TP de programmation fonctionnelle et logique
Corrigé du TP 2 : arbres n-aires et fonctionnelles

Arbres n-aires

Un arbre n-aire est soit un nœud interne (composé d'un entier, la clef, et d'une liste de sous-arbres), soit une feuille, c'est-à-dire un nœud sans fils. Nous définissons un arbre n-aire par le type suivant :

```
# type arbre = Noeud of (int * arbre list);;
type arbre = Noeud of (int * arbre list)
# let arbre1 = Noeud(1,[]);;
val arbre1 : arbre = Noeud (1, [])

# let arbre2 = Noeud(2,[Noeud(3,[Noeud(4,[])]);Noeud(5,[])]);;
val arbre2 : arbre = Noeud (2, [Noeud (3, [Noeud (4, [])]); Noeud (5, [])])

# let arbre3 = Noeud(2,[Noeud(3,[Noeud(6,[])]);Noeud(5,[])]);;
val arbre3 : arbre = Noeud (2, [Noeud (3, [Noeud (6, [])]); Noeud (5, [])])
```

Fonctions élémentaires sur les arbres

1. Écrire une fonction `make_arbre` qui construit un arbre à partir d'un entier et d'une liste de sous-arbres.

```
# let make_arbre clé fils =
    Noeud (clé, fils);;
val make_arbre : int -> arbre list -> arbre = <fun>
```

2. Écrire des fonctions `clé` et `fils` qui extraient respectivement d'un nœud interne sa clé et la liste de ses fils.

```
# let clé  = function Noeud (c, _) -> c;;
val clé : arbre -> int = <fun>

# let fils = function Noeud (_, f) -> f;;
val fils : arbre -> arbre list = <fun>
```

3. Définissez une fonction `feuille` qui renvoie le booléen `true` si et seulement si son argument est une feuille.

```
# let feuille = function
    Noeud(_,[]) -> true
  |      _      -> false;;
val feuille : arbre -> bool = <fun>
```

Calcul sur les arbres

1. Définissez une fonction qui calcule la hauteur d'un arbre.

```
# let rec hauteur =
   function Noeud (_, f) -> 1 + (List.fold_right
                                    max
                                    (List.map hauteur f)
                                    0);;
```

```
val hauteur : arbre -> int = <fun>
```

```
# let rec hauteur =
   function Noeud (_, f) ->
      1 + (List.fold_right
                (function x -> function y -> max (hauteur x) y)
                f
                0);;
val hauteur : arbre -> int = <fun>
```

```
# hauteur arbre1;;
- : int = 1
```

```
# hauteur arbre2;;
- : int = 3
```

2. Construisez les fonctions `nb_feuilles` et `nb_noeuds` qui calculent respectivement le nombre de feuilles et de nœuds (nœuds internes et feuilles) d'un arbre.

```
# let rec nb_feuilles = function
     Noeud (_,[]) -> 1
   | Noeud (_, f) -> List.fold_right
                          (function x -> function y-> x + y)
                          (List.map nb_feuilles f)
                          0;;
val nb_feuilles : arbre -> int = <fun>
```

```
# let rec nb_feuilles = function
   Noeud (_,[]) -> 1
 | Noeud (_, f) -> List.fold_right
                        (function x -> function y-> (nb_feuilles x) + y)
                        f
                        0;;
val nb_feuilles : arbre -> int = <fun>
```

```
# let rec nb_noeuds = function
     Noeud (_,[]) -> 1
   | Noeud (_, f) -> 1 + (List.fold_right
                               (function x -> function y-> x + y)
                               (List.map nb_noeuds f)
                               0);;
val nb_noeuds : arbre -> int = <fun>
```

```
# let rec nb_noeuds = function
   Noeud (_,[]) -> 1
 | Noeud (_, f) -> 1 + (List.fold_right
                          (function x-> function y-> (nb_noeuds x) + y)
                           f
                           0);;
val nb_noeuds : arbre -> int = <fun>
```

```
# nb_feuilles arbre2;;
- : int = 2
```

```
# nb_noeuds   arbre2;;
- : int = 4
```

Parcours dans les arbres

1. Écrivez une fonction `parcours_profondeur` qui énumère les clefs d'un arbre en profondeur d'abord.

```
# let rec parcours_profondeur = function
    Noeud (c, [])   -> print_int c
  | Noeud (c, fils) -> print_int c;
                       List.iter parcours_profondeur fils;;
val parcours_profondeur : arbre -> unit = <fun>
```

```
# let rec parcours_profondeur =
   function Noeud (c, fils) -> print_int c;
                               List.iter parcours_profondeur fils;;
val parcours_profondeur : arbre -> unit = <fun>
```

```
# parcours_profondeur arbre2;;
2345- : unit = ()
```

2. Écrivez une fonction `appartient` qui renvoie `true` si et seulement si l'entier argument est contenu dans l'arbre étudié.

```
# let rec appartient n= function
    Noeud (c, [])   -> c = n
  | Noeud (c, fils) -> c = n or
                      (List.fold_right
                          (function x -> function y-> x or y)
                          (List.map (appartient n) fils)
                          false);;
val appartient : int -> arbre -> bool = <fun>
```

```
# let rec appartient n= function Noeud (c, fils) ->
      c = n or
      (List.fold_right
            (function x -> function y-> (appartient n x) or y)
            fils
            false);;
val appartient : int -> arbre -> bool = <fun>
```

```
# appartient 1 arbre2;;
- : bool = false
```

```
# appartient 5 arbre2;;
- : bool = true
```

3. Écrivez une fonction `arbres_égaux` qui teste l'égalité de deux arbres.

```
# let rec combine a b = match (a,b) with
     [] ,  []  -> []
  | t::r, u::s -> (t,u)::combine r s
  |   _ ,   _  -> failwith "Listes non combinables";;
val combine : 'a list -> 'b list -> ('a * 'b) list = <fun>
```

```
# let rec arbres_égaux a b = match (a,b) with
    Noeud(c,[]),Noeud(d,[]) -> c = d
  | Noeud(c,[]),Noeud(d,_)  -> false
  | Noeud(c,_), Noeud(d,[]) -> false
  | Noeud(c,e), Noeud(d,f)  ->
      List.fold_right
           (function (g,h) -> function y -> (arbres_égaux g h) & y)
           (combine e f)
           true;;
val arbres_égaux : arbre -> arbre -> bool = <fun>
```

```
# arbres_égaux arbre1 arbre1;;
```

```
- : bool = true

# arbres_égaux arbre2 arbre2;;
- : bool = true

# arbres_égaux arbre2 arbre3;;
- : bool = false

# arbres_égaux arbre2 arbre1;;
- : bool = false
```

4. Écrivez une fonction `parcours_largeur` qui énumère les clefs d'un arbre en largeur d'abord.

```
# let rec parcours_largeur a=
   let rec parcours_liste = function
      (Noeud (c, fils))::l   -> print_int c; parcours_liste (l@fils)
    |           _            -> ()
   in
     parcours_liste [a];;
val parcours_largeur : arbre -> unit = <fun>
```