Programmation Fonctionnelle :
des concepts aux applications web
(NFP119)

Listes et fonctionnelles

1. **map.** Rappelons les définitions suivantes vue en cours :

   ```
   let rec doublist l = match l with [] -> [] | (h::t) -> (2*h)::(doublist t)
   ```

   ```
   let rec inclist l  = match l with [] -> [] | (h::t) -> (h+1)::(inclist t)
   ```

   La fonction `doublist` double chaque élément de la liste, alors que `inclist` incrémente les éléments. Par exemple :

   ```
   doublist [1;2;3;4] = [2;4;6;8]
   inclist  [1;2;3;4] = [2;3;4;5]
   ```

   Ecrire une fonction `map : ('a -> 'b) -> 'a list -> 'b list` qui prend en paramètre la fonction `f` à appliquer à chaque élément de la liste; et qui généralise ainsi `doublist` et `inclist` :

   ```
   let doublist = map (fun x -> 2 * x)
   let inclist = map (fun x -> x + 1)
   ```

   Autrement dit, `map` doit vérifier l'équation suivante :

   ```
   map f [x1; x2; ...; xn] = [(f x1);(f x2);...;(f xn)]
   ```

2. Modifier la fonction `map` de la première question pour qu'elle prenne en paramètre deux listes. Son type devient alors `('a * 'b -> 'c) -> 'a list * 'b list -> 'c list`. Donner deux définitions de cette fonction (appelons la `map2`), l'une par récurrence et l'autre en utilisant la fonction `zip` vue en cours (et la fonction `map` de la première question).

3. La fonction `filter: ('a -> bool) -> 'a list -> 'a list` prend en paramètre un prédicat (i.e. une fonction booléenne) et extrait d'une liste tous les éléments qui vérifient ce prédicat. Par exemple, si l'on définit :

   ```
   let even n = (n mod 2 = 0)
   ```

   En appliquant alors `filter even` à la liste `[1; 2; 3; 4; 5; 6]` on doit obtenir la liste `[2; 4; 6]`. Ecrire la fonction `filter`. Peut-on la définir facilement à partir de `map` ? Que vaut l'expression `map even [1; 2; 3; 4; 5; 6]` ?

4. **fold.** Rappelons les définitions suivantes vues en cours :

   ```
   let rec sum l    = match l with [] -> 0  | (h::t) -> h + (sum t)
   ```

   ```
   let rec concat l = match l with [] -> [] | (h::t) -> h @ (concat t)
   ```

   La fonction `sum` calcule la somme des éléments d'une liste, alors que `concat` concatène une liste de listes. Par exemple :

   ```
   sum [10; 20; 30] = 60
   concat [[1; 2]; [3; 4]; [5; 6; 7]] = [1; 2; 3; 4; 5; 6; 7]
   ```

   Ecrire une fonction `fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` qui prend en paramètre la fonction `f` à appliquer à chaque élément de la liste; et qui généralise ainsi `sum` et `concat`.

```
let sum l = fold_right (fun x y -> x + y) l 0
let concat l = fold_right (fun x y -> x @ y) l []
```

Autrement dit, `fold_right` doit vérifier l'équation suivante :

```
fold_right f [a1; ...; an] init = f a1 (f a2 (... (f an init) ...))
```

De la même manière, définir `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` qui doit vérifier l'équation suivante :

```
fold_left f init [b1; ...; bn] = f (... (f (f init b1) b2) ...) bn
```

On définit maintenant :

```
let sum' l = fold_left (fun x y -> x + y) 0 l
let concat' l = fold_left (fun x y -> x @ y) [] l
```

Que valent les expressions suivantes ?

```
sum' [10; 20; 30]
concat' [[1; 2]; [3; 4]; [5; 6; 7]]
```

5. On définit maintenant :

```
let mystery l = fold_right (fun h t -> h::t) l []
let mystery' l = fold_left (fun t h -> h::t) [] l
```

Que font ces fonctions ? Quelle sont les différences entre `fold_left` et `fold_right` ?

6. Vérifier sur des exemples que :

```
fold_right f l init = fold_left (fun x y -> f y x) init (rev l)
```