),c)::(gen (v+1) c)
         in
            (gen 2 c)@[(Valet,c);(Dame,c);(Roi,c);(As,c)];;
val generer_couleur : 'a -> (valeur * 'a) list = <fun>

# let generer_jeu = generer_couleur(Trefle)@generer_couleur(Coeur)
                  @generer_couleur(Pique)@generer_couleur(Carreau);;
```

--- ✂

▸ Exercice 2. *Génération de mains*

On appelle **main** *toute liste de 5 cartes. On veut pouvoir obtenir des mains à partir d'un jeu complet mélangé (voir TP précédent). Pour cela, il faut pouvoir renvoyer les cartes retenues ainsi que le jeu privé de ces cartes. On utilisera pour cela des fonctions renvoyant des couples ou des listes.*

1. *Programmer la fonction* `tirer_une_carte` *qui prend en paramètre une liste de cartes et qui renvoie le couple formé par la première carte et la liste privée de cette carte :*

```
# tirer_une_carte [ (Petite 7,Pique) ; (Valet,Trefle) ; (Petite 8,Coeur);
  (Valet,Coeur) ; (Petite 7,Trefle) ];;
- : (valeur * couleur) * (valeur * couleur) list = (Petite 7, Pique),
                        [Valet, Trefle; Petite 8, Coeur; Valet, Coeur;
                        Petite 7, Trefle]
```

✂ ---

```
# let tirer_une_carte l = match l with x::y -> (x,y);;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val tirer_une_carte : 'a list -> 'a * 'a list = <fun>
```

--- ✂

2. *En utilisant* `tirer_une_carte`*, écrire la fonction* `tirer_n_cartes` *qui prend un nombre* n *de cartes à tirer et une liste de cartes, et qui renvoie le couple formé par les* n *cartes et la liste privée de ces cartes :*

```
# tirer_n_cartes 2 [ (Petite 7,Pique) ; (Valet,Trefle) ; (Petite 8,Coeur);
  (Valet,Coeur) ; (Petite 7,Trefle) ];;
- : (valeur * couleur) list * (valeur * couleur) list = [Petite 7, Pique;
                           Valet, Trefle], [Petite 8, Coeur; Valet, Coeur;
                           Petite 7, Trefle]
```

✂ -

```
# let rec tirer_n_cartes n l =
    if n=0 then ([],l)
    else match (tirer_une_carte l) with
        (a,y) -> match (tirer_n_cartes (n-1) y) with
                 (y,z) -> (a::y,z);;
val tirer_n_cartes : int -> 'a list -> 'a list * 'a list = <fun>
```

- ✂

3. *En déduire une fonction* `tirer_n_mains` *qui prend un nombre* n *et une liste de cartes, et qui renvoie la liste constituée de la liste privée des* $5 * n$ *cartes tirées et des* n *mains :*

```
# tirer_n_mains 1 [ (Petite 9,Carreau) ; (Petite 7,Carreau) ;
  (Petite 8,Carreau) ; (Petite 6,Carreau) ; (Petite 10,Carreau);
  (Petite 9,Carreau) ; (Roi,Coeur) ];;

- : (valeur * couleur) list list * (valeur * couleur) list =
([[(Petite 9, Carreau); (Petite 7, Carreau); (Petite 8, Carreau);
   (Petite 6, Carreau); (Petite 10, Carreau)]],
 [(Petite 9, Carreau); (Roi, Coeur)])
```

✂ -

```
# let rec tirer_n_mains n l =
    if n=0 then ([],l)
    else match (tirer_n_cartes 5 l) with
        (a,y) -> match (tirer_n_mains (n-1) y) with
                 (y,z) -> (a::y,z);;
val tirer_n_mains : int -> 'a list -> 'a list list * 'a list = <fun>
```

- ✂

▸ **Exercice 3.** ***Valeur d'une main***

On souhaite maintenant pouvoir évaluer la valeur d'une main. Les valeurs possibles que l'on utilisera sont, dans l'ordre, la quinte flush (5 cartes qui se suivent de même couleur), le carré (4 cartes identiques), le full (un brelan plus une paire), la couleur (5 cartes de même couleur), la quinte (5 cartes qui se suivent), le brelan (3 cartes identiques), la double-paire, la paire, et la main qui ne vaut rien.

Pour cela, on se dote d'un type `jeu` *décrivant ces valeurs :*

```
type jeu=
  Quinte_flush of valeur (*carte la plus élevée*)
| Carre of valeur        (*carte du carré*)
| Full of valeur*valeur  (*carte du brelan et de la paire*)
| Couleur of valeur      (*carte la plus élevée*)
| Quinte of valeur       (*carte la plus élevée*)
| Brelan of valeur       (*carte du brelan*)
| Double_paire of valeur*valeur
                         (*carte de la plus grande paire et de la plus petite*)
| Paire of valeur        (*carte de la paire*)
| Rien ;;
```

En vous aidant de fonctions de tri par valeur et par couleur, écrire les fonctions suivantes (chaque fonction prend une main en paramètre) :

1. `quinte_flush` *: renvoie* `Quinte_flush(n)` *où* `n` *est la valeur de la plus haute carte si on a une quinte flush et* `Rien` *sinon.*

```
# quinte_flush [ (Petite 7,Carreau) ; (Petite 8,Carreau) ;
  (Petite 6,Carreau) ; (Petite 10,Carreau) ; (Petite 9,Carreau) ];;
- : jeu = Quinte_flush Petite 10
```

✂ -

```
# let quinte_flush l =
    let m = (tri_couleur l)
    in
    match m with
    [(_,c1);_;_;_;(_,c2)] -> if (c1<>c2) then Rien else
    match m with
      [(Petite(2),_);_;_;_;(Petite(6),_)] -> Quinte_flush(Petite(6))
    | [(Petite(3),_);_;_;_;(Petite(7),_)] -> Quinte_flush(Petite(7))
    | [(Petite(4),_);_;_;_;(Petite(8),_)] -> Quinte_flush(Petite(8))
    | [(Petite(5),_);_;_;_;(Petite(9),_)] -> Quinte_flush(Petite(9))
    | [(Petite(6),_);_;_;_;(Petite(10),_)] -> Quinte_flush(Petite(10))
    | [(Petite(7),_);_;_;_;(Valet,_)] -> Quinte_flush(Valet)
    | [(Petite(8),_);_;_;_;(Dame,_)] -> Quinte_flush(Dame)
    | [(Petite(9),_);_;_;_;(Roi,_)] -> Quinte_flush(Roi)
    | [(Petite(10),_);_;_;_;(As,_)] -> Quinte_flush(As)
    | _ -> Rien;;
```

- ✂

2. `carre` *: renvoie* `Carre(n)` *si on a un carré et* `Rien` *sinon.*

```
# carre [ (Petite 7,Pique) ; (Valet,Trefle) ; (Petite 7,Coeur) ;
  (Petite 7,Coeur) ; (Petite 7,Trefle) ];;
- : jeu = Carre Petite 7
```

```
# let carre l =
    let m = (tri_valeur l)
    in
    match m with
    [(a,_);(b,_);_;(c,_);(d,_)] ->
      if (a=c || b=d) then Carre(a) else Rien;;
```

---

3. `full` *: renvoie* `Full(n,m)` *si on a un full et* `Rien` *sinon.*

```
# full [ (Petite 7,Pique) ; (Valet,Trefle) ; (Petite 7,Coeur) ;
  (Valet,Carreau) ; (Petite 7,Trefle) ];;
- : jeu = Full (Petite 7, Valet)
```

```
# let full l =
    let m = (tri_valeur l)
    in
    match m with
    [(a,_);(b,_);(c,_);(d,_);(e,_)] ->
      if (a=b && b=c && d=e) then Full(a,e)
      else if (a=b && c=d && d=e) then Full(e,a)
      else Rien;;
```

---

4. `couleur` *: renvoie* `Couleur(n)` *si on a une couleur et* `Rien` *sinon.*

```
# couleur [ (Petite 7,Pique) ; (Valet,Pique) ; (Petite 7,Pique) ;
  (As,Pique) ; (Petite 3,Pique) ];;
- : jeu = Couleur (As)
```

```
# let couleur l =
    match (tri_couleur l) with
    [(_,a);_;_;_;(_,e)] -> if a=e then
      match (tri_valeur l) with
    [(b,_);_;_;_;_] -> Couleur(a,b)
      else Rien;;
```

---

5. `quinte` *: renvoie* `Quinte(n)` *si on a une quinte et* `Rien` *sinon.*

```
# quinte [ (Petite 7,Pique) ; (Petite 8,Carreau) ; (Petite 6,Coeur) ;
  (Petite 10,Trefle) ; (Petite 9,Pique) ];;
- : jeu = Quinte Petite 10
```

✂ -

```
# let quinte l =
    let m = (tri_valeur l)
    in
    match m with
      [(Petite(2),_);_;_;_;(Petite(6),_)] -> Quinte(Petite(6))
    | [(Petite(3),_);_;_;_;(Petite(7),_)] -> Quinte(Petite(7))
    | [(Petite(4),_);_;_;_;(Petite(8),_)] -> Quinte(Petite(8))
    | [(Petite(5),_);_;_;_;(Petite(9),_)] -> Quinte(Petite(9))
    | [(Petite(6),_);_;_;_;(Petite(10),_)] -> Quinte(Petite(10))
    | [(Petite(7),_);_;_;_;(Valet,_)] -> Quinte(Valet)
    | [(Petite(8),_);_;_;_;(Dame,_)] -> Quinte(Dame)
    | [(Petite(9),_);_;_;_;(Roi,_)] -> Quinte(Roi)
    | [(Petite(10),_);_;_;_;(As,_)] -> Quinte(As)
    | _ -> Rien;;
```

- ✂

6. `brelan` *: renvoie* `Brelan(n)` *si on a un brelan et* `Rien` *sinon.*

```
# brelan [ (Dame,Pique) ; (Dame,Trefle) ; (Petite 4,Coeur) ;
  (Dame,Coeur) ; (Petite 8,Carreau) ];;
- : jeu = Brelan Dame
```

✂ -

```
# let brelan l =
    let m = (tri_valeur l)
    in
    match m with
    [(a,_);(b,_);(c,_);(d,_);(e,_)] ->
      if ((a=b && b=c) || (b=c && c=d) || (c=d && d=e)) then Brelan(c)
      else Rien;;
```

- ✂

7. `double_paire` *: renvoie* `Double_paire(n,m)` *si on a une double-paire et* `Rien` *sinon.*

```
# double_paire [ (Petite 8,Trefle) ; (Valet,Trefle) ; (Petite 5,Pique) ;
  (Valet,Coeur) ; (Petite 8,Carreau) ];;
- : jeu = Double_paire (Valet, Petite 8)
```

```
# let double_paire l =
    let m = (tri_valeur l)
    in
    match m with
    [(a,_);(b,_);(c,_);(d,_);(e,_)] ->
      if ((a=b && (c=d || d=e)) || (b=c && d=e)) then Double_paire(b,d)
      else Rien;;
```

8. `paire` *: renvoie* `Paire(n)` *si on a une paire et* `Rien` *sinon.*

```
# paire [ (Petite 7,Pique) ; (Petite 8,Carreau) ; (Petite 9,Coeur) ;
  (Petite 10,Trefle) ; (Petite 9,Pique) ];;
- : jeu = Paire Petite 9
```

```
# let paire l =
    let m = (tri_valeur l)
    in
    match m with
    [(a,_);(b,_);(c,_);(d,_);(e,_)] ->
      if ((a=b) || (b=c)) then Paire(b)
      else if ((c=d) || (d=e)) then Paire(d)
      else Rien;;
```

9. *Naturellement, une même main peut satisfaire plusieurs valeurs (un brelan contient une paire, par exemple). Programmer une fonction* `meilleur_jeu` *qui renvoie la plus forte valeur pour une main donnée.*

```
# meilleur_jeu [ (Petite 8,Pique) ; (Petite 8,Coeur) ; (Petite 9,Coeur) ;
  (Petite 9,Trefle) ; (Petite 9,Pique) ];;
- : jeu = Full (Petite 9, Petite 8)
# meilleur_jeu [ (As,Pique) ; (Dame,Coeur) ; (Petite 2,Carreau) ;
  (Petite 7,Pique) ; (Petite 10,Pique) ];;
- : jeu = Rien
```

```
# let meilleur_jeu l =
    match quinte_flush(l) with
    Quinte_flush(x) -> Quinte_flush(x)
    | _ ->
    match carre(l) with
    Carre(x) -> Carre(x)
    | _ ->
```

```
match quinte(l) with
Quinte(x) -> Quinte(x)
| _ ->
match full(l) with
Full(x,y) -> Full(x,y)
| _ ->
match brelan(l) with
Brelan(x) -> Brelan(x)
| _ ->
match double_paire(l) with
Double_paire(x,y) -> Double_paire(x,y)
| _ ->
match paire(l) with
Paire(x) -> Paire(x)
| _ -> Rien;;
```