Université Paris Diderot — Programmation Fonctionnelle
L2 Math-Info/L3 Informatique/M1 Linguistique — Année 2013-2014

TP n°4

Arbres binaires

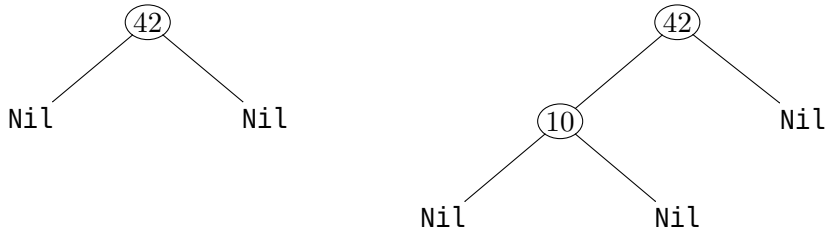
Il est possible d'effectuer en OCaml des *déclarations de types*, c'est-à-dire créer de nouveaux types de valeurs. On peut ainsi représenter en OCaml des arbres binaires à l'aide de la déclaration de type suivante :

```
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. Nous avons choisi ici de placer ces éléments au niveau des nœuds de l'arbre et non pas dans les feuilles. 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 :



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 :

1. `taille : 'a tree -> int` renvoyant la taille d'un arbre, c'est-à-dire son nombre de noeuds internes.
2. `hauteur : 'a tree -> int` renvoyant la hauteur d'un arbre, c'est-à-dire la longueur de sa plus longue branche.

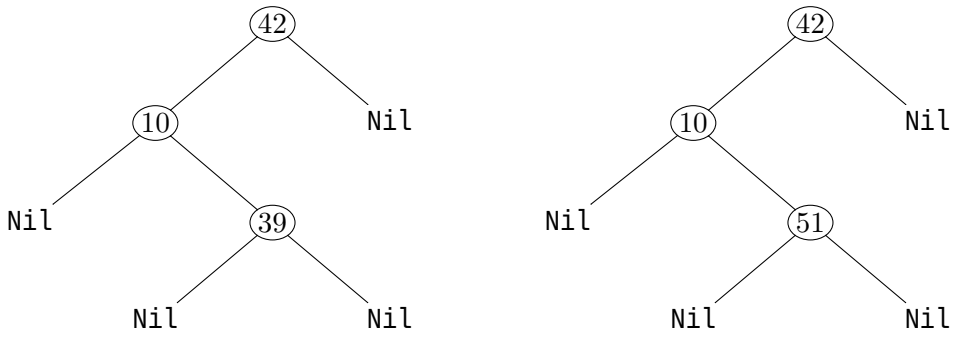
3. `mem : 'a -> 'a tree -> bool`, telle que `mem x a` renvoie `true` si et seulement si l'un des nœuds de l'arbre `a` est étiqueté par `x`.
4. `complet : 'a tree -> bool` déterminant si un arbre est complet, c'est-à-dire si toutes ses feuilles sont à la même profondeur.
5. `elements : 'a tree -> 'a list`, qui renvoie la liste des éléments présents dans l'arbre, dans l'ordre de leurs apparitions de gauche à droite dans l'arbre.

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. ($\star$) `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.