Feuille d'exercices n°4

Cette feuille d'exercice porte sur l'utilisation de fonctions d'ordre supérieur (itérateurs) sur les listes.

Exercice I : Schéma d'application

On considère les trois fonctions suivantes:

`inverse (xs:float list) : float list`

telle que `(inverse [x1; ..; xn])` vaut `[ 1.0 /. x1; ..; 1.0 /. xn]`.

`dpoints (xs:int list) (a:int) (b:int) : (int * int) list`

telle que `(dpoints [x1; ..; xn] a b)` vaut `[ (x1, a*x1 + b); ..; (xn, a*xn + b) ]`.

`ecrete (xs:int list) : int list`

telle que `(ecrete [x1; ..;xn])` vaut `[x1'; .., xn']` où, pour tout `i` entre `1` et `n`:

- `xi'` = `-10`, si `xi` < -10
- `xi'` = `10`, si $10 <$ `xi`
- `xi'` = `xi`, sinon

Définir ces trois fonctions en utilisant l'itérateur

```
List.map (f:'a -> 'b) (xs:'a list) : 'b list
```

.

Exercice II : Schéma de filtrage

On considère les trois fonctions suivantes

`no_x (strs:string list) : string list`

telle que `(no_x strs)` donne la listes de chaînes de `strs` qui ne contiennent pas le caractère `'x'`.

`list_inter (xs:int list) (m1:int) (m2:int) : int list`

telle que `(list_inter xs m1 m2)` donne la liste des valeurs de `xs` comprises (au sens large) entre `m1` et `m2`.

`sum_less (xys: (int * int) list) (m:int) : (int * int) list`

telle que `(sum_less xys m)` donne la liste des couple `(x,y)` de `xys` tels que `x` + `y` $<$ `m`

Définir ces trois fonctions en utilisant l'itérateur

```
List.filter (p:'a -> bool) (xs:'a list) : 'a list
```

Nota: l'application `(String.contains str c)` vaut `true` si la chaîne `str` contient le caractère `c` et `false`, sinon.

Exercice III : Schéma d'accumulation

On utilise dans cet exercice les itérateurs

```
List.fold_right (f:'a -> 'b -> 'b) (xs:'a list) (z:'b) : 'b
```

```
List.fold_left (f:'a -> 'b -> 'a) (r:'a) (xs:'b list) : 'a
```

On considère les trois fonctions suivantes

```
sum_tuples (xys:(int * int) list) : int
```

telle que `(sum_tuples [(x1,y1); ..; (xn,yn)])` donne la somme `x1 + y1 + .. + xn + yn`.

```
sum_assoc (xs:'a list) (alist:('a * int) list) : int
```

telles que `(sum_assoc [x1; ..; xn] alist)` donne la somme `(List.assoc x1 alist)+ .. +(List.assoc xn alist)`.

```
parenthese (strs:string list) : string
```

qui donne la chaîne de caractères obtenue en mettant entre paranthèse et en concaténant les chaînes de `strs`. Par exemple: `(parenthese ["do"; "re"; "mi"])` vaut `"(do)(re)(mi)"`.

Pour chacune de ces fonctions, donnez une définition qui utilise `List.fold_right` et une autre qui utilise `List.fold_left`.

Exercice IV : Listes d'association

On redéfinit dans cet exercices quelques fonctions portant sur les *listes d'association*. Certaines d'entre elles existent déjà dans la bibilothèque standard. Nous les redéfinissons à titre d'exercice.

Une liste d'association est une liste de couples. Le type d'une liste d'association est une instance du type général (polymorphe) `('a * 'b) list`. Le premier élément du couple est appelé *clé* et le second *valeur*. On s'en sert pour stocker des informations (les valeurs) que l'on retrouve à l'aide des clés.

Q1 – Définir la fonction

```
assoc (k:'a) (kvs:('a * 'b) list) : 'b
```

qui donne la valeur associée à la clé `k` dans la liste d'association `kvs`. La fonction déclenche l'exception `Not_found` si une telle valeur n'exsite pas.

Q2 – Définir la fonction

```
remove_assoc (k:'a) (kvs:('a * 'b) list) : ('a * 'b) list
```

qui donne la liste `kvs` privée du premier couple, s'il existe, correspondant à la clé `k`.

Q3 – Définir la fonction

```
set_assoc (k:'a) (v:'b) (kvs:('a * 'b) list) : ('a * 'b) list
```

qui donne la liste obtenue en remplaçant dans `kvs` l'ancienne valeur, si elle existe, associée à `k` par `v`. Si une tellevaluer n'existe pas, la nouvelle association `(k,v)` est ajoutée.

Exemple:

`(set_assoc 123 "info" [])` donne `[(123,"info")]`

`(set_assoc 234 "nouvelle info" [(123,"info sur 123"); (234,"info"); (567, "autre info")])` donne `[(123,"info sur 123"); (234,"nouvelle info"); (567, "autre info")]`