Université Paris Diderot
Informatique

Programmation Fonctionnelle
Année 2015-2016

TP n°3 - Correction

Arbres binaires

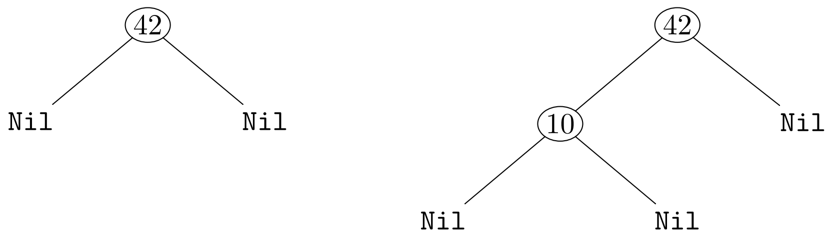
Il est possible d'effectuer en OCaml des *déclarations de types*, c'est-à-dire créer de nouveaux types de valeurs. On peut ainsi représenter en OCaml des arbres binaires à l'aide de la déclaration de type suivante :

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

`Nil` et `Node` seront appelés les *constructeurs* du type des arbres binaires. Cette définition est *polymorphe* : les éléments contenus dans ces arbres ont un type `'a` qui n'est pas connu à l'avance. `Nil` est l'arbre vide. Les éléments se trouvent dans tous les nœuds de l'arbre. Une *feuille* est un arbre contenant exactement un élément. Une fois ce type déclaré, on peut construire par exemple des arbres binaire d'entiers, de type `int tree` :

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

qui correspondent respectivement aux représentations graphiques suivantes (le premier arbre est une feuille) :



Ou bien encore on peut construire des arbres de chaînes, de type `string tree` :

```
Node ("abc", Nil, Node ("def", Nil, Nil));;
```

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

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

Exercice 1. Écrire les fonction suivantes (Ne pas oublier de les tester sur des cas pertinents) :

1. `taille : 'a tree -> int` renvoyant la taille d'un arbre, c'est-à-dire son nombre de noeuds.
2. `hauteur : 'a tree -> int` renvoyant la hauteur d'un arbre, c'est-à-dire la longueur de sa plus longue branche. On suppose que `Nil` est de hauteur 0 et une feuille de hauteur 1.

3. `mem : 'a -> 'a tree -> bool`, telle que `mem x a` renvoie `true` si et seulement si l'un des nœuds de l'arbre `a` est étiqueté par `x`.
4. `complet : 'a tree -> bool` déterminant si un arbre est complet, c'est-à-dire si toutes ses feuilles sont à la même profondeur et tous les nœuds ont soit 0 fils soit 2 fils.
5. `elements : 'a tree -> 'a list`, qui renvoie la liste des éléments présents dans l'arbre, dans l'ordre de leurs apparitions de gauche à droite dans l'arbre. Sauriez-vous écrire une version sans utiliser la concatenation (`@`) des listes ?

Correction :

```
(* 1 *)
let rec taille = function
 | Nil -> 0
 | Node (_,g,d) -> 1 + taille g + taille d ;;

(* 2 *)
let rec hauteur = function
 | Nil -> 0
 | Node (_,g,d) -> 1 + max (hauteur g) (hauteur d) ;;

(* 3 *)
let rec mem x = function
 | Nil -> false
 | Node (y,g,d) -> x=y || mem x g || mem x d ;;

(* 4 *)
(* Il n'est pas suffisant de juste tester que les deux sous-arbres sont complets *)
let rec complet = function
  | Nil -> true
  | Node(_,g,d) -> complet g && complet d && hauteur g = hauteur d ;;

(* version alternative: *)
let complet a =
  let rec check h = function
   | Nil -> h=0
   | Node (_,g,d) -> check (h-1) g && check (h-1) d
  in check (hauteur a) a ;;

(* petite optimisation possible ci-dessus :
   replacer (hauteur a) par (prof_gauche a), avec: *)
let prof_gauche = function
  | Nil -> 0
  | Node (_,g,_) -> 1 + prof_gauche g

(* 5 *)
let rec elements = function
  | Nil -> []
  | Node (x,g,d) -> (elements g) @ [x] @ (elements d) ;;

(* version sans @ *)
let elements a =
  let rec visit a acc = match a with
   | Nil -> acc
   | Node (x,g,d) -> visit g (x :: visit d acc)
```

```
  in visit a [] ;;
```

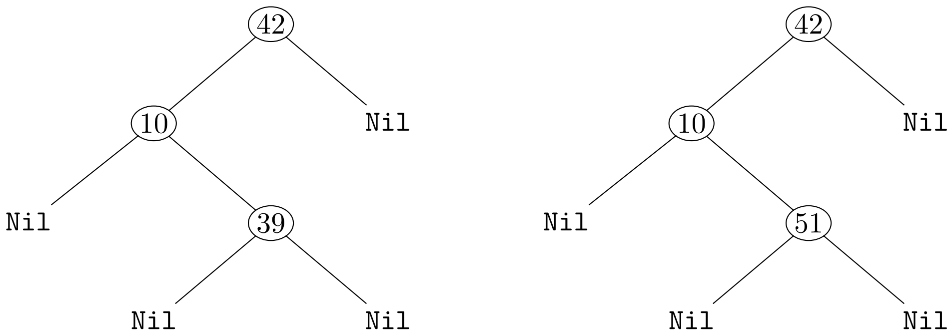
Un *arbre binaire de recherche* (ABR) est un arbre dans lequel *tous* les nœuds `Node(x,g,d)` vérifient la propriété suivante :

— toutes les étiquettes apparaissant dans le fils gauche `g` sont strictement inférieures à `x`.

— toutes les étiquettes apparaissant dans le fils droit `d` sont strictement supérieures à `x`.

La comparaison utilisée ici est la comparaison générique $<$ d'OCaml.

Par exemple, parmi les deux arbres binaires d'entiers suivants, le premier est un ABR et le deuxième non (pourquoi ?)



Exercice 2. Écrire les fonctions suivantes :

1. `mem_abr : 'a -> 'a tree -> bool` qui attend un élément `x` et un ABR `a`, et renvoie `true` si et seulement si `x` apparaît dans l'arbre `a`. L'exploration de `a` devra être minimale, en tenant compte du fait qu'il s'agit d'un ABR.
2. `add_abr : 'a -> 'a tree -> 'a tree` qui permet d'ajouter un élément dans un ABR s'il n'y est pas déjà. Le résultat devra encore être un ABR. Si l'élément est déjà dans l'arbre, cette fonction laissera l'arbre à l'identique.
3. `is_abr : 'a tree -> bool` prenant en argument un arbre quelconque, et déterminant s'il s'agit d'un ABR. Il pourra être utile de définir plusieurs fonctions auxiliaires.
4. `make_abr : 'a list -> 'a tree` transformant une liste triée d'éléments en un ABR contenant ces mêmes éléments. Vérifier sur des exemples que `elements (make_abr l) = l`. Pour limiter au maximum la profondeur de l'arbre obtenu, il pourra être utile d'écrire d'abord une fonction découpant une liste en deux moitiés de tailles similaires.

Les fonctions des points 2 et 4 de cet exercices renvoient des arbres. Pour les tester, il est possible d'utiliser la fonction `print_tree : int tree -> unit` qui affiche les valeurs de types `int tree`. Cette fonction est disponible sur didel, dans le fichier `print_tree.ml`, contenant aussi les définitions d'un certain nombre de valeur de type `int tree`.

Correction :

```
(* 1 *)
let rec mem_abr x = function
 | Nil -> false
 | Node (y,g,d) ->
   if x = y then true
   else if x < y then mem_abr x g
   else (* x > y *) mem_abr x d ;;
```

```
(* 2 *)
let rec add_abr x a = match a with
 | Nil -> Node (x,Nil,Nil)
 | Node (y,g,d) ->
   if x = y then a
   else if x < y then Node (y, add_abr x g, d)
   else (* x > y *) Node (y, g, add_abr x d) ;;

(* 3 *)
let rec majorant x = function
 | Nil -> true
 | Node (y,g,d) -> y<x && majorant x g && majorant x d ;;

let rec minorant x = function
 | Nil -> true
 | Node (y,g,d) -> x<y && minorant x g && minorant x d ;;

(* Rq: majorant et minorant pourraient etre faits via une seule
   fonction d'ordre superieur... *)

let rec is_abr = function
 | Nil -> true
 | Node (x,g,d) -> is_abr g && is_abr d && majorant x g && minorant x d ;;

(* version plus efficace : on encadre chaque arbre,
   avec des bornes utilisant le type option *)

let encadrant inf sup x =
 (match inf with None -> true | Some i -> i < x) &&
 (match sup with None -> true | Some s -> x < s);;

let rec check_abr inf sup = function
  | Nil -> true
  | Node (x,g,d) ->
    encadrant inf sup x &&
    check_abr inf (Some x) g &&
    check_abr (Some x) sup d;;

let is_abr a = check_abr None None a ;;

(* 4 *)

(* version naive, ne construisant pas un arbre de hauteur minimale *)

let rec make_abr = function
 | [] -> Nil
 | x :: l -> add_abr x (make_abr l) ;;

let moities l =
  let rec decoupe n l = match n,l with
    | 0,_ -> [],l
    | _,[] -> [],[]
    | _,x::l -> let l1,l2 = decoupe (n-1) l in x::l1, l2
```

```
  in decoupe ((List.length l)/2) l ;;

(* Note : grace a l'arrondi lors du /2,
   la 1ere moitie est moins longue que la 2nde moitie (ou =) *)

let rec make_abr l = match moities l with
 | _, [] -> Nil
 | l1, x::l2 -> Node (x, make_abr l1, make_abr l2);;
```

Exercice 3.

1. Sans déclarer de fonctions auxiliaires et sans vous servir des fonctions prédéfinies, écrire :

   `forall_labels : ('a -> bool) -> 'a tree -> bool`

   telle que `forall_labels p a` renvoie `true` si et seulement si chaque étiquette `x` de `a` vérifie
   `(p x) = true`. A partir de cette seule fonction, écrire une fonction

   `is_uniform : 'a -> 'a tree -> bool`

   tel que `is_uniform v a` renvoit `true` si et seulement si chaque étiquette de `a` est égale à `v`. Cette fonction ne devra pas être déclarée comme récursive, mais doit déléguer entièrement la récurrence à la précédente.
   Correction :

   ```
   let rec forall_labels p a = match a with
       Nil -> true
     | Node(x, g, d) -> p x && (forall_labels p g) && (forall_labels p d)
   ;;

   let is_uniform x a = forall_labels (fun n -> n = x) a;;
   ```

2. Toujours sans fonctions auxiliaires ou prédéfinies, écrire :

   `forall_subtrees : ('a -> 'a tree -> 'a tree -> bool) -> 'a tree -> bool`

   telle que `forall_subtrees p a` renvoie `true` si et seulement si pour chaque sous-arbre de `a` de la forme `Node(x, g, d)`, on a `p x g d = true`.
   Rappelons qu'un arbre est un *peigne droit* s'il est réduit à l'arbre vide, ou si son fils gauche est vide et son fils droit est un peigne droit. A partir de la seule fonction `forall_subtrees`, écrire :

   `est_peigne_droit : 'a tree -> bool`

   tel que `est_peigne_droit a` renvoit `true` si et seulement si `a` est un peigne droit, et déléguant entièrement la récurrence à la fonction `forall_subtrees`.
   Correction :

   ```
   let rec forall_subtrees pn a = match a with
       Nil -> true
     | Node(x, g, d) ->
          pn x g d &&
         (forall_subtrees pn g) &&
         (forall_subtrees pn d)
   ;;
   ```

```
let est_peigne_droit a =
  forall_subtrees (fun _ g _ -> g = Nil) a;;
;;
```

3. On définit pour les arbres un itérateur de la manière suivante :

   ```
   let rec fold_tree fn vf a = match a with
       Nil -> vf
     | Node(n, g, d) -> fn n (fold_tree fn vf g) (fold_tree fn vf d);;
   ```

   Noter que `vf` spécifie la valeur à renvoyer si l'arbre `a` est réduit à un arbre vide. Si c'est un noeud interne, l'itérateur est appliqué aux sous-arbres avec les mêmes arguments, et les valeurs de retour sont combinées à la valeur étiquetant la racine via la fonction `fn`.

   Quel est le type de `fold_tree` ?

4. En utilisant uniquement `fold_tree`, et en lui déléguant la récurrence, écrire :

   ```
   somme_etiquettes : int tree -> int
   ```

   renvoyant la somme des étiquettes d'un arbre étiqueté par des entiers. Avec les mêmes contraintes, écrire :

   ```
   map_tree : ('a -> 'b) -> 'a tree -> 'b tree
   ```

   telle que `map_tree f a` renvoie l'arbre obtenu en remplaçant chaque étiquette `x` de `a` par `(f x)`.

   Correction :

   ```
   let somme_labels a =
     fold_up (fun n sg sd -> n + sg + sd) 0 a
   ;;
   let map_tree f a =
     fold_up (fun n gm gd -> Node(f n, gm, gd)) a
   ;;
   ```