NFP119 : corrigé de la feuille d'exercices 3

María-Virginia Aponte

20 mars 2010

Exercice 1

Tapez ces phrases en TP et expliquez les réponses données par Ocaml.

```
# [1;2;3];;
- : int list = [1; 2; 3]

# [];;
- : 'a list = []

# [1;3;1,2];;
This expression has type int * int but is here used with type int

# [1;true;5;false];;
This expression has type bool but is here used with type int

# [1,true; 5,false];;
- : (int * bool) list = [(1, true); (5, false)]

# [1;3] @[4;8];;
- : int list = [1; 3; 4; 8]

# (@);;
- : 'a list -> 'a list -> 'a list = <fun>

# [1;3] @ ['a'];;
This expression has type char but is here used with type int

# [[1];[2;3]];;
- : int list list = [[1]; [2; 3]]

# [[1] @ [2;3]];;
- : int list list = [[1; 2; 3]]

# List.length [[1];[2;3]];;
- : int = 2

# List.length [[1];[2;3]];;
- : int = 2

# [1] @ 2;;
This expression has type int but is here used with type int list

# [1] @ [2];;
```

```
- : int list = [1; 2]

#   1::[2];;
- : int list = [1; 2]

# [1]::[2];;
This expression has type int but is here used with type int list

# [1]::[[2]; [3;4]];;
- : int list list = [[1]; [2]; [3; 4]]
```

Exercice 2

Premières fonctions sur les listes

Que fait la fonction suivante ? Donnez un déroulement récursif pour exppliquer son fonctionnement.

```
# let rec indice x l =
match l
with [] -> failwith "indice"
  |  a::reste -> if x=a then 1 else 1+(indice x reste)
val indice : 'a -> 'a list -> int = <fun>

# indice 4 [5;7;1];;
Exception: Failure "indice".

# indice 4 [5;7;4;9];;
- : int = 3
```

Solution : Cette fonction donne la position de la première occurrence d'un élément dans une liste, s'il est présent et échoue sinon.

En utilisant le filtrage, écrire des fontions pour :

1. Tester si une liste commence par 0 ou par 1.
2. Tester si une liste est vide.
3. Tester si une liste est composée d'au mois un et d'au plus deux élèments.

```
# let unZero l =
match l
with 1::_ -> true
  |  0::_ -> true
  |  _ -> false
val unZero : int list -> bool = <fun>

#  let vide l =
match l
with [] -> true
  |  _ -> false      ;;
val vide : 'a list -> bool = <fun>


# let unDeux l =
match l
with  [_;_] -> true
```

```
  |   [_] -> true
  | _ -> false      ;;
val unDeux : 'a list -> bool = <fun>

# unDeux [1;2;3];;
- : bool = false
```

Exercice 3

Écrivez les fonctions suivantes. Il s'agit de fonctions présentes dans le module `List` dont vous pourrez tester le fonctionnement et comparer avec celui des votres.

1. `nieme` (`List.nth`). Renvoie le n-ième élément d'une liste.
2. `inverse` (`List.rev`). Renvoie la liste argument en ordre inversé.
3. `applatit` (`List.flatten`). Concatène les éléments d'une liste de listes.
4. `existe` `List.exist`) Teste si au moins un des éléments de la liste satisfait une condition passée en argument.

```
let rec nieme l i =
match l
with []     -> failwith "nieme"
 |   a::r  -> if i<0 then failwith "nieme"
              else if i=0 then a else nieme r (i-1);;
val nieme : 'a list -> int -> 'a = <fun>

# nieme  [1] (-2);;
Exception: Failure "nieme".

# nieme [1] 2;;
Exception: Failure "nieme".

# nieme [3;5;7] 2;;
- : int = 5

let rec inverse l =
match l
with []   -> []
  |  a::r -> (inverse r)@[a];;
val inverse : 'a list -> 'a list = <fun>

# inverse [1;2;3];;
- : int list = [3; 2; 1]

# let rec applatit l =
  match l
  with [] -> []
  |  a::r -> a @ (applatit r);;
val applatit : 'a list list -> 'a list = <fun>

# applatit [[1;2]; [3;4]; [5]];;
- : int list = [1; 2; 3; 4; 5]

# let rec existe cond l =
match l
with [] -> false
```

```
 |  a::r -> cond a || existe cond r;;
val existe : ('a -> bool) -> 'a list -> bool = <fun>

# existe (fun x -> x mod 2 = 0) [1;7;6;9];;
- : bool = true

# existe (fun x -> x mod 2 = 0) [1;7;9];;
- : bool = false
```

Exercice 4

Considérez la fonction `remplace` vue en cours. Modifiez cette fonction en une fonction `remplace_tous` de sorte qu'elle remplace toutes les occurrences de l'élément cherché.

```
# let rec remplace e x l =
match l
with [] -> []
  |  a::reste -> if e=a then x::reste else a::(remplace e x reste)
val remplace : 'a -> 'a -> 'a list -> 'a list = <fun>


# let rec remplace_tous e x l =
match l
with [] -> []
  |  a::reste -> if e=a then x::(remplace_tous e x reste)
                 else a::(remplace_tous e x reste)        ;;
val remplace_tous : 'a -> 'a -> 'a list -> 'a list = <fun>

# remplace_tous 3 5 [3;4;3;7;9];;
- : int list = [5; 4; 5; 7; 9]

# remplace_tous 3 5 [1;2;0;9];;
- : int list = [1; 2; 0; 9]
```

Exercice 5

Fonctions récursives sur les listes

1. Ecrire une fonction `range_list` qui prend deux entiers (i, n) et fabrique une liste formée de l'intervalle `[i ; (i+1) ; (i+2) ; .. (i+n-1)]`. Exemple : l'appel `range_list (3,5)` renvoie la liste `[3 ; 4 ; 5 ; 6 ; 7]`.
2. Ecrire la fonction calcule l'intersection de deux listes. On suppose que celles-ci n'ont pas d'éléments répétés. Utilisez si nécessaire des fonctions du module `List`.

```
# let rec range_list i n =
  if n<=0 then []
  else i::(range_list (i+1) (n-1));;
    val range_list : int -> int -> int list = <fun>

# range_list 1 0;;
- : int list = []

# range_list 2 4;;
```

```
- : int list = [2; 3; 4; 5]

# let rec intersect (l1,l2) =
  match l1
  with []   -> []
    |  x::r -> if List.mem x l2 then x::(intersect (r,l2))
               else intersect (r,l2);;
val intersect : 'a list * 'a list -> 'a list = <fun>

# intersect ([1;2;3], [7;2;5]);;
- : int list = [2]
```

Exercice 6

Dans cet exercice, on s'intéresse à une notion d'ensemble où il est possible qu'un même élément apparaisse plusieurs fois. Un tel ensemble est appelé *multi-ensemble*, et à chaque élément e d'un multi-ensemble, on associe le nombre de fois où il apparaît, appelé *multiplicité* de e. Ainsi, on peut voir un élément d'un multi-ensemble comme un couple constitué de la valeur proprement dite et de sa multiplicité. Par exemple, le multi-ensemble où 1, 3 et 2 apparaissent respectivement 20, 4 et 1 fois, sera représenté de la manière suivante : $\{(1, 20), (3, 4), (2, 1)\}$.

On souhaite modéliser les multi-ensembles d'entiers à l'aide d'une liste en Ocaml. Les éléments de cette liste auront le type suivant :

```
type multiElement = {e: int; mult: int}
```

où le champ `e` correspond à l'élémént et `mult` à sa multiciplité dans l'ensemble.

Vous devez implanter les opérations suivantes sur les multi-ensembles :

1. Donnez une variable mE qui contient le multi-ensemble de l'exemple donné plus haut.

   ```
   # let mE = [{e=1; mult=20}; {e=3; mult=4}; {e=2; mult=1}];;
   val mE : multiElement list =
     [{e = 1; mult = 20}; {e = 3; mult = 4}; {e = 2; mult = 1}]
   ```

2. `ajout : int * multiElement list -> multiElement list` prend un entier n et un multi-ensemble e, et renvoie le nouveau multi-ensemble où n est ajouté. Si n appartient déjà au multi-ensemble, alors cette opération incrémente sa multiplicite ; s'il n'appartient pas au multi-ensemble, alors il est rajouté avec une multiplicité de 1.

   ```
   # let rec ajout (i,me) = match me
     with []              -> [ {e=i; mult=1}]
       | {e=x; mult=n}::r ->
               if x=i then  {e=x; mult=n+1}::r
               else  {e=x; mult=n}::(ajout (i,r));;
   val ajout : int * multiElement list -> multiElement list = <fun>

   # ajout (2, mE);;
   - : multiElement list =
   [{e = 1; mult = 20}; {e = 3; mult = 4}; {e = 2; mult = 2}]
   ```

3. `enleve` prend un entier n et un multi-ensemble e, et renvoie le nouveau multi-ensemble où n est soit enlevé si sa multiciplité est de 1, soit laissé avec une multiplicité decrémentée.

   ```
   let rec enleve i me = match me
    with  []              -> []
   ```

```
    | {e=x; mult=n}::r ->
            if x=i then
                  if n<=1 then r
                  else {e=x; mult=n-1}::r
            else  {e=x; mult=n}::(enleve i r);;
val enleve : int -> multiElement list -> multiElement list = <fun>

# enleve 3 mE;;
- : multiElement list =
[{e = 1; mult = 20}; {e = 3; mult = 3}; {e = 2; mult = 1}]

# enleve 2 mE;;
- : multiElement list = [{e = 1; mult = 20}; {e = 3; mult = 4}]
```

4. `appartient` teste si un entier n appartient à un multi-ensemble.

```
#let rec appartient i me =  match me
 with  []              -> false
    | {e=x; mult=n}::r -> x=i && n>0 || appartient i r;;
val appartient : int -> multiElement list -> bool = <fun>

# appartient 3 mE;;
- : bool = true

# appartient 6 mE;;
- : bool = false
```

5. `multi` prend un entier n et un multi-ensemble e et renvoie la multiciplité de n dans le multi-ensemble. Renvoie 0 si n n'appartient pas à e.

```
# let rec multi i me =  match me
 with  []              -> 0
    | {e=x; mult=n}::r -> if x=i then n
                          else multi i r;;
val multi : int -> multiElement list -> int = <fun>

# multi 1 mE;;
- : int = 20
# multi 6 mE;;
- : int = 0
```

6. `union` renvoie l'union de deux multi-ensembles. Les multiplicités des éléments communs sont ajoutées dans le multi-ensemble résultat de l'union.

```
# let rec supprimme i m = match m
  with  []             -> []
    | {e=x; mult=n}::r ->
            if x=i then r
            else  {e=x; mult=n}::(supprimme i r);;
val supprimme : int -> multiElement list -> multiElement list = <fun>

# supprimme 2 mE;;
- : multiElement list = [{e = 1; mult = 20}; {e = 3; mult = 4}]

# let rec union m1 m2 = match m1
 with  []              -> m2
    | {e=x; mult=n}::r ->
         let k = multi x m2  in
```

```
         if k>0 then  {e=x; mult=n+k}::(union r (supprimme x m2))
         else  {e=x; mult=n}::(union r m2);;
val union : multiElement list -> multiElement list -> multiElement list =  <fun>


let m2 = [{e=5;mult=10}; {e=1;mult=2}; {e=2;mult=2}];;

# union mE m2;;
- : multiElement list =
[{e = 1; mult = 22}; {e = 3; mult = 4}; {e = 2; mult = 3};
 {e = 5; mult = 10}]
```

7. On aimerait rendre les fonctions précédentes génériques pour n'importe quel type de donnée à ajouter dans le multi-ensemble, porvu que cette donnéé soit munie d'une multiplicité. Peut-on définir un type de multi-ensembles générique ? Donnez la définition de ce type.

```
type 'a multiElement = {e: 'a; mult: int}
```