Programmation fonctionnelle et logique
TD5 et TP2 : Arbres

Arbres binaires

Un arbre binaire est soit une feuille (un entier), soit un nœud interne (composé d'un entier, la clef, et de deux sous-arbres).

Nous définissons un arbre binaire est défini par le type suivant :

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

Fonctions élémentaires sur les arbres

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

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

2. Écrire des fonctions `clé`, `fg` et `fd` qui extraient respectivement d'un nœud interne sa clé, son fils gauche et son fils droit.

   ```
   let clé = function
            Feuille n     -> n
    | Noeud (c, f_g, f_d) -> c;;
   val clé : arbre -> int = <fun>
   let fg = function
       Noeud (c, f_g, f_d) -> f_g
     | _  -> failwith "fg";;
   val fg : arbre -> arbre = <fun>
   let fd = function
       Noeud (c, f_g, f_d) -> f_d
     | _  -> failwith "fd";;
   val fg : arbre -> arbre = <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
    | _ -> 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 (_, fg, fd) -> 1 +  max (hauteur fg) (hauteur fd);;
    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 (_, fg, fd) -> (nb_feuilles fg) + (nb_feuilles fd);;
val nb_feuilles : arbre -> int = <fun>
let rec nb_noeuds = function
       Feuille _   -> 1
 | Noeud (_, fg, fd) -> 1 + (nb_noeuds fg) + (nb_noeuds fd);;
val nb_feuilles : 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, fg, fd) -> print_int c ;
                        (parcours_profondeur fg);
                        (parcours_profondeur fd);;
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, fg, fd) -> c = n or
                           (appartient n fg) or
                           (appartient n fd);;
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, fg, fd))::l -> print_int c; parcours_liste (l@[fg]@[fd])
    |           _            -> ()
  in
    parcours_liste [a];;
val parcours_largeur : arbre -> unit = <fun>
```

Arbres n-aires

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

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
 | _ -> 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>
```