UNIVERSITÉ D'ORLÉANS **DEUG MIAS**
IF 2.01 - Informatique **Année 2001-2002**

Feuille d'exercices n°3

Exercice 4 Travail sur les listes

1. la fonction `somme` : elle prend en argument une liste d'entiers et renvoie la somme de tous les éléments de la liste. Pour ce faire, on procède par récursivité : si la liste est vide, la somme vaut zéro. Sinon, on extrait le premier élément de la liste, et on l'ajoute à la `somme` du reste de la liste.

```
# let rec somme l = match l with
        []      -> 0 |
        e::m    -> e + somme m;;
val somme : int list -> int = <fun>
```

   Dans ce cas, CAML s'occupe tout seul de la correspondance entre `l` et `e : :m`. Ainsi, `e` est le premier élément de la liste, et `m` est la liste formée des autres éléments de `l`.

2. la fonction `nb_occur` : le principe est le même : dans la liste vide, il y a nécessairement zéro occurence de l'élément. Sinon, on teste sur le premier élément, et on appelle récursivement la fonction sur le reste de la liste.

```
# let rec nb_occur a l = match l with
        []      -> 0 |
        e::m    -> (if e=a then 1 else 0) + nb_occur a m;;
val nb_occur : 'a -> 'a list -> int = <fun>
```

3. la fonction `appartient` : autant utiliser ce qui existe déjà, même si ce n'est pas le plus efficace :

```
# let appartient a l = (nb_occur a l > 0);;
val appartient : 'a -> 'a list -> bool = <fun>
```

4. la fonction `ensemble` : on va construire une liste contenant l'ensemble des éléments de `l` : on prend l'élément de tête de la liste, on regarde s'il `appartient` déjà à la liste que l'on construit. Si oui, on le laisse tomber, sinon, on l'ajoute à la liste qu'on est en train de construire. Lorsque la liste de départ est vide, alors la liste que l'on construit est complète.

```
# let rec ens l nouv = match l with
        []      -> nouv |
        e::m    -> if (appartient e nouv) then (ens m nouv)
                                          else (ens m (e::nouv));;
val ens : 'a list -> 'a list -> 'a list = <fun>
```

   Bien entendu, ce n'est pas exactement ce que l'on nous demandait...

```
# let ensemble l = ens l [];;
val ensemble : 'a list -> 'a list = <fun>
```

   On remarquera que cette fonction inverse l'ordre des éléments de la liste. Cependant, ceci n'est pas très gênant dans le cas présent. On écrira, à la fon de cet exercice, une fonction `renverser` qui inverse les éléments d'une liste, et qui pourra donc rétablir l'ordre de cette liste si nécessaire.

5. la fonction `regroupe` : l'idée est simplement de prendre le résultat de `ensemble`, et de remplacer chaque élément de cette liste par un couple formé de cet élément et de son nombre d'occurences dans la liste. Nous aurons pour cela besoin de la fonction `map`, qui se trouve dans la bibliothèque `List` mais que l'on peut aisément redéfinir :

```
# let rec map f l = match l with
        []      -> [] |
        e::m    -> (f e)::(map f m);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let regroupe l =
        let f a = (a, nb_occur a l) in
                map f (ensemble l);;
val regroupe : 'a list -> ('a * int) list = <fun>
```

6. la fonction `taille` : là encore, on pourrait aisément le faire par une fonction récursive :

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

mais autant utiliser ce que nous avons déja défini : on `regroupe` la liste, on garde le deuxième élément (le nombre d'occurences) de chaque couple de la liste, et on additionne tout ça...

```
# let taille l =
        let snd (a,b) = b in
                somme (map snd (regroupe l));;
val taille : 'a list -> int = <fun>
```

7. la fonction `inclus` : cette fonction est assez difficile à écrire. En effet, il faut tenir compte du nombre d'occurences de chaque élément dans chacune des deux listes. Il vaut donc mieux travailler avec les listes « regroupées ». Dans ce cas, lorsqu'on a un élément et son nombre d'occurences dans la première liste, on regarde son nombre d'occurences dans la deuxième liste. Si la première liste est vide, elle est nécessairement incluse dans la deuxième.

   Commençons donc par tester l'inclusion d'une liste regroupée dans une liste normale :

```
# let rec r_inclus m1 l2 = match m1 with
        []      -> true |
        e::m    -> match e with
                   (a,b) -> ((nb_occur a l2 >= b) && (r_inclus m l2));;
val inclus : ('a * int) list -> 'a list -> bool = <fun>
```

   Maintenant, il ne reste plus grand'chose à faire pour obtenir la fonction voulue :

```
# let inclus l1 l2 = r_inclus (regroupe l1) l2;;
val inclus : 'a list -> 'a list -> bool = <fun>
```

8. la fonction `liste_egal` : il suffit de tester la double inclusion :

```
# let liste_egal l1 l2 = ((inclus l1 l2) && (inclus l2 l1));;
val liste_egal : 'a list -> 'a list -> bool = <fun>
```

9. la fonction `dernier` : la gestion d'erreurs n'étant pas encore au programme, il faudra supposer que la liste fournie en argument n'est pas vide. Nous écrivons donc :

```
# let rec dernier l = match l with
        [a]     -> a |
        e::m -> dernier m;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val dernier : 'a list -> 'a = <fun>
```

CAML nous préviens que, dans le cas de la liste vide, cela ne marche pas. Dans les autres cas, si la liste a un seul argument, c'est nécessairement le dernier, et sinon, le dernier élément de la liste est également le dernier élément de la queue de la liste...

10. la fonction `subst` : dans le cas général (substitution de toutes les occurences), il s'agit simplement d'appliquer une fonction de substitution à la liste :

```
# let subst a b l =
        let f x = if x=a then b else x in
                map f l;;
val subst : 'a -> 'a -> 'a list -> 'a list = <fun>
```

11. la fonction `subst1` : dans ce cas, il faut traiter la liste élément par élément, et ne remplacer que la première occurence.

```
# let rec subst1 a b l = match l with
        []      -> [] |
        e::m    -> if e=a then b::m else (e::subst1 a b m);;
val subst1 : 'a -> 'a -> 'a list -> 'a list = <fun>
```

Cette fonction parcours la liste jusqu'à ce qu'elle trouve une occurence de `a`. Si l'élément courant de la liste est un `a`, on le remplace par b et on recopie, telle quelle, la fin de la liste. Sinon, on va voir plus loin dans la liste...

12. la fonction `renverser` : on va simplement prendre le premier élément et l'ajouter à la fin de queue de la liste, qui sera renversée récursivement :

```
# let rec renverser l = match l with
        []      -> [] |
        e::m    -> (renverser m)@[e];;
val renverser : 'a list -> 'a list = <fun>
```

Ici, on a besoin de l'opérateur de concaténatin de listes, car l'opérateur : : ne permet d'ajoute run élément qu'en début de liste.

Faisons quelques tests des fonctions que nous venons d'écrire :

```
# let l = [12; 18; 34; 12; 65; 34; 22; 18; 68; 12; 17; 65; 35;
23; 12; 34; 88; 68; 18; 15; 4; 12];;
# let m = [34 ;65; 22; 18; 17; 12; 68; 15; 4; 12; 12; 12];;
# appartient 88 m;;
- : bool = false
# appartient 88 l;;
- : bool = true
# ensemble l;;
- : int list = [4; 15; 88; 23; 35; 17; 68; 22; 65; 34; 18; 12]
# renverser (ensemble l);;
- : int list = [12; 18; 34; 65; 22; 68; 17; 35; 23; 88; 15; 4]
# ensemble m;;
- : int list = [4; 15; 68; 12; 17; 18; 22; 65; 34]
# regroupe l;;
- : (int * int) list = [4, 1; 15, 1; 88, 1; 23, 1; 35, 1; 17, 1;
68, 2; 22, 1; 65, 2; 34, 3; 18, 3; 12, 5]
# regroupe m;;
- : (int * int) list = [4, 1; 15, 1; 68, 1; 12, 4; 17, 1; 18, 1;
22, 1; 65, 1; 34, 1]
# taille l;;
- : int = 22
# taille m;;
- : int = 12
# inclus m l;;
- : bool = true
inclus l m;;
- : bool = false
# subst 18 23 l;;
- : int list = [12; 23; 34; 12; 65; 34; 22; 23; 68; 12; 17; 65;
35; 23; 12; 34; 88; 68; 23; 15; 4; 12]
# subst1 18 13 l;;
- : int list = [12; 13; 34; 12; 65; 34; 22; 18; 68; 12; 17; 65;
35; 23; 12; 34; 88; 68; 18; 15; 4; 12]
```