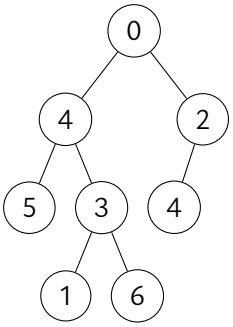
TP 15 : Parcours d'arbres

En cours, nous avons vu 4 façons différentes de *parcourir* un arbre, c'est-à-dire de visiter tous ses nœuds pour y effectuer une action. Nous allons mettre en oeuvre les trois parcours en profondeur dans ce TP, en laissant le parcours en largeur de côté pour l'instant.

Exercice 1 – Parcours en profondeur inefficaces

On donne l'exemple d'arbre suivant :



et le type suivant :

```OCaml
type 'a arbre =
  | V
  | N of 'a * 'a arbre * 'a arbre
```

▶ **Question 1** Définir une variable `ex : int arbre` implémentant l'arbre ci-dessus.

▶ **Question 2** Écrire une fonction récursive `prefixe_inefficace : 'a arbre -> 'a list` renvoyant la liste des étiquettes de son entrée dans l'ordre du parcours préfixe. On s'autorise à utiliser l'opérateur `@`.

▶ **Question 3** De même, écrire les fonctions `infixe_inefficace` et `postfixe_inefficace`. *On rappelle que l'on dispose des raccourcis `Ctrl+C` et `Ctrl+V` pour copier-coller du texte dans un éditeur, et que les utiliser n'est pas forcément une mauvaise idée.*

▶ **Question 4** Pourquoi appelle-t-on ces fonctions « inefficaces » ? L'illustrer par un exemple où l'exécution de la fonction demande un nombre quadratique d'appels récursifs à la concaténation.

Les fonctions de parcours écrites plus haut sont inefficaces car elles utilisent l'opérateur `@`, ce qui entraîne un nombre quadratique d'opérations selon la taille de l'arbre. Dans la suite, on va écrire des fonctions de parcours efficaces utilisant une fonction auxiliaire et un accumulateur pour éviter ce problème.
Dans la suite, on va travailler avec le type suivant, permettant de représenter des arbres binaires stricts dont les nœuds internes et les feuilles peuvent être étiquetées par deux types différents.

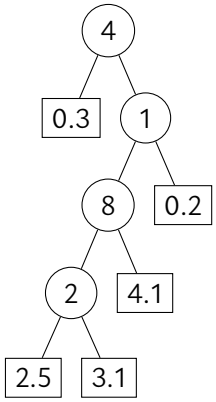
```OCaml
type ('a, 'b) arbre =
  (* 'a est le type étiquettes des feuilles *)
  | Feuille of 'a
  (* 'b est le type des étiquettes des nœuds internes *)
  | Noeud of 'b * ('a, 'b) arbre * ('a, 'b) arbre
```

Exercice 2 – Rappel de fonctions élémentaires sur les arbres

▶ **Question 1** Représenter graphiquement l'arbre suivant :

```ocaml
let exemple1 =
  Noeud (12,
    Noeud (4,
      Noeud (7, Feuille true, Feuille false),
      Noeud (14, Feuille false, Feuille true)),
  Feuille false)
```

▶ **Question 2** De même, proposer une représentation en OCaml de l'arbre suivant, nommée `exemple2 : (float, int) arbre` :



★ **Question 3** Écrire une fonction `taille : ('a, 'b) arbre -> int` qui calcule le nombre de nœuds (internes ou feuilles) d'un arbre. De même, écrire une fonction `hauteur : ('a, 'b) arbre -> int` qui calcule la hauteur d'un arbre.

★ **Question 4** Écrire une fonction `dernier : ('a, 'b) arbre -> 'a` qui renvoie l'étiquette de la feuille la plus à droite de l'arbre.

Exercice 3 – Liste des étiquettes

L'objectif maintenant est d'implémenter une fonction qui, à un arbre binaire de type `('a, 'b) arbre` 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
```

Ainsi, les fonctions de cet exercice auront le type `('a, 'b) arbre -> ('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 s'inspirera fortement de la fonction `postfixe_inefficace` portant sur le type d'arbre précédent, notamment en utilisant de la même manière l'opérateur de concaténation @.*

```OCaml
# postfixe_naif exemple2 ;;
- : (float, int) token list
= [F 0.3; F 2.5; F 3.1; N 2; F 4.1; N 8; F 0.2; N 1; N 4]
```

Là encore, le temps de calcul de cette fonction sera élevée à cause des concaténations successives, avec le même contre-exemple que dans l'Exercice 1.

▶ **Question 2** Écrire une fonction `postfixe` renvoyant la liste des étiquettes des nœuds d'un arbre dans l'ordre postfixe. *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 interdit l'utilisation de l'opérateur `@`*[a].

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

a. Et pour ceux qui ont la ref, cela donnera une complexité linéaire en la taille de l'arbre.

Exercice 4 – ★ Adressage

Dans un arbre binaire, chaque nœud peut être identifié par une séquence d'éléments $b_0, b_1, \dots, b_{p-1} \in \{\leftarrow, \rightarrow\}$ indiquant le chemin à suivre (vers le fils gauche pour $\leftarrow$ ou le fils droit pour $\rightarrow$) pour atteindre le nœud depuis la racine. Par exemple, dans l'exemple 2, le nœud interne 2 est identifié par la séquence $\rightarrow\leftarrow\leftarrow$. Informatiquement, on utilisera un booléen `false` pour $\leftarrow$ et `true` pour $\rightarrow$. Dans le sujet, on utilisera l'entier 0 pour $\leftarrow$ et 1 pour $\rightarrow$.

★ **Question 1** Dresser le tableau des adresses des nœuds de l'exemple 2.

★ **Question 2** Écrire une fonction `lire_etiquette : bool list -> ('a, 'b) arbre -> ('a, 'b) token` prenant en argument une adresse et un arbre et renvoyant l'étiquette du nœud (interne ou feuille) correspondant à l'adresse dans l'arbre. *La fonction lèvera une exception si l'adresse ne correspond pas à un nœud de l'arbre.*

★ **Question 3** Écrire une fonction `incremente : bool list -> (int, int) arbre -> (int, int) arbre` prenant en argument une adresse et un arbre et renvoyant un arbre identique à l'arbre donné, sauf que l'étiquette du nœud correspondant à l'adresse est augmentée de 1.

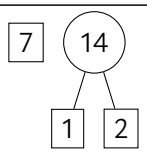
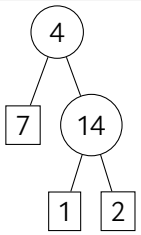
Exercice 5 – Reconstruction d'un arbre à partir de son parcours

L'objectif de cet exercice est d'écrire des fonctions de type `('a, 'b) token list -> ('a, 'b) arbre` 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 qui reconstruit morceau par morceau l'arbre recherché en parcourant la liste de ses étiquettes. Quand on lit une étiquette `e` :

- si `e = F x`, on rajoute cette feuille à la liste de droite ;
- si `e = N y`, on retire les deux premiers éléments de la liste de droite `d` et `g` et on ajoute à la liste `Noeud (y, g, d)`.

Liste des étiquettes (*tête de la liste à gauche*)	Liste des morceaux d'arbres *tête de la liste à droite*
`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`	7
`F 2; N 14; N 4; F 20; N 12`	7 1
`N 14; N 4; F 20; N 12`	7 1 2
`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` reconstruisant un arbre à partir de son parcours postfixe. *On pourra utiliser une fonction auxiliaire aux : ('a, 'b) arbre list -> ('a, 'b) token list -> ('a, 'b) arbre.*

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

Exercice 6 – Parcours dans les arbres généraux

Les parcours d'arbres préfixe et postfixe se généralisent très bien aux arbres généraux, en continuant à parcourir les fils d'un noeud de gauche à droite. On utilisera le type ci-dessous :

```OCaml
type 'a arbre_gen = N of 'a * 'a arbre_gen list
```

▶ **Question 1** Écrire une fonction `prefixe_general : 'a arbre_gen -> 'a list` renvoyant la liste des étiquettes des nœuds d'un arbre général dans l'ordre préfixe. Faire de même pour le parcours postfixe.

★ **Question 2** Si vous avez utilisé l'opérateur `@` dans les fonctions précédentes, écrire des fonctions efficaces n'utilisant pas de concaténation de listes.