Programmation fonctionnelle : feuille d'exercices 3

María-Virginia Aponte

19 mars 2011

Exercice 1

Filtrage et premières listes

Question 1

Complétez le tableau suivant :

Motif	Valeur comparée	Réussite	Liaisons
3	1		
_	1		
x	(2,true)		
(x,y,z)	(2,true)		
(x,2)	2		
`(0,x,y)`	`(0,1,6)`		
	`(0,true,[])`	oui	a=0, b=[]
{`cle=k ; contenu=c`}	{`cle=5 ; contenu="abc"`}	oui	
{`contenu=c`}	{`cle=5 ; contenu="abc"`}		
	1	échec	
`(1,x)`	`(1,2,3)`		
`(_,x,_)`	`(0,1,6)`		
`x : :y : :z`	`[1]`		
`x : :y`	`[]`		
`x : :y : :z`	`[1 ;2]`		
	`[1 ;2 ;3 ;4]`	oui	a=1, b=2, c= [3 ;4]
	`[1 ;2 ;3 ;4]`	oui	a=1, c= [4]

Lorsqu'il s'agit de compléter la colonne "Motifs" plusieurs réponses sont en général possibles.

Question 2

Considérez la définition de type suivante :

```
type user = {uid: string; group: int; lastDate: int*int*int}
```

Ecrire en utilisant le filtrage les fonctions suivantes :

1. la fonction qui donne la dernière date de connexion des utilisateurs dont le uid est "root" et dont le groupe est 0 ou 1 ;
2. la fonction qui donne l'uid des utilisateurs dont la dernière connexion date de mars 2011 ;

Question 3

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

```
[1;2;3];;

[];;

[1;3;1,2];;

[1;true;5;false];;

[1,true; 5,false];;

[1;3] @[4;8];;

(@);;

[1;3] @ ['a'];;

[[1];[2;3]];;

[[1] @ [2;3]];;

List.length [[1];[2;3]];;

[1] @ 2;;

[1] @ [2];;

1::[2];;

[1]::[2];;

[1]::[[2]; [3;4]];;

List.rev [1;2;3];;
```

Exercice 2

Premières fonctions sur les listes

Question 1

Que fait la fonction suivante ? Donnez un déroulement récursif pour expliquer 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];;
Exception: Failure "indice".
```

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

Question 2

La fonction `appartient` teste si un élément appartient à une liste :

```
# let rec appartient e l = match l
   with [] -> false
    | a::reste -> e=a || appartient e reste;;
val appartient : 'a -> 'a list -> bool = <fun>

# appartient 5 [6;5;7];;
- : bool = true

# appartient 5 [2;6];;
- : bool = false

# appartient 3 [];;
- : bool = false
```

On se donne une nouvelle version de cette fonction :

```
# let rec appartientBis e l = match l
   with [] -> failwith "appartientBis"
    | a::reste -> e=a || appartientBis e reste;;
val appartientBis : 'a -> 'a list -> bool = <fun>
```

1. Testez ces deux fonctions avec différents appels. Dans quels cas leur comportement est identique ? Dans quels cas il est diffèrent ?
2. D'après vous, ces deux fonctions sont elles acceptables en tant que solution au problème de teste si un élément appartient à une liste ?

Question 3

Voici deux versions de la fonction qui calcule la longueur d'une liste.

```
# let rec longueur l =
   match l
   with [] -> 0
     | _::reste -> 1 + longueur reste;;
val longueur : 'a list -> int = <fun>

# longueur [];;
- : int = 0

# longueur ["a"; "salut"];;
- : int = 2

# let rec longueurBis l =
   match l
   with [] -> failwith"longueur"
     | _::reste -> 1 + longueurBis reste;;
val longueurBis : 'a list -> int = <fun>
```

Testez les et comparez leur comportement. Quelle est votre conclusion ?

1. Quelle est la différence d'utilisation de `failwith` dans la fonction `indice` et ici ?
2. Quelle est votre conclusion sur la bonne utilisation de `failwith` dans les fonctions récursives ?

Question 4

En utilisant le filtrage, écrire des font ions 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 composé d'au moins un et d'au plus deux élèments.

Exercice 3

Fonctions récursives sur les listes
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 vôtres. Testez toutes vos fonctions en TP.

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.

Exercice 4

Considérez la fonction `remplace` vue en cours.

```
# 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>
```

Question 1

Considérez cette nouvelle version :

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

Testez ces deux fonctions sur plusieurs exemples. Pouvez vous expliquer leur comportement ?

Question 2

Modifiez la fonction remplace vue en cours en une fonction `remplace_tous` de sorte qu'elle remplace toutes les occurrences de l'élément cherché.
Testez le fonctionnement de ces deux versions sur des exemples.

Exercice 5

Fonctions récursives sur les listes

1. Écrire 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. Écrire 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`.

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.
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.
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.
4. `appartient` teste si un entier n appartient à un multi-ensemble.
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.
6. `union` renvoie l'union de deux multi-ensembles. Les multiplicités des éléments communs sont ajoutées dans le multi-ensemble résultat de l'union.
7. On aimerait rendre les fonctions précédentes génériques pour n'importe quel type de donnée à ajouter dans le multi-ensemble, pourvu que cette donnée soit munie d'une multiplicité. Peut-on définir un type de multi-ensembles générique ? Donnez la définition de ce type.