TP 5 : Arbres binaires et arbres généraux en OCaml

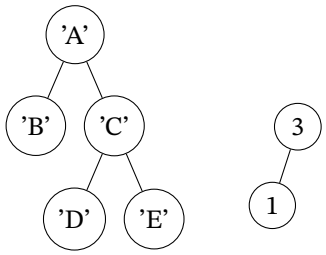
On commence par manipuler simplement des arbres en OCaml. On va commencer par les arbres binaires non stricts, puis on pourra passer aux arbres binaires stricts. Ici, les ★ veulent simplement dire que c'est plus difficile / qu'on n'a pas encore forcément touché au cours nécessaire pour répondre.

Exercice 1 – Arbres binaires non stricts en OCaml

On va utiliser le type suivant :

```OCaml
type 'a arbre =
  | Vide
  | Noeud of 'a * 'a arbre * 'a arbre
```

▶ **Question 1** Représenter les arbres suivants avec ce type :



- un arbre vide,
- un arbre de hauteur 3 à 4 nœuds,
- un arbre de hauteur 3 à 15 nœuds (réfléchir une seconde avant de faire un exemple à 15 lignes),
- ★ un arbre de hauteur 11 à 4095 nœuds (réfléchir une seconde avant de faire un exemple à 4095 lignes),

▶ **Question 2** Tester vos fonctions `hauteur` `nombre_noeuds` `liste_sous_arbres` sur ces exemples et vérifier que le résultat correspond à vos attentes.

▶ **Question 3** Écrire une fonction `est_feuille : 'a arbre -> bool` vérifiant si un nœud est une feuille. *On s'interdira d'utiliser cette fonction dans la suite, mais on notera et on chérira la beauté du motif que l'on y utilise.*

▶ **Question 4** Écrire une fonction `nb_feuilles : 'a arbre -> int` qui renvoie le nombre de feuilles d'un arbre.

▶ **Question 5** Écrire une fonction `somme_etiquettes : int arbre -> int` qui renvoie la somme des étiquettes des nœuds de l'arbre.

▶ **Question 6** Écrire une fonction `max_etiquettes : 'a arbre -> 'a` qui renvoie le maximum des étiquettes des nœuds de l'arbre. On lèvera une exception si l'arbre est vide.

▶ **Question 7** Écrire une fonction `somme_feuilles : int arbre -> int` qui renvoie la somme des étiquettes de toutes les feuilles.

★ **Question 8** Écrire une fonction `max_min_arbre : int arbre -> int * int` qui renvoie le minimum et le maximum des étiquettes d'un arbre. Chaque nœud ne devra être parcouru qu'une seule fois. On lèvera une exception si l'arbre est vide. Attention : il ne doit y avoir, lors d'un appel de la fonction, qu'au plus un appel récursif sur chaque fils.

★ **Question 9** Une branche est un chemin de la racine vers une feuille. Le poids d'une branche est la somme des étiquettes des nœuds qui la composent. Écrire une fonction `max_somme_branche : int arbre -> int` qui renvoie le poids de la branche de poids maximal.

On considère maintenant le type :

```OCaml
type ('a, 'b) arbre_strict =
  | Feuille of 'a
  | Noeud_interne of 'b * 'a arbre_strict * 'a arbre_strict
(* le type ('a, 'b) arbre_strict est un arbre binaire strict
 ↪ dont les feuilles
   contiennent des étiquettes de type 'a et les nœuds
    ↪ internes des étiquettes de type 'b *)
```

★ **Question 10** Certains des arbres définis à la section précédente ne peuvent plus l'être avec ce nouveau type. Lesquels ? Pourquoi ?

★ **Question 11** Reprendre toutes les questions de l'exercice avec ce nouveau type.

Exercice 2 – Arbres généraux et forêts

Dans cet exercice, on s'intéresse aux nœuds d'arité quelconque, avec un point de vue légèrement différent du cours. Pour cela, on introduit la notion de *forêt* : une forêt est une liste d'arbres. On peut donc voir les fils d'un nœud d'un arbre général

comme une forêt. En OCaml on peut aussi définir les arbres généraux ainsi :

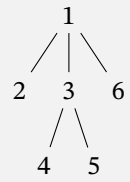
```ocaml
type 'a arbre_general =
  | Vide
  | Noeud of 'a * 'a arbre_general list
```

On peut vouloir donner un nom plus explicite à une liste d'arbres qui est une forêt ! Mais alors pour définir une forêt, il faut savoir définir un arbre et réciproquement. De la même manière que l'on peut définir des fonctions mutuellement récursives, on peut implémenter cette définition circulaire en utilisant des *types mutuellement récursifs* :

```ocaml
type 'a arbre_general = Vide | Noeud of 'a * 'a foret
  and 'a foret = 'a arbre_general list
```

Exemple 1

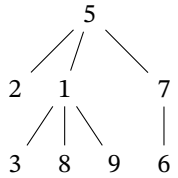
Considérons l'arbre suivant :



Cet arbre est représenté dans le type `int arbre_general` par :

```ocaml
Noeud (1,
  [ Noeud (2, []) ;
    Noeud (3,
      [ Noeud (4, []) ;
        Noeud (5, [])
      ]) ;
    Noeud (6, []) ])
```

▶ **Question 1** Implémenter cet arbre en OCaml.



Les notions de taille et profondeur existent toujours sur les arbres généraux. On peut les programmer en OCaml à l'aide de deux fonctions mutuellement récursives :

```ocaml
let rec taille arbre =
  match arbre with
  | Vide -> 0
  | Noeud (_, fils) -> 1 + taille_foret fils
and taille_foret foret =
  match foret with
  | [] -> 0
  | arbre :: freres -> (taille arbre) + (taille_foret freres)
```

▶ **Question 2** Écrire la fonction `hauteur : 'a arbre_general -> int`. On pourra définir la fonction mutuellement récursive `max_hauteur_foret : 'a arbre_general list -> int`.

▶ **Question 3** Écrire une fonction `degre : 'a arbre_general -> int` qui donne le degré d'un arbre, c'est-à-dire le plus grand degré de ses nœuds.

Si on suppose que l'on n'a pas besoin de l'arbre vide (ce qui est assez naturel dans certaines études) on peut se passer du constructeur de type. Pour bien insister sur le fait que l'on manipule un arbre et non un couple quelconque, on propose le type suivant :

```ocaml
type 'a arbre_general = {etiquette : 'a; fils : 'a
 ↪  arbre_general list}
```

On aurait pu aussi utiliser un type somme avec un unique constructeur paramétré, comme on l'a mentionné dans le cours.

★ **Question 4** Réécrire les fonctions `taille` et `hauteur` avec ce nouveau type.

★ **Question 5** Écrire une fonction `nb_feuilles : 'a arbre_general -> int` qui renvoie le nombre de feuilles d'un arbre.

★ **Question 6** Écrire une fonction `max_etiquettes : 'a arbre_general -> 'a` qui renvoie le maximum des étiquettes des nœuds de l'arbre.

★ **Question 7** Adapter la fonction `max_somme_branche : int arbre_general -> int` qui renvoie le poids de la branche de poids maximal. *Une branche est un chemin allant de la racine jusqu'à une feuille.*

Exercice 3 – Liste des étiquettes

Dans cet exercice, on utilise le type d'arbre binaire strict suivant :

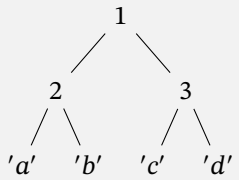
```OCaml
type ('a, 'b) arbre_strict =
  | Feuille of 'a
  | Noeud of 'b * ('a, 'b) arbre_strict * ('a, 'b)
   ↪  arbre_strict
```

L'objectif maintenant est d'implémenter une fonction qui, à un arbre binaire de type `('a, 'b) arbre_strict` donné, associe la liste des étiquettes des nœuds de l'arbre dans l'ordre des parcours vus en cours. Puisqu'on ne peut pas définir de listes contenant des valeurs de types différents, on est obligé de définir un autre type permettant de les « unir » de la façon suivante :

```OCaml
type ('a, 'b) token = F of 'a | N of 'b
```

Exemple 2

On considère l'arbre suivant :



La liste des étiquettes des nœuds de cet arbre dans l'ordre préfixe est alors `[N 1; N 2; F 'a'; F 'b'; N 3; F 'c'; F 'd']`, cette liste sera donc de type `(char, int) token list`.

Ainsi, les fonctions de cet exercice auront le type `('a, 'b) arbre_strict -> ('a, 'b) token list`.

▶ **Question 1** Écrire de la façon la plus simple possible une fonction `postfixe_naif` renvoyant la liste des étiquettes des nœuds d'un arbre dans l'ordre postfixe. *On utilisera l'opérateur de concaténation* `@`.

```OCaml
# postfixe_naif exemple_strict;;
- : (char, int) token list = [F 'a'; F 'b'; N 2; F 'c'; F
 ↪  'd'; N 3; N 1]
```

L'utilisation d'un opérateur de concaténation est coûteuse en complexité, on peut montrer que la complexité de l'appel `postfixe_naif a` est en $\mathcal{O}\left(n(a)^2\right)$.

★ **Question 2** Écrire une fonction `postfixe` renvoyant la liste des étiquettes des nœuds d'un arbre dans l'ordre postfixe en temps linéaire en la taille de l'arbre. *On utilisera une fonction auxiliaire* `postfixe_aux` *prenant en argument une liste accumulateur et un arbre et renvoyant la liste des étiquettes des nœuds de l'arbre concaténée à la liste accumulateur. On impose une complexité linéaire en la taille de l'arbre.*

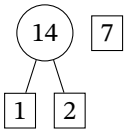
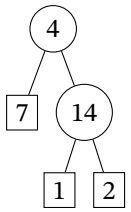
★ **Question 3** De même, écrire des fonctions `prefixe`, `infixe` efficaces.

Exercice 4 – Reconstruction d'un arbre à partir de son parcours

L'objectif de cet exercice est d'écrire des fonctions de type `('a, 'b) token list -> ('a, 'b) arbre_strict` permettant de reconstruire un arbre à partir de son parcours.

▶ **Question 1** Donner deux arbres dont le parcours infixe produit la liste `[F 0; N 1; F 2; N 3; F 4]`. Que peut-on en déduire ?

Pour reconstruire un arbre à partir de son parcours postfixe, on propose l'algorithme suivant :

Étiquettes lues (tête de la liste à gauche)	Forêt (tête de la liste à gauche)
`F 7; F 1; F 2; N 14; N 4; F 20; N 12`	
`F 1; F 2; N 14; N 4; F 20; N 12`	
`F 2; N 14; N 4; F 20; N 12`	
`N 14; N 4; F 20; N 12`	
`N 4; F 20; N 12`	
`F 20; N 12`	

▶ **Question 2** Terminer l'exécution de cet algorithme.

▶ **Question 3** Écrire une fonction `reconstruit_postfixe : ('a, 'b) token list -> ('a, 'b) arbre_strict` reconstruisant un arbre à partir de son parcours postfixe. *On pourra utiliser une fonction auxiliaire* `aux : ('a, 'b) arbre_strict list -> ('a, 'b) token list -> ('a, 'b) arbre_strict`, *la liste d'arbres jouant le rôle de pile fonctionnelle.*

▶ **Question 4** De même, écrire une fonction `reconstruit_prefixe : ('a, 'b) token list -> ('a, 'b) arbre_strict` reconstruisant un arbre à partir de son parcours préfixe.