Arbres génériques

OPTION INFORMATIQUE - TP nº 1.4 - Olivier Reynet

On se dote d'un type pour représenter les arbres génériques :

```
type 'a gtree =
| Empty
| Node of 'a * 'a gtree list;;
```

C'est ainsi que l'on peut représenter en machine l'arbre générique exposé dans le cours :

```
let gt = Node('A', [Node('B', [Node('C', []); Node('D', []); Node('E',[])]);
                Node('F', [Node('G', []); Node('H',[])]);
                Node('I', [Node('J', [Node('K', []); Node('L', []); Node('M',[]);
                    Node('N',[])])]);
                Node('O', [])]);;
```

---

1. Écrire une fonction de signature `hauteur : 'a gtree -> int` qui renvoie la hauteur d'un arbre générique. On pourra proposer deux solutions : une mutuellement récursive et une utilisant `List.filst_left` et `max`

Solution :

```
let rec hauteur a =
    match a with
        | Empty -> -1
        | Node (_, []) -> 0
        | Node (_, fils) -> 1 + List.fold_left (fun acc f -> max acc (hauteur
            f)) 0 fils;;

let rec h a =
    match a with
        | Empty -> -1
        | Node (_, []) -> 0
        | Node (_, fils) -> 1 + h_fils fils
    and h_fils fils =
        match fils with
            | [] -> 0
            | f::t -> max (h f) (h_fils t);;
```

---

2. Même question pour la taille d'un arbre générique.

Solution :

```
let rec taille a =
    match a with
        | Empty -> 0
```

```
        | Node (_, []) -> 1
        | Node (_, fils) -> 1 + List.fold_left (fun acc f -> acc +  (taille f
            )) 0 fils;;

let rec size a =
    match a with
        | Empty -> 0
        | Node (_, []) -> 1
        | Node (_, fils) -> 1 + s_fils fils
    and s_fils fils =
        match fils with
            | [] -> 0
            | f::q ->  size f  +  s_fils q;;
```

3. Écrire une fonction de signature `bfs : 'a gtree -> 'a list` qui renvoie le parcours par niveaux d'un arbre générique sous la forme d'une liste. On s'appuiera sur le module `Queue` d'OCaml (cf. documentation en ligne) pour implémenter une file d'attente.

Solution :

```
let bfs a =
    let file = Queue.create () and parcours = ref [] in
    let rec ajouter_fils fl =
        match fl with
            | [] -> ()
            | h::t -> Queue.add h file; ajouter_fils t in
    Queue.add a file;
    while Queue.length file > 0 do
        let tt = Queue.take file in
        match tt with
            | Empty -> ()
            | Node(e, fils) -> parcours := e :: !parcours; ajouter_fils fils;
    done;
    List.rev !parcours;;
```

4. Écrire une fonction de signature `dfs : 'a gtree -> 'a list` génère le parcours de l'arbre générique en profondeur dans l'ordre préfixe sous la forme d'une liste.

Solution :

```
let rec dfs a =
    match a with
        | Empty -> []
        | Node(e, fils) -> List.fold_left (fun acc e ->  acc @ dfs e) [e]
            fils;;
```

On se dote d'un type pour représenter les arbres binaires :

```
type 'a btree =
            | BEmpty
```

```
                | BNode of 'a * 'a btree * 'a btree;;
```

5. Écrire une fonction de signature `convert_to_btree : 'a gtree -> 'a btree` qui convertit un arbre générique en arbre binaire. On choisit la convention de coder le fils d'un nœud à gauche et ses frères à droite.

Solution :

```
let rec convert_to_btree a =
    match a with
        | Empty -> BEmpty
        | Node (e, []) -> BNode (e, BEmpty, BEmpty)
        | Node (e, f :: t) ->
            let left = convert_to_btree f
            and right = convert_forest t in
            BNode (e, left, right)
    and convert_forest lst =
        match lst with
            | [] -> BEmpty
            | Empty::t ->  convert_forest t
            | Node(b, rb) :: t ->  BNode (b, convert_forest rb,
                convert_forest t);;
```