AP1 - 2009 - feuille de TP 6 / TD 7

Exercice 1 (**Trions ...**)

1. Définir une fonction `lex` qui définit l'ordre lexicographique sur des couples quelconques de type `t1*t2`. Elle va donc prendre en paramètre la fonction de comparaison sur les éléments de type `t1` et la fonction de comparaison sur les éléments de type `t2`. Elle doit fournir en résultat la fonction d'ordre lexicographique sur les couples de type `t1*t2`. La fonction attendue est donc de type `('a -> 'a -> int) -> ('b -> 'b -> int) -> ('a*'b -> 'a*'b -> int)`.

   Les fonctions de comparaison en argument et en résultat doivent retourner 0 si les arguments sont égaux, un entier positif si le 1er argument est plus grand, un entier négatif si le 2eme argument est le plus grand.

   ```
   let lex comp1 comp2 = function (x1, y1) -> function (x2, y2) ->
    let c = comp1 x1 x2 in if c = 0 then comp2 x2 y2 else c;;
   ```

2. Utilisez la fonction de tri sort du module `List` pour trier des couples d'entiers et de booléens selon l'ordre lexicographique associé.

   Rappel : dans la bibliothèque standard on trouve le module `List`. Dans ce module on fonction `sort` dont voici ce qu'il est dit :

   ```
   val sort :  ('a -> 'a -> int) -> 'a list -> 'a list
   ```

   Sort a list in increasing order according to a comparison function. The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller. The resulting list is sorted in increasing order.

   ```
   let compbool b1 b2 = if b1=b2 then 0 else if b1 then 1 else -1;;
   let tri_ent_bool l = List.sort (lex (-) compbool) l;;
   ```

3. Ecrire une fonction `est_triée` qui prend en paramètre une fonction de comparaison et une liste l et retourne true si la liste l est triée selon la fonction de comparaison paramètre, false sinon.

   Trier une liste d'entiers dans l'ordre décroissant avec la fonction List.sort. Vérifier que le résultat obtenu est bien trié.

   ```
   let rec est_triee comp l = match l with
    | x::y::r -> if comp x y <= 0  then est_triee comp (y::r) else false
    | _ -> true;;
   val est_triee : ('a -> 'a -> int) -> 'a list -> bool = <fun>

   let dec x y = y - x in  est_triee dec (List.sort dec [5;4;1;10]);;
   ```

Exercice 2 (**Application partielle**)

```
let plus = function x -> function y -> x + y;;
(*ou encore*)
let plus = (+);;
```

Que fait la fonction suivante définie par `let ttt l = List.map (plus 2) l` ?

La fonction `ttt` ajoute 2 à tous les éléments d'une liste. On remarque ici que `plus 2` (il s'agit d'une application partielle) est une fonction de type `int -> int`. C'est la fonction qui à tout y associe 2+y

Exercice 3

1. Ecrire une fonction `flatten` qui prend en paramètre une liste l de listes et retourne la concaténation des éléments de l. Elle sera de type `'a list list -> 'a list`.

   Vous l'écrirez d'abord comme une fonction récursive. Puis vous l'écrirez en utilisant `fold_left` ou `fold_right`.

2. Ecrire la fonction `prodcart` qui réalise le produit cartésien de deux listes quelconques. Par exemple, `prodcart [1;0] [4;5]` retourne la liste `[(1, 4); (1, 5); (0, 4); (0, 5)]`. Vous l'écrirez en utilisant la fonctionnelle `map` et la fonction `flatten`.

```
# let rec flatten l = match l with
  [] -> []
  | x::r -> x@(flatten r);;
val flatten : 'a list list -> 'a list = <fun>
# let l = [[1;2;3];[];[3]];;
val l : int list list = [[1; 2; 3]; []; [3]]
# flatten l;;
- : int list = [1; 2; 3; 3]
# let flatten l = List.fold_right (@) l [];;
val flatten : 'a list list -> 'a list = <fun>
# flatten l;;
- : int list = [1; 2; 3; 3]

let prodcart l1 l2 = flatten (List.map (function a -> (function b -> List.map (a,b) l2) l1);;

prodcart [1;0] [4;5];;
- : (int * int) list = [(1, 4); (1, 5); (0, 4); (0, 5)]
```

Exercice 4

Ecrire une fonction `puis` qui calcule f^n avec f^0=identité et $f^n = fof^{n-1}$ pour $n > 0$. Elle prend en paramètre f, n (n est supposée être un entier naturel) et retourne une fonction. On pourra utiliser la fonction `compose` donnée ci-dessous pour composer deux fonctions.

```
let compose f g x  = f(g x);;
```

En utilisant la fonction `puis`, écrire une fonction qui calcule la dérivée nième d'une fonction quelconque. Puis la dérivée 3ième de sinus, soit `sin'''`. Calculer `sin''' pi` en approximant pi par (4. *. atan 1.);;

Exercice 5 (**La date du lendemain**)

Ecrire la fonction qui calcule la date du lendemain.

Exercice 6 (**Cocktails et jus de fruits**)

Une boisson est soit un jus de fruits, soit un sirop, soit un alcool, soit un cocktail qui est un mélange de boissons. Chaque ingrédient de base (jus, sirop, alcool) a un nom. Un alcool est également caractérisé par son degré.

1. Définir le type `boisson`.

2. Définir le punch comme un mélange de rhum blanc à 40° et de sirop de canne.

```
type boisson =
| Jus of string | Sirop of string | Alcool of string * int |
Cocktail of boisson list;;

let punch = Cocktail [Alcool ("rhum blanc", 40); Sirop "canne"];;
```

3. Définir une fonction qui teste si une boisson est alcoolisée. En donner deux versions : avec deux fonctions mutuellement récursives et avec un itérateur.

```
let rec alcoolisé b = match b with
| Alcool _ -> true
| Jus _ | Sirop _ -> false
| Cocktail bl -> alcoolisé_liste bl
and alcoolisé_liste bl = match bl with
| [] -> false
| b::bl' ->
    (alcoolisé b) || (alcoolisé_liste bl');;
alcoolisé nectar_de_fruits;;
alcoolisé mixture;;
```

Il nous faut donc trouver ici avec quel `f` et quel `e` utiliser la fonction `fold_right`. Cependant, l'opérateur binaire qui combine les éléments de la liste des ingrédients du cocktail est ici plus difficile à deviner que dans les exemples précédents.

Pour la fonction `alcoolisé`, on doit avoir :

```
alcoolisé (Cocktail [b_1; b_2; \ldots ;b_n]) =
(alcoolisé b_1) ||
   (alcoolisé b_2) ||\ldots || (alcoolisé b_n)=
(alcoolisé b_1) ||
   ((alcoolisé b_2) ||(\ldots || ((alcoolisé b_n)|| false)\ldots))
```

Si on introduit l'opérateur Θ défini par $a\ \Theta\ b$ = `(alcoolisé a) || b`, l'expression précédente peut être réécrite en :

`b_1` Θ `(b_2` Θ `(...` Θ `(b_n` Θ $false$`)...))`

```
let rec alcoolisé b = match b with
| Alcool _ -> true
| Cocktail bl ->
     List.fold_right
      (function a -> function b -> (alcoolisé a) || b)
       bl false
| _ -> false;;
```

On remarquera que la fonction `alcoolisé` reste récursive (de manière directe).