TP 3 Programmation fonctionnelle

Exercice 1. Donner les définition de types suivantes :

1. Définir des listes (polymorphes) non vides

   Solution.

   ```
   type 'a nelist =
     | Nil of 'a
     | Cons of 'a * 'a nelist
   ```

   □

2. Définir un record point avec 2 champs x and y de type int. Puis implémenter la fonction `move_left` qui décrément la coordonnée x

   Solution.

   ```
   type point  = { x : int; y : int }
   let move_left p = { x = p.x - 1; y = p.y }
   ```

   □

3. Définir le type énuméré de 4 points cardinaux

   Solution.

   ```
   type pc = N | S | E | W
   ```

   □

4. Définir la fonction `multi_move : pos -> pc list-> pos`

   Solution.

   ```
   let move p = function
   | N -> { x = p.x; y = p.y - 1 }
   | S -> { x = p.x; y = p.y + 1 }
   | W -> { x = p.x - 1; y = p.y }
   | E -> { x = p.x + 1; y = p.y }

   let rec multi_move p = function
   | [] -> p
   | pc :: pcs -> multi_move (move p pc) pcss
   ```

   □

Exercice 2. Supposons la définition de type suivante :
`type student = {first_name : string; last_name : string; gpa : float}`
Écrire une fonction qui extrait le nom de l'étudiant et une fonction qui crée un record étudiant

Solution.

```
type student = { first_name : string ; last_name : string ; gpa : float }

(* expression with type [student] *)
let s =
  { first_name = "Ezra"; last_name = "Cornell"; gpa = 4.3 }

(* expression with type [student -> string * string] *)
let get_full_name student =
  student.first_name, student.last_name

(* expression with type [string -> string -> float -> student] *)
let make_stud first last g =
  { first_name = first; last_name = last; gpa=g }
```

□

Exercice 3. Écrire une fonction `take : int -> 'a list -> 'a list` qui renvoie les n premiers éléments d'une liste. Si la liste a moins de n éléments, renvoie-les tous.

Solution.

```
let rec take n lst =
  if n = 0 then [] else match lst with
    | [] -> []
    | x :: xs -> x :: take (n - 1) xs
```

□

Exercice 4. Écrivez une fonction `drop : int -> 'a list -> 'a list` qui renvoie tous les éléments de liste sauf les n premiers. Si la liste a moins de n éléments, renvoie la liste vide.

Solution.

```
let rec drop n lst =
  if n = 0 then lst else match lst with
    | [] -> []
    | _ :: xs -> drop (n - 1) xs
```

□

Exercice 5. Une fonction à récursivité terminale est une fonction où l'appel récursif est la dernière instruction à être évaluée.

Réviser vos solutions pour take et drop afin qu'elles soient récursives terminales (suggestion : utiliser des accumulateurs), si elles ne le sont pas déjà.

Solution.

```
let rec take_rev n xs acc =
  if n = 0 then acc else match xs with
    | [] -> acc
    | x :: xs' -> take_rev (n - 1) xs' (x :: acc)
```

□

Exercice 6. Un arbre binaire est une type de donné :

```
type 'a bintree =
| Nil
| Node of 'a * 'a bintree * 'a bintree;;
```

1. Trouver l'hauteur d'un arbre
2. Trouver la valeur maximale contenue dans un arbre binaire d'entiers (utiliser les types options)
3. Convertir un arbre en une liste
4. Convertir une liste en un arbre

Solution.

```
type 'a bintree =
  | Nil
  | Node of 'a * 'a bintree * 'a bintree;;

let tree = Node(3, Node(2,Nil,Nil),Node(5, Node(4,Nil,Nil),Nil))

let rec height t =
  match t with
  | Nil -> 0
  | Node(_,t1,t2) -> 1 + max (height t1) (height t2)

let rec maxtree t =
  match t with
  | Nil -> None
  | Node(x,t1,t2) ->
      match (maxtree t1, maxtree t2) with
      | (None, None) -> Some x
      | (None, Some y) -> Some (max x y)
      | (Some y, None) -> Some (max x y)
      | (Some y, Some z)-> Some (max x (max y z))

let rec tolist t  =
  match t with
  | Nil -> []
  | Node(x,t1,t2) -> x::(tolist t1)@(tolist t2)

let rec totree l =
  match l with
  | [] -> Nil
  | x::xs -> Node(x,Nil,(totree xs))
```

□