Dans tout ce TP le type `'a gtree` fait référence au type OCaml ci-dessous défini. Attention, ce n'est pas exactement celui du cours.

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

On remarque qu'à la différence du type des arbres généraux proposés en cours, ce type permet la représentation d'un arbre vide.

Exercice 1 : Manipulations simples

Q. 1 Définir la fonction `size : 'a gtree -> int` qui calcule la taille (nombre de nœuds) de son argument. Vous utiliserez d'abord une récurrence mutuelle, puis l'itérateur `List.fold_left`.

Q. 2 Définir la fonction `height : 'a gtree -> int` qui calcule la hauteur de son argument.

Q. 3 Définir une fonction `to_list : 'a gtree -> 'a list` qui donne la liste des étiquettes de son argument. Peu importe l'ordre dans lequel les étiquettes sont rangées dans la liste ; ce qui importe est qu'elles y soient toutes.

Feuilles d'un arbre général. Sur les arbres binaires, on n'avait qu'une seule notion de *feuille* : les arbres de la forme `Node(x, Empty, Empty)`. Avec les arbres généraux, la chose est plus complexe : les arbres de la forme `Node(x, []`, ou `Node(x, [Empty])`, ou `Node(x, [Empty; Empty])`, etc. peuvent être vus comme des *feuilles* car il n'a y aucune autre étiquette que `x` dans de tels arbres. On appelle *forêt vide* les listes d'arbres de la forme `[]`, `[Empty]`, `[Empty; Empty]`, etc. On appelle donc `feuille` un arbre non vide, de la forme `Node(x, gts)` où `gts` est une *forêt vide*.

Q. 4 Définir la fonction `empty_forest : ('a gtree) list -> bool` qui donne `true` si et seulement si son argument est une forêt vide. Si ce n'est pas déjà fait, proposer une version de cette fonction utilisant l'itérateur `List.for_all`.

Q. 5 Définir la fonction `nb_leaves : 'a gtree -> int` qui calcule le nombre de feuilles de son argument.

Q. 6 Définir la fonction `leaves : 'a gtree -> 'a list` qui donne la liste de toutes les étiquettes des feuilles de son argument.

Exercice 2 : Normalisation des arbres généraux non vides

Q. 1 Définir la fonction `remove_Empty : 'a gtree -> 'a gtree` qui donne l'arbre obtenu en supprimant les (sous)arbres vides inutiles. On suppose que l'arbre donné «au départ» est non vide et `(remove_Empty Empty)` déclenche l'exception `(Invalid_argument "remove_Empty")`.

Q. 2 Pour tester la fonction précédente, définissez la fonction `not_exists_Empty: 'a gtree -> bool` qui vaut `true` si et seulement si son argument ne contient aucun constructeur `Empty`. Vous *pouvez* utiliser l'itérateur `List.for_all`.

Q. 3 Définir une fonction `to_gtree_cours : 'a gtree -> 'a gtree_cours` permettant la transformation d'un arbre général non vide en un arbre général de type `'a gtree_cours` qui correspond au type des arbres généraux du cours. La définition en est rappelée ci-dessous.

```
type 'a gtree_cours =
  | Node of ('a * ('a gtree_cours) list)
```

Exercice 3 : Recherche dans un arbre général

Q. 1 Définir la fonction `find : 'a -> 'a gtree -> 'a gtree` telle que `(find x0 gt)` donne un sous-arbre de `gt` dont l'étiquette est égale à `x0`. L'application `(find x0 gt)` déclenche l'exception `Not_found` si `x0` n'apparaît pas dans `gt`. Si plusieurs sous-arbres de `gt` portent l'étiquette `x0`, on prend la première qui se présente.

Vous utiliserez la construction `try-with` pour rechercher dans une forêt.

Exercice 4 : Transformation en arbre binaire

Q. 1 Implanter en OCaml l'algorithme du cours permettant la traduction d'un arbre général en un arbre binaire.

Q. 2 Implanter en OCaml l'algorithme du cours permettant la traduction d'un arbre binaire en un arbre général.

Exercice 5 : Dictionnaires

L'objectif de cet exercice est de proposer une implantation efficace d'un dictionnaire. Ce dictionnaire doit permettre de faire les opérations d'insertion et de recherche de mots de manière efficace. Pour ce faire nous allons utiliser des arbres généraux. Chercher un mot dans un dictionnaire reviendra alors à parcourir l'arbre en suivant les lettres de notre mot. Par exemple l'arbre représenté en Figure 1 code le dictionnaire contenant les mots : {`"a"`,`"af"`,`"ggy"`,`"ggz"`,`"hu"`,`"ica"`,`"iy"`,`"m"`}. Les mots acceptés sont donc ceux pour lesquels il existe un chemin (étiqueté par les lettres du mot) de la racine de l'arbre à un nœud étiqueté OK. Comme le montre l'exemple, les fils d'un nœud seront rangés par ordre alphabétique sur l'étiquette de l'arête menant à ce nœud.

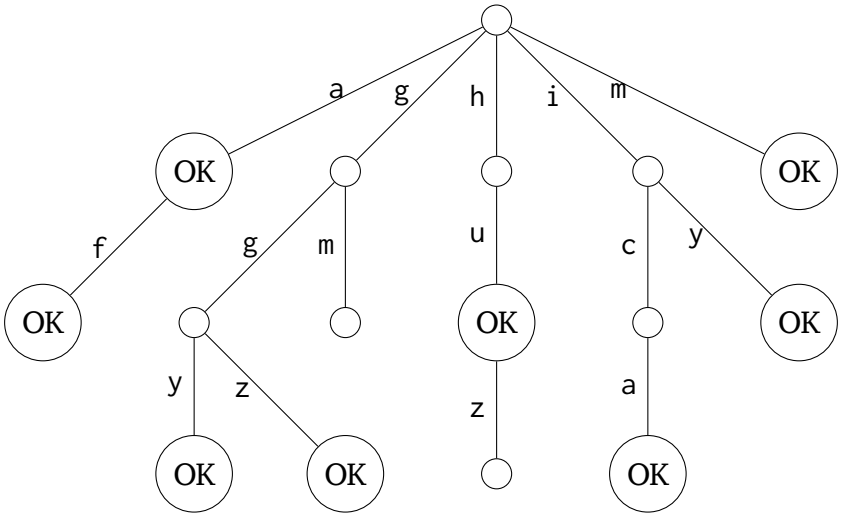

FIGURE 1 – Un exemple de dictionnaire

Q. 1 Définir le type `('a, 'b) gtree` des arbres généraux dont les arêtes sont étiquetées par des éléments de type `'b` et les nœuds par des éléments de type `'a`. Pour représenter l'ensemble des fils d'un nœud et l'étiquette de l'arête allant d'un nœud à son fils on utilisera une liste d'association. On remarquera ici qu'une feuille n'est rien d'autre qu'un nœud n'ayant aucun fils.

Q. 2 Notre implantation de dictionnaire nécessite que les chemins soient étiquetés par des caractères et que chaque nœud indique si oui ou non il est acceptant (le token OK dans l'arbre de la Figure 1). Définir le type `dict` des dictionnaires en fonction du type `('a, 'b) gtree`.

Q. 3 Définir le dictionnaire vide `empty_dict`.

Q. 4 Donner la fonction `is_leaf : ('a,'b) gtree -> bool` permettant de tester si un arbre est réduit à une feuille.

Exemples :

```
is_leaf Node(false,[]) = true
is_leaf Node(false,[('a', Node(true,[]))]) = false
```

Q. 5 On rappelle que la hauteur d'un arbre est la plus grande longueur de chemin entre la racine et une feuille de l'arbre. Définir la fonction `hauteur` calculant la hauteur d'un arbre général.

Q. 6 On rappelle que la taille d'un arbre est le nombre de nœuds se trouvant dans l'arbre. Définir la fonction `taille` calculant la taille d'un arbre général.

Q. 7 Donner la fonction `assoc_ord : char -> (char * 'a) list -> 'a option` telle que `assoc_ord 'a' l` calcule

- `Some v` si `('a',v)` apparaît dans `l`, en cas d'ambiguïté on renverra un `v` quelconque.
- `None` sinon.

On suppose de plus que la liste `l` passée en argument est ordonnée sur le premier élément du couple (le caractère).

Exemples :

```
assoc_ord 'c' [('a', 1); ('c', 2); ('m', 0)] = Some 2
assoc_ord 'c' [('a', 1); ('m', 0)] = None
```

Q. 8 Donner la fonction `find_subtree : char -> dict -> dict option` telle que `find_subtree c d = Some d'` si `d'` est le dictionnaire obtenu après lecture du caractère `c` dans le dictionnaire `d` et `find_subtree c d = None` s'il n'existe pas de tel dictionnaire.

Exemples :

```
find_subtree : 'c' (Node(true,[('a', Node(false, [])); ('c', Node(true,
[]))])) = Some (Node(true, []))
find_subtree : 'c' (Node(true,[('a', Node(false, [])); ('d', Node(true,
[]))])) = None
```

Q. 9 Étant donné le type `word = char list`, donner la fonction `mem : dict -> word -> bool` qui teste si un mot est dans un dictionnaire.

Exemples :

si `d` est l'arbre de la Figure 1

```
mem d ['g';'g';'y'] = true
mem d ['g';'g';'u'] = false
mem d ['g';'m'] = false
```

Q. 10 ⋆ Définir deux fonctions `add_word : dict -> word -> dict` et `add_word_to_dict_list : (char * dict) list -> word -> (char * dict) list` mutuellement récursives et ajoutant un mot à un dictionnaire (pour `add_word`) et à une liste de couple caractère × dictionnaire.

Les trois questions suivantes proposent une définition alternative de la fonction `add_word`.

Q. 11 Définir une fonction `list_split : 'a -> ('a * 'b) list -> (('a * b') list * 'b option * ('a * 'b) list)` telle que `list_split a l` coupe la liste `l` en 3 parties : les paires dont le premier élément est plus petit que `a`, l'élément (s'il existe) associé à `a` dans `l` et enfin les paires dont le premier élément est plus grand que `a`. On inversera l'ordre des éléments plus petits que `a`.

Exemples :

```
list_split 2 [(0,true); (1, true); (2, false); (3, true); (4, false)] =
([(1, true); (0, true)], Some false, [(3, true); (4, false)])

list_split 2 [(0,true); (1, true); (3, true); (4, false)] = ([(1, true);
(0, true)], None, [(3, true); (4, false)])
```

Q. 12 Définir la fonction `list_glue : 'c list -> 'c list -> 'c list` telle que : `list_glue [l1;...;ln] [h1;...;hm] = [ln;...;l1;h1;...hm]`.

Exemples :

```
list_glue [3;2;1] [4;5;6] = [1;2;3;4;5;6]
```

Q. 13 Définir la fonction `add_word : dict -> word -> dict` prenant en arguments un dictionnaire et un mot et qui calcule le dictionnaire augmenté du mot passé en argument.

Q. 14 Définir la fonction `from_word_list : word list -> dict` calculant un dictionnaire contenant exactement les mots de la liste qui lui est passée en argument.