TP n°4 - Correction

Arbres binaires

1 Listes (révisions)

Rappelons que la fonction `List.map` permet d'appliquer une fonction a chacun des elements d'une liste. Par exemple `List.map (fun x -> x + 1) [3; 2; 7]` s'evalue en `[4; 3; 8]`.

1. Ecrire une fonctions `enum` telle que `enum n` renvoie la liste `[0;1;...;n]`.
2. A partir de cette fonction et de `List.map`, ecrire une fonction `enum_droite` telle que `enum_droite i n` renvoie la liste `[(i,0);(i,1);...;(i,n)]`.
3. En deduire le code de `enum_paires : int -> int -> int list` telle que `enum_paires n p` renvoie la liste des elements de $\{0,\ldots,n\} \times \{0,\ldots,p\}$ dans l'ordre lexicographique. Exemple :

   `enum_paires 1 2 = [(0,0);(0,1);(0,2);(1,0);(1,1);(1,2)]`
4. En utilisant cette fonction, ecrire une fonction `table_mult n` calculant la table de multiplication de 0 a n, sous la forme d'une liste de triplets $(i, j, i \times j)$. Exemple :

```
table_mult 2 = [(0,0,0);(0,1,0);(0,2,0);
                (1,0,0);(1,1,1);(1,2,2);
                (2,0,0);(2,1,2);(2,2,4)]
```

Correction :

```
let rec enum n =
  if n = 0 then [0] else (enum (n-1))@[n]
;;
let rec enum_droite i n = List.map (fun x -> (i,x)) (enum n)
;;
let rec enum_paire n p =
  if n = 0 then enum_droite 0 p
  else (enum_paire (n-1) p)@(enum_droite n p)
;;
let rec tabl_mult n =
  List.map (fun (x,y) -> (x,y,x*y)) (enum_paire n n);;
```

2 Arbres binaires

On peut e ectuer en Caml des *declarations de types*, c'est-a-dire creer de nouveaux types de valeurs. Les arbres binaires etiquees par des entiers peuvent par exemple être representes en Caml a l'aide de la declaration de type suivante :

```
type tree = Nil | Node of int * tree * tree;;
```

`Nil` et `Node` seront appeles les *constructeurs* du type `tree`. Une fois ce type declare, on peut construire des valeurs de type `tree` :

```
Nil;;
Node (42, Nil, Nil);;
Node (42, Node(10, Nil, Nil), Nil);;
```

Comme pour les listes, une fonction peut être de nie par cas sur la forme d'un arbre :

```
let rec f a = match a with
  Nil       -> ...
| Node(n,g,d) -> ...
;;
```

Exercice 1

Ecrire les fonction suivantes :

- { `taille : tree -> int` renvoyant la taille d'un arbre, c'est-a-dire son nombre de noeuds internes.
- { `hauteur : tree -> int` renvoyant la hauteur d'un arbre, c'est-a-dire la longueur de sa plus longue branche.
- { `detect : tree -> int -> bool`, telle que `detect a n` renvoie `true` si et seulement si l'un des n uds de l'arbre est etiquete par n.
- { `complet : tree -> bool` determinant si un arbre est complet, c'est-a-dire si toutes ses feuilles sont a la même profondeur,

Correction :

```
type tree = Nil | Node of int * tree * tree;;

let rec taille t = match t with
  Nil -> 0
| Node(_,g,d) -> 1 + (taille b) + (taille c)
;;

let rec hauteur t = match t with
  Nil -> 1
| Node(_,g,d) -> 1 + max (hauteur g) (hauteur d)
;;

let rec detect t n = match t with
  Nil -> false
| Node(m,g,d) -> (n = m) || (detect g n) || (detect d n)
;;

let rec complet t = match t with
  Nil -> true
| Node(_,g,d) -> (complet g) && (complet d) && (hauteur g = hauteur d)
;;
```

Exercice 2

On rappelle qu'un *arbre binaire de recherche* est un arbre dans lequel pour *chaque* noeud, on a la propriete suivante :

- soit n l'etiquette de ce n ud :
  - toutes les etiquettes du ls gauche de ce n ud sont $<$ n.
  - toutes les etiquettes du ls droit de ce n ud sont $>$ n.

Ecrire les fonctions suivantes :

- `detect_abr : int -> tree -> bool` tel que si a est un ABR et n un entier, `detect_abr` n a renvoie `true` si n apparaît dans l'arbre, et `false` sinon.
  L'exploration de a devra être minimale, en tenant compte du fait qu'il s'agit d'un ABR.
- `ajout_abr : int -> tree -> tree` tel que si a est un ABR et n est un entier, `ajout_abr` renvoie l'arbre a dans lequel on a ajoute n s'il n'y est pas deja. Le resultat doit être encore un ABR. Si n est deja dans l'arbre a, cette fonction doit renvoyer simplement a.
- `est_abr : tree -> bool` prenant en argument un arbre quelconque, et determinant s'il s'agit d'un ABR. A noter que cette fonction necessite de declarer plusieurs fonctions auxiliaires, qui pourront être declarees en fonctions locales.

Correction :

```
let rec detect_abr n a = match a with
  Nil -> false
| Node(m,g,d) ->
    m = n || (n < m && detect_abr n g) || (n > m && detect_abr n d)
;;
let rec ajout_abr n a = match a with
  Nil -> Node(n,Nil,Nil)
| Node(m,g,d) ->
    if m = n then a else
    if n < m then Node(m, ajout_abr n g,d) else
    Node(m, g, ajout_abr n d)
;;
(* est_abr : *)
(* une correction possible *)
let rec min_abr a = match a with
  Node (n, Nil, _) -> n
| Node (_, g, _)   -> min_abr g
;;
let rec max_abr a = match a with
  Node (n, _, Nil) -> n
| Node (_, _, d)   -> max_abr d
;;
let est_abr a = match a with
  Nil -> true
| Node (n, g, d) ->
    (g = Nil || (est_abr g && max_abr g < n) &&
    (d = Nil || (est_abr d && min_abr d > n)
;;

(* une autre, avec conversion de l'arbre en liste *)
let rec list_of_arbre a = match a with
  Nil -> []
| Node (n, g, d) -> (list_of_arbre g)@[n]@(list_of_arbre d)
;;
let rec est_triee l = match l with
  [] -> true
| [x] -> true
| x::y::l' -> x < y && est_triee (y::l')
```

```
;;

let est_abr a = est_triee (list_of_arbre a)
;;

(* une autre, avec une seule lecture de l'arbre, *)
(* et sans conversion en un autre type de donnees   *)
let est_abr a =
  let rec sup n a = match a with
    Nil -> true
  | Node (q,g,d) -> q > n && (sup q d) && (entre n q g)
  and entre n p a = match a with
    Nil -> true
  | Node (q,g,d) -> n < q && q < p && (entre n q g) && (entre q p d)
  and inf n a = match a with
    Nil -> true
  | Node (q,g,d) -> q < n && (inf q g) && (entre q n d)
  in match a with
  Nil -> true
| Node (n,g,d) -> (sup n d) && (inf n g)
;;
```