Université Paris Diderot
Informatique

Programmation Fonctionnelle
Année 2016-2017

TP n 3 - Correction

Arbres binaires

1 Arbres binaires

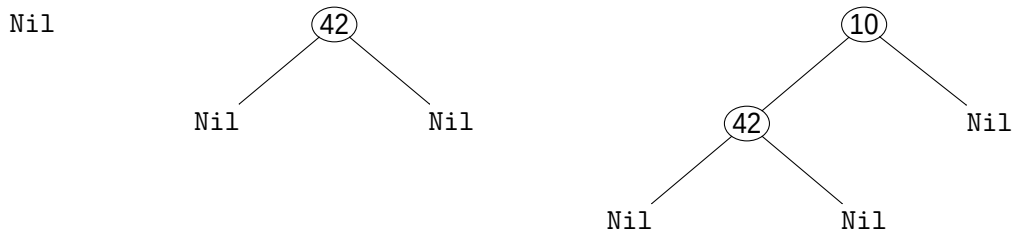
Il est possible d'effectuer en OCaml des *déclarations de types*, c'est-à-dire créer de nouveaux types de valeurs. Les arbres binaires dont les nœuds sont étiquetés par des entiers peuvent par exemple être représentés en OCaml à l'aide de la déclaration de type suivante :

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

`Nil` et `Node` sont appelés les *constructeurs* du type `int_tree`. Une fois ce type déclaré, on peut construire des valeurs de type `int_tree` :

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

Ces expressions peuvent être représentées graphiquement de la manière suivante :



La première expression, `Nil`, représente l'arbre *vide*, c'est-à-dire l'arbre ne contenant aucun nœud. La seconde représente une *feuille*, c'est-à-dire un arbre non vide, dont les sous-arbres gauche et droit sont vides. La troisième représente un arbre à deux nœuds, dont le sous-arbre droit est vide, et dont le sous-arbre gauche est la feuille précédente.

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       ->      ...       (* cas ou a est l'arbre vide             *)
 | Node(n, g, d) ->  ...       (* cas ou a est un arbre non vide,       *)
                               (* etiquete par un entier n a la racine  *)
                               (* de sous-arbres gauche et droit g et d *)
;;
```

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

1. `taille : int_tree -> int` renvoyant la taille d'un arbre, c'est-à-dire son nombre de noeuds.

2. `hauteur : int_tree -> int` renvoyant la hauteur d'un arbre, c'est-à-dire la longueur de sa plus longue branche. Par exemple : l'arbre vide est de hauteur `0` ; une feuille est de hauteur `1`, etc.
3. `contient : int -> int_tree -> bool`, telle que `contient x a` renvoie `true` si et seulement si l'un des nœuds de l'arbre `a` est étiqueté par `x`.
4. `somme : int_tree -> int`, qui renvoie la somme des étiquettes d'un arbre.
5. `elements : int_tree -> int list`, qui renvoie la liste des éléments présents dans l'arbre, en les collectant de la gauche vers la droite (étiquettes du sous-arbre gauche, étiquette à la racine, étiquettes du sous-arbre droit). Sauriez-vous écrire une version sans utiliser la concatenation (`@`) des listes ?

(⋆) 5. `complet : int_tree -> bool` déterminant si un arbre est complet, c'est-à-dire si toutes ses feuilles sont à la même profondeur et tout noeud a 0 ou 2 fils non vides. Noter que tester la complétude de ses sous-arbres gauche et droit ne suffit pas (pourquoi ?)

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 -> []
```

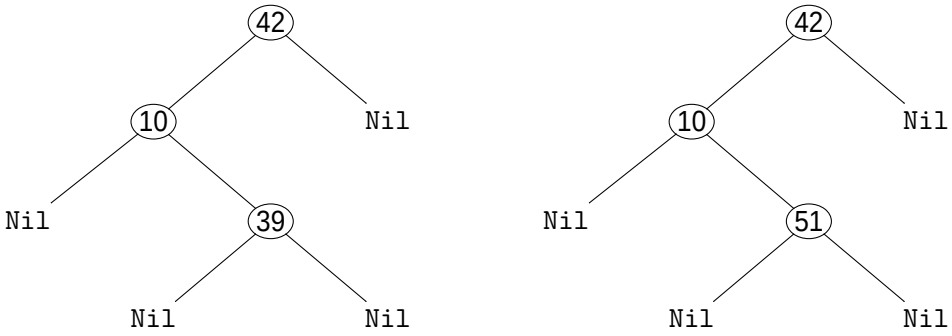
```
  | Node (n, g, d) -> (elements g) @ [n] @ (elements d) ;;

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

2 Arbres binaires de recherche

Un *arbre binaire de recherche* (ABR) est un arbre dont *tout* sous-arbre non vide vérifie la propriété suivante :

Soit `n` l'étiquette de sa racine, soient `g` et `d` ses sous-arbres gauche droit.
— toutes les étiquettes apparaissant dans `g` sont strictement inférieures à `n`.
— toutes les étiquettes apparaissant dans `d` sont strictement supérieures à `n`.

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. `contient_abr : int -> int_tree -> bool` tel que, si `a` est un ABR, `contient x a` renvoie `true` si et seulement si l'un des nœuds de l'arbre `a` est étiqueté par `x`. L'exploration de `a` devra être minimale, en tenant compte du fait qu'il s'agit d'un ABR.
2. `ajout_abr : int -> int_tree -> int_tree` tel que, lorsque `a` est un ABR et `n` est un entier, `ajout_abr` renvoie l'arbre `a` dans lequel on a ajouté `n` s'il n'y est pas déjà – si `x` apparaît déjà dans `a`, le résultat est encore `a`. Le résultat doit être encore un ABR.
3. `cons_abr : int list -> int_tree` construisant à partir d'une liste d'entiers, un ABR contenant les mêmes éléments. Cette fonction peut utiliser la précédente. Vérifier sur des exemples que `elements (cons_abr l)` et `l` contiennent bien les mêmes éléments.

   Quelle forme d'ABR obtient-on lorsque la liste est triée ? Est-ce un problème ?

(⋆) 4. `cons_abr_opt : int list -> int_tree` transformant une liste *triée et sans répétitions* d'entiers en un ABR contenant les mêmes éléments, et de profondeur minimale. Vérifier sur des exemples que `elements (cons_abr_opt l) = l`.

   Pour limiter la profondeur de l'arbre obtenu, on pourra écrire une fonction auxiliaire découpant une liste en deux moitiés.

(⋆⋆) 5. `est_abr : int_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.

Pour tester ces fonctions, vous pouvez utiliser la fonction `print_int_tree : int_tree -> unit` qui affiche des valeurs de types `int_tree`. Cette fonction est disponible sur Moodle, dans le fichier `print_int_tree.ml`, contenant aussi les définitions d'un certain nombre d'arbres.

Correction :

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

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

(* 3 *)
let rec cons_abr = function
 | [] -> Nil
 | x :: l -> ajout_abr x (cons_abr l);;


(* 4 *)

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


(* 5 *)

(* version naive *)
let rec majorant x a = match a with
 | Nil -> true
 | Node (y, g, d) -> y < x && majorant x g && majorant x d ;;

let rec minorant x a = match a with
```

```
 | Nil -> true
 | Node (y, g, d) -> x < y && minorant x g && minorant x d;;

let rec est_abr a = match a with
 | 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 est_abr a = check_abr None None a ;;
```

3 Modélisation à l'aide de types algébriques

Exercice 3. On souhaite à présent représenter des arbres étiquetés à la fois par des `int` et des `float`. Reprendre la déclaration du type `int_tree`, et déclarer un nouveau type `int_float_tree` premettant de représenter ces arbres – il y a deux sortes de nœuds : ceux étiquetés par un `int`, et ceux étiquetés par un `float`, donc il faudra un constructeur pour chaque sorte.
Correction :

```
type int_float_tree =
   Nil
 | Node_int   of int   * int_float_tree * int_float_tree
 | Node_float of float * int_float_tree * int_float_tree
;;
```

Exercice 4. On souhaite représenter des arbres dont chaque nœud est étiqueté par une couleur choisie dans un ensemble fini de couleurs (par exemple, rouge, blanc, noir). Définir un type `color` permettant de représenter ces couleurs (il faut un constructeur par couleur). Donner ensuite la déclaration d'un type `color_tree` utilisant le type précédent, et permettant de représenter ces arbres.
Correction :

```
type color =
   Red
 | White
```

```
 | Black
;;
type color_tree =
   Nil
 | Node of color * color_tree * color_tree
;;
```

Exercice 5. Dans un jeu de 52 cartes usuel, chaque carte est étiqueté par une *valeur* et une *couleur*. Les valeurs possibles sont 1, 2, ...10, Valet, Dame, Roi. Les quatre couleurs sont Pique, Cœur, Carreau et Trèfle. A ces 52 cartes, on ajoute parfois deux cartes spéciales, sans couleur : les jokers.

Proposer un ou plusieurs ensembles de déclarations de types permettant de modéliser un jeu de cartes, avec ou sans jokers (il y a plusieurs solutions).

Correction : type couleur = Pique | Coeur | Carreau | Trefle ; ; type valeur = Point of int | Valet | Dame | Roi ; ; type carte = Joker | Carte of valeur * couleur ; ;

Exercice 6. Dans un jeu de tarot usuel, il y a trois sortes de cartes :

- les *atouts*, étiquetés seulement par un entier compris entre 1 et 21.
- une carte spéciale appelée l'*excuse*,
- toutes les autres cartes, étiquetées à la fois par une valeur (1, ..., 10, Valet, Cavalier, Dame, Roi) et une couleur (Pique, Cœur, Carreau, Trèfle).

Proposer un ou plusieurs ensembles de déclarations de types permettant de modéliser un jeu de tarot.

Correction : type couleur = Pique | Coeur | Carreau | Trefle ; ; type valeur = Point of int | Valet | Cavalier | Dame | Roi ; ; type carte = Atout of int | Excuse | Carte of valeur * couleur ; ;

4 Pour aller plus loin : les types algébriques polymorphes

Une déclaration de type peut contenir des *inconnues* de types, qui ne seront remplacées par des types définis qu'au moment de la construction de valeurs. La déclaration ci-dessous suffit pour représenter, pour chaque type défini – `int`, `string` ...– des arbres étiquetés par des valeurs de ce type :

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

Un tel type est dit *polymorphe*. Une fois le type `'a tree` déclaré, on peut par exemple construire successivement des valeurs de type

- `int tree` :
  ```
  Node(42, Nil, Node(10, Nil, Nil));;
  ```
- `string tree` :
  ```
  Node("abc", Nil, Node("def", Nil, Nil));;
  ```
- `(int list) tree` :
  ```
  Node([2; 3], Nil, Node([], Nil, Nil));;
  ```
- `(int -> int) tree` :
  ```
  Node(fun x -> x + 1, Nil, Node(fun y -> 42, Nil, Nil));;
  ```
- etc.

Exercice 7. Remplacez la déclaration de `int_tree` par celle de `'a tree` au début de votre code. Les déclarations de fonctions des exercices 1 et 2 sont-elles encore valides ? Observez les nouveaux types assignés à ces fonctions.

Exercice 8. La déclaration d'un type polymorphe peut contenir *plusieurs* inconnues de types. On les rassemble dans ce cas avant le nom du type déclaré avec une syntaxe de la forme :

```
type ('a, 'b) mon_type = ....
```

En vous inspirant de la solution de l'exercice 3, donner la déclaration d'un type polymorphe permettant de représenter des arbres dont les étiquettes seront de deux types quelconques.
Correction :

```
type ('a, 'b) tree =
  Nil
  | Node_a of 'a * ('a, 'b) tree * ('a, 'b) tree
  | Node_b of 'b * ('a, 'b) tree * ('a, 'b) tree;;

Node_a (1, Node_b(1.3,Nil,Nil),Nil);;
```