Programmation fonctionnelle et logique
TD6 et TP3 : itérateurs et exceptions

Itérateurs sur arbres n-aires

```
type arbre = Feuille of int | Noeud of (int * arbre list);;
```

Rappel sur les itérateurs

`List.iter : ('a -> unit) -> 'a list -> unit`.
: `List.iter f [a1 ; ... ; an]` est équivalent à `begin f a1 ; ... ; f an ; () end`.

`List.map : ('a -> 'b) -> 'a list -> 'b list`.
: `List.map f [a1 ; ... ; an]` est équivalent à `[f a1 ; ... ; f an]`.

`List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`.
: `List.fold_left f a [b1 ; ... ; bn]` est équivalent à `f ( ... (f (f a b1) b2) ... ) bn`.

`List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` .
: `List.fold_right f [a1 ; ... ; an] b` est équivalent à `f a1 (f a2 (... (f an b) ...))`.

Fonctions élémentaires sur les arbres

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

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

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

```
let clé = function
         Feuille n     -> n
 | Noeud (c, f) -> c;;
val clé : arbre -> int = <fun>
let fils = function
    Noeud (c, f) -> f
  | _  -> failwith "fils";;
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
   Feuille _    -> true
 | 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
       Feuille _   -> 1
 | Noeud (_, f) -> 1 + (List.fold_right max
                                        (List.map hauteur f)
                                        (hauteur (List.hd f)));;
val hauteur : arbre -> int = <fun>
```

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
       Feuille _   -> 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_noeuds = function
       Feuille _   -> 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>
```

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
       Feuille f   -> print_int f
 | Noeud (c, fils) -> print_int c ;
                      List.iter parcours_profondeur fils;;
val parcours_profondeur : arbre -> unit = <fun>
```

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
       Feuille f   -> f = 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>
```

3. 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
      (Feuille f)::l         -> print_int f; parcours_liste l
    | (Noeud (c, fils))::l   -> print_int c; parcours_liste (l@fils)
    |          _             -> ()
  in
    parcours_liste [a];;
val parcours_largeur : arbre -> unit = <fun>
```

Exceptions

On se place de nouveau dans le cadre des arbres binaires définis par le type suivant :

```
type arbre = Feuille of int | Noeud of (int * arbre  * arbre);;
```

- Écrivez une fonction `appartient` qui renvoie `true` si et seulement si l'entier argument est contenu dans l'arbre étudié ; et qui se termine dès que l'élément recherché a été trouvé.

```
exception Trouvé;;
let appartient n tree=
   let rec appartient_aux = function
             Feuille f   -> if (f = n) then raise Trouvé else false
     | Noeud (c, fg, fd) -> if c = n
                              then  raise Trouvé
                              else
                                 (appartient_aux fg) or
                                 (appartient_aux fd)
   in
   try appartient_aux tree with Trouvé -> true;;
val appartient : int -> arbre -> bool = <fun>
```

- Écrivez une fonction `appartient_prof` ayant le même comportement que la fonction `appartient`, mais renvoyant en plus la profondeur du nœud contenant l'élément recherché (on renverra 0 si l'élément recherché est absent de l'arbre).

```
exception Trouvé_prof of int;;
let appartient_prof n tree=
   let rec appartient_prof_aux prof = function
             Feuille f   -> if (f = n) then raise (Trouvé_prof prof) else false
     | Noeud (c, fg, fd) -> if c = n
                              then  raise (Trouvé_prof prof)
                              else
                                 (appartient_prof_aux (prof+1) fg) or
                                 (appartient_prof_aux (prof+1) fd)
   in
   try (appartient_prof_aux 1 tree,0) with Trouvé_prof n -> (true,n);;
val appartient_prof : int -> arbre -> bool * int = <fun>
```

- Écrivez une fonction `appartient_arbre` ayant le même comportement que la fonction `appartient`, mais renvoyant en plus le sous-arbre dont la racine est l'élément recherché (on renverra l'arbre de départ si l'élément recherché en est absent).

```
exception Trouvé_arbre of arbre;;
let appartient_arbre n tree=
   let rec appartient_arbre_aux = function
             (Feuille f) as t   -> if (f = n) then raise (Trouvé_arbre t) else false
     | (Noeud (c, fg, fd)) as t -> if c = n
                              then  raise (Trouvé_arbre t)
                              else
                                 (appartient_arbre_aux fg) or
                                 (appartient_arbre_aux fd)
   in
   try (appartient_arbre_aux tree, tree) with Trouvé_arbre t -> (true, t);;
val appartient_arbre : int -> arbre -> bool * arbre = <fun>
```

- Écrivez une fonction `appartient_arbre_père` qui recherche un élément dans un arbre et qui renvoie le sous-arbre dont un des fils a l'élément recherché pour racine (si celui existe).

```
exception Trouvé_arbre;;
exception Trouvé_père of arbre;;
```

```
let appartient_arbre_père n tree=
   let rec appartient_arbre_père_aux = function
               Feuille f        -> if (f = n) then raise Trouvé_arbre else false
     | (Noeud (c, fg, fd)) as t -> if c = n
                               then  raise Trouvé_arbre
                               else
                                 try
                                    (appartient_arbre_père_aux fg) or
                                    (appartient_arbre_père_aux fd)
                                 with
                                   Trouvé_arbre -> raise (Trouvé_père t)
   in
   try
        (appartient_arbre_père_aux tree); failwith "Élément absent de l'arbre"
   with    Trouvé_père t -> t
         | Trouvé_arbre  -> failwith "Élément à la racine";;
val appartient_arbre_père : int -> arbre -> arbre = <fun>
```