# Abres en OCaml

Dans tout le début du TP, on utilisera le type suivant :

```
type ('a, 'b) arbre =
  | Feuille of 'b
  | Interne of 'a * ('a, 'b) arbre * ('a, 'b) arbre
```

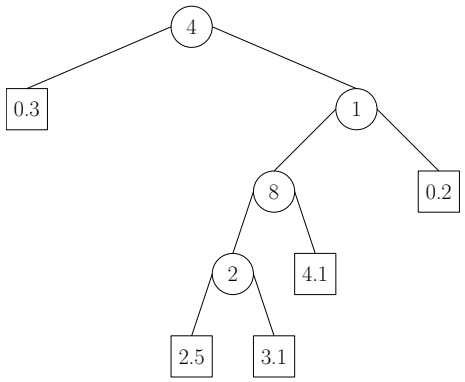
## 1 Fonctions élémentaires sur les arbres

### 1.1 Exercice

1. Représenter graphiquement l'arbre suivant :

```
let exemple1 =
  Interne (12,
           Interne (4,
                    Interne (7, Feuille 20, Feuille 30),
                    Interne (14, Feuille 1, Feuille 2)),
           Feuille 20)
```

2. Donner le type et la définition en OCaml de l'arbre ci-contre :




L'arbre `exemple2`.

### 1.2 Exercice

1. Écrire une fonction `hauteur : ('a, 'b) arbre -> int` qui calcule la hauteur d'un arbre (définie comme la longueur maximale d'un chemin reliant la racine à une feuille).
   On doit avoir `hauteur exemple1 = 3` et `hauteur exemple2 = 4`.
2. Écrire une fonction `taille : ('a, 'b) arbre -> int` qui renvoie le nombre total de nœuds (internes ou non) dans un arbre.
   On doit avoir `taille exemple1 = 9` et `taille exemple2 = 9`.
3. Écrire une fonction `dernier : ('a, 'b) arbre -> 'b` qui renvoie l'étiquette de la feuille située le plus à droite de l'arbre.
   On doit avoir `dernier exemple1 = 20` et `dernier exemple2 = 0.2`.

## 2 Parcours d'arbres

### 2.1 À la main

Donner la liste des étiquettes de l'arbre `exemple2` :

1. dans l'ordre du parcours en largeur.
2. dans l'ordre préfixe.

3. dans l'ordre infixe.
4. dans l'ordre postfixe.

## 2.2 Parcours en profondeur

1. Écrire une fonction `affiche_prefixe : (int, int) arbre -> unit` qui affiche toutes les étiquettes des nœuds (internes ou non) de l'arbre passé en argument dans l'ordre préfixe. On utilisera :
   — `print_int : int -> unit` pour afficher un entier ;
   — `print_newline : unit -> unit` pour revenir à la ligne.
2. Écrire de même des fonctions `affiche_infixe` et `affiche_postfixe`.

```
# affiche_prefixe exemple1;;
12
4
7
20
30
14
1
2
20
- : unit = ()
```

## 2.3 Liste des étiquettes

On souhaite écrire des fonctions permettant d'obtenir la liste des étiquettes d'un arbre dans un ordre spécifié (préfixe, infixe ou postfixe). Comme les étiquettes des nœuds internes et celles des feuilles peuvent être de types différents, il nous faut pour cela créer un type à deux variantes :

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

Les fonctions à écrire seront donc de type `('a, 'b) arbre -> ('a, 'b) token list`.

1. Écrire une fonction `postfixe_naif` la plus simple possible.
2. Expliquer pourquoi cette fonction risque d'être inefficace.
3. Écrire une fonction `postfixe` plus efficace, et préciser sa complexité. On utilisera une fonction

   `aux : ('a, 'b) arbre -> ('a, 'b) token list -> ('a, 'b) token list`

   telle que `aux a liste` renvoie `u @ liste`, où `u` est la liste des étiquettes de $a$ dans l'ordre postfixe.
   *Cette fonction ne fera aucune concaténation : le* `u @ liste` *est juste là pour donner la spécification.*
4. En utilisant la même technique, écrire des fonctions `prefixe` et `infixe` efficaces.

## 2.4 Version récursive terminale

On souhaite écrire une version récursive terminale de la fonction `postfixe`. La difficulté vient du fait qu'un appel à `postfixe` sur un arbre de la forme `Interne (x, g, d)` donne *a priori* deux appels récursifs : un sur l'arbre `g` et un sur `d`. Évidemment, au plus l'un de ces appels peut être en position terminale. Pour surmonter cette difficulté, on va maintenir une liste d'arbres à traiter (on parle de *forêt*) ; à chaque étape, on récupérera le premier élément de cette forêt, traitera sa racine, et, le cas échéant, ajoutera ses sous-arbres gauche et droit à la forêt. Le principe est illustré ci-dessous :

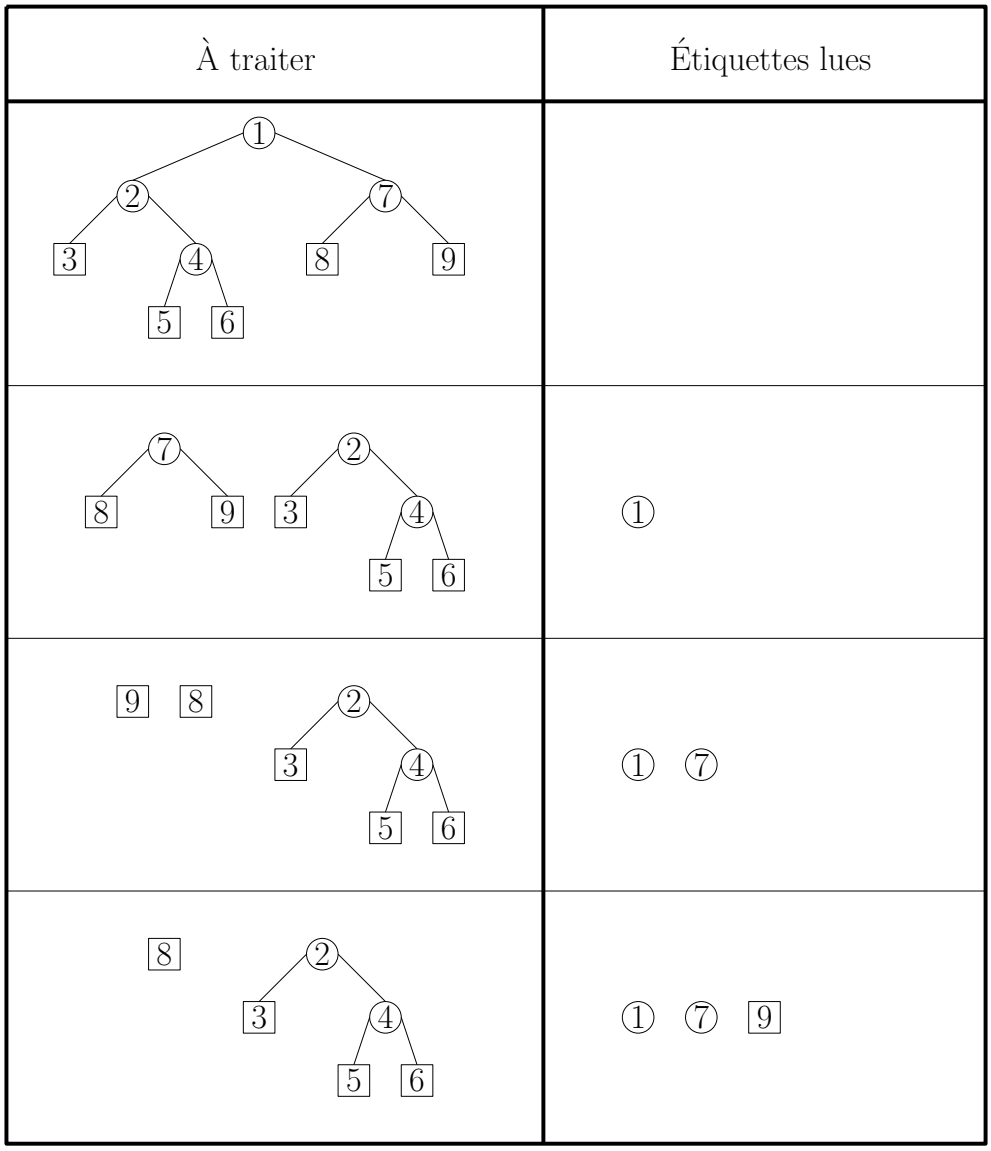



FIGURE 1 – Principe du parcours en profondeur récursif terminal

1. Poursuivre l'exécution simulée de l'algorithme jusqu'à sa conclusion.
2. Écrire une fonction `postfixe_term` ayant les mêmes spécifications que `postfixe` mais étant récursive terminale. On utilisera une fonction auxiliaire

   ```
   aux : ('a, 'b) arbre list -> ('a, 'b) token list -> ('a, 'b) token list
   ```

   prenant en entrée la forêt qui reste à traiter et les étiquettes déjà lues, et renvoyant la liste complète des étiquettes en suivant l'algorithme illustré ci-dessus.

## 2.5 Parcours en largeur

On souhaite écrire une fonction `largeur : ('a, 'b) arbre -> ('a, 'b) token list` similaire à celle écrite à l'exercice précédent. Pour ce faire, on va utiliser l'algorithme illustré ci-dessous :

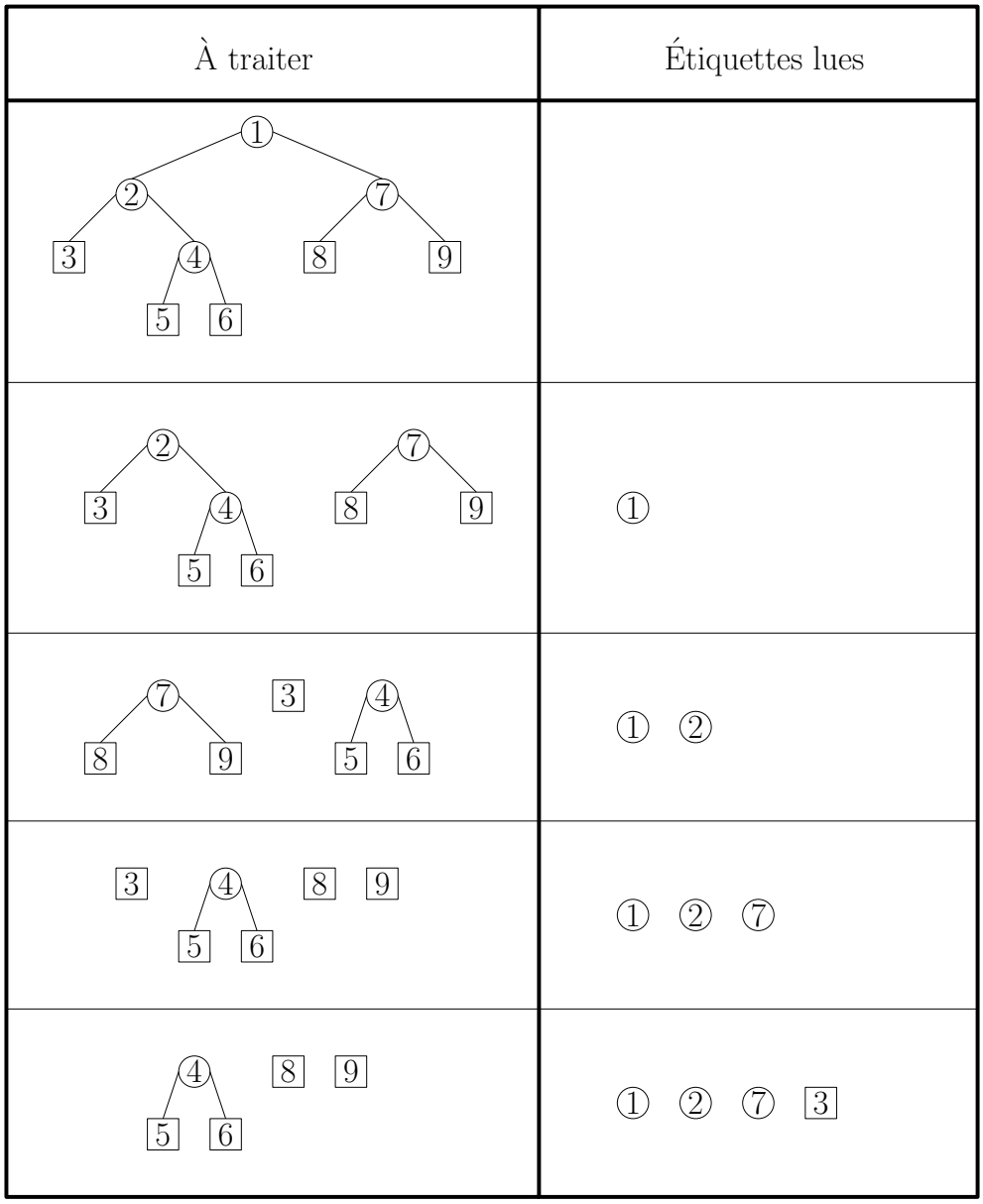



FIGURE 2 – Algorithme de parcours en largeur

1. Poursuivre l'exécution simulée de cet algorithme jusqu'à sa conclusion.
2. Expliquer pourquoi il est raisonnable d'utiliser une liste pour stocker les étiquettes lues, et pourquoi faire de même pour la forêt à traiter pose un problème de performances.
3. Quelle structure de données étudiée cette année peut-on utiliser pour stocker de manière efficace les arbres à traiter ?
4. Écrire la fonction `etiquettes_largeur` ; on pourra se référer aux indications présentes dans le squelette fourni, et écrire au choix une fonction récursive terminale ou itérative.

# 3 Adressage

Dans un arbre binaire, un nœud situé à profondeur $p$ (où la profondeur de la racine vaut 0) peut être repéré par une suite de $p$ éléments $(x_0, \ldots, x_{p-1})$ de $\{\leftarrow, \rightarrow\}$ : on part de la racine et pour $i = 0 \ldots p-1$ on descend vers le fils gauche ou droit du nœud actuel suivant que $x_i$ vaut $\leftarrow$ ou $\rightarrow$. Informatiquement, les valeurs $\leftarrow$ et $\rightarrow$ seront codées par les booléens `false` et `true` ou par les entiers `0` et `1`.

Pour l'arbre `exemple1`, on obtient :

| Adresse | Étiquette |
|---|---|
| ∅ | 12 |
| 0 | 4 |
| 00 | 7 |
| 000 | 20 |
| 001 | 30 |
| 01 | 14 |
| 010 | 1 |
| 011 | 2 |
| 1 | 20 |

### 3.1 Exercice

1. Dresser le tableau des adresses et étiquettes correspondantes pour l'arbre `exemple2`.
2. Écrire une fonction `lire_etiquette : bool list -> ('a, 'b) arbre -> ('a, 'b) token` prenant en arguments une adresse et un arbre et renvoyant l'étiquette correspondante. Si l'adresse ne correspond à aucun nœud, on lèvera une exception à l'aide de `failwith`.

```
utop[50]> lire_etiquette [false; true; false] exemple1;;
- : (int, int) token = F 1
utop[51]> lire_etiquette [false] exemple1;;
- : (int, int) token = N 4
```

3. Écrire une fonction `incremente : (int, int) arbre -> adresse -> (int, int) arbre` prenant en entrée un arbre $\mathcal{T}$ et une adresse $s$ et renvoyant l'arbre $\mathcal{T}'$ obtenu à partir de $\mathcal{T}$ en incrémentant de un l'étiquette du nœud d'adresse $s$. À nouveau, on lèvera une exception si l'adresse est invalide.
4. Écrire une fonction `tableau_adresses : (int, int) arbre -> unit` affichant la liste des adresses de l'arbre avec les étiquettes correspondantes. On s'aidera de la fonction `affiche_avec_adresse` fournie.

```
utop[58]> affiche_avec_adresse (12, [true; false; false]);;
100 : 12
utop[59]> tableau_adresses exemple1;;
 : 12
0 : 4
00 : 7
01 : 14
010 : 1
011 : 2
1 : 20
```

## 4 Reconstruction d'un arbre

On s'intéresse ici à la possibilité d'écrire des fonctions réciproques de `prefixe`, `postfixe`, `largeur`, `infixe`. Notre entrée est donc une `('a, 'b) token list` (dont on sait à quel ordre elle correspond), et notre sortie doit être l'unique `('a, 'b) arbre` donnant cette liste.

### 4.1 Exercice

1. Donner deux arbres distincts dont le parcours infixe produit la liste `[F 0; N 1; F 2; N 3; F 4]`. Que peut-on en conclure ?
2. Pour reconstruire l'arbre à partir de son parcours postfixe, on propose l'algorithme illustré ci-dessous :

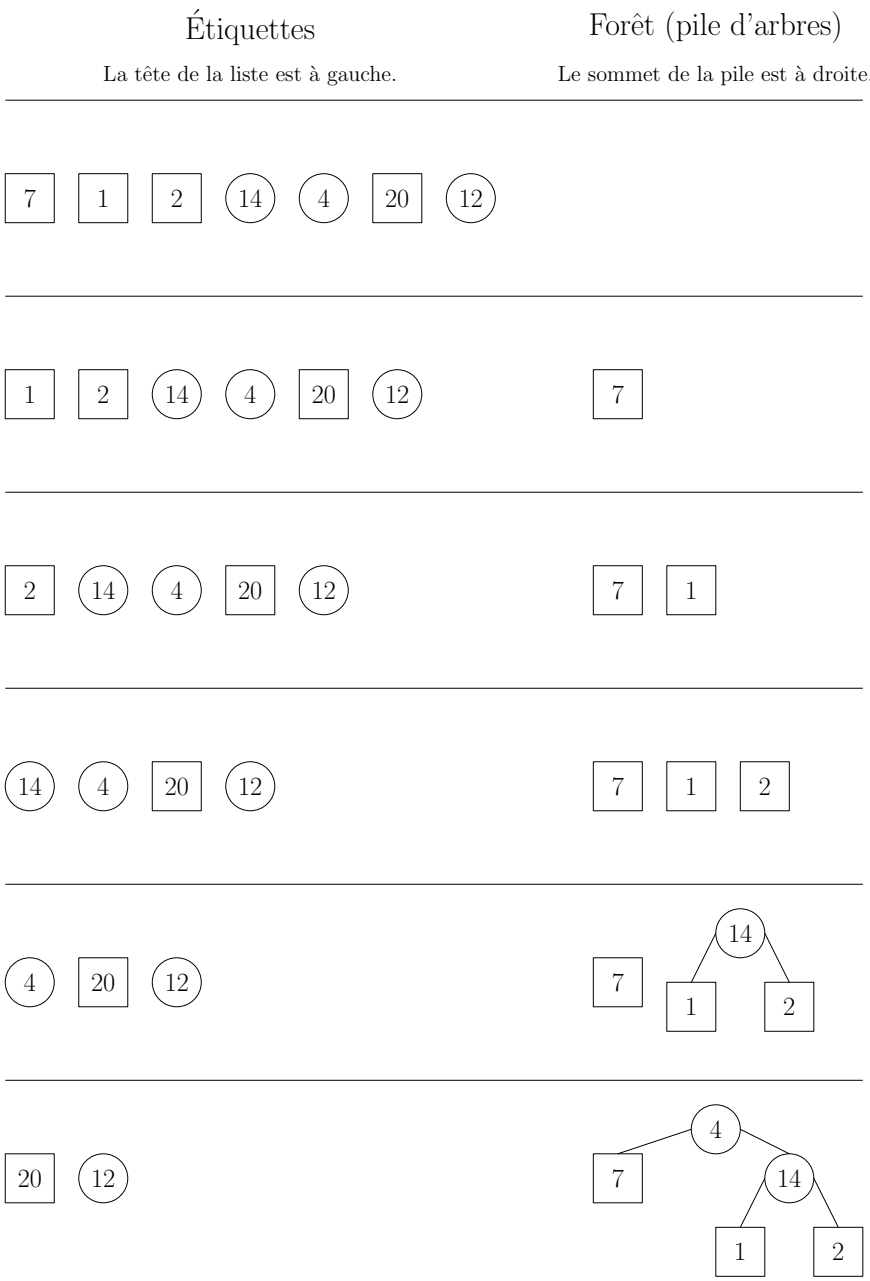



FIGURE 3 – Principe de reconstruction à partir du parcours postfixe.

(a) Terminer la simulation de l'exécution de l'algorithme sur l'exemple donné.

(b) À l'aide d'une fonction auxiliaire
`aux : ('a, 'b) arbre list -> ('a, 'b) token list -> ('a, 'b) arbre`, écrire une fonction `lire_postfixe` reconstruisant un arbre à partir du résultat de son parcours postfixe.

3. Écrire une fonction `lire_prefixe` (on peut utiliser une idée similaire).
4. Écrire une fonction `lire_largeur` (il faudra utiliser une file).

# 5 Au cas où

## 5.1 Retour sur les nombres de Catalan

On considère les types suivants :

```
type cote = G | D

(* arbre binaire entier non étiqueté *)
type arbre = Feuille | Noeud of arbre * arbre

(* on annote chaque noeud interne avec le cardinal du sous-arbre
   correspondant *)
type arbre_annote =
  | F
  | N of int * arbre_annote * arbre_annote
```

ainsi que la fonction suivante qui renvoie le cardinal (nombre total de nœuds) d'un arbre annoté :

```
(* arbre_annote -> int *)
let card = function
  | F -> 1
  | N(taille, _, _) -> taille
```

1. Écrire une fonction `annote : arbre -> arbre_annote` (dont la spécification devrait aller de soi).
2. Écrire une fonction `insere : arbre_annote * int * cote -> arbre_annote * int` correspondant à la fonction *insere* du cours. L'entier pris en argument sera donc un numéro valable de nœud pour l'arbre de départ (dans l'ordre préfixe) et l'entier renvoyé un numéro valable de feuille dans l'arbre d'arrivée.
3. Écrire une fonction `efface : arbre_annote * int -> arbre_annote * int * cote`, réciproque de la précédente.
4. Écrire une fonction `test` prenant en entrée un `arbre` et vérifiant sur cet arbre que les deux fonctions précédentes sont bien réciproques l'une de l'autre (on fera toutes les insertions et toutes les suppressions légales).