# TP10: Arbres

MP2I Lycée Pierre de Fermat

Ce TP comporte 4 exercices: pour chacun, créez un fichier dans lequel vous mettrez les types et fonctions implémentées, ainsi que les tests. Vous ***devez*** commenter vos fonctions et faire des tests: ce n'est pas facultatif ! Chaque exercice devra avoir une fonction de test de la forme suivante:

```
let test () =
  assert (factorielle 5 = 120);
  assert (factorielle (-1) = 1);
  assert (est_vide []);
  let l = [1;2;3] in
  assert (not (est_vide l));
  (* ... *)
  assert (is_provable hypothese_riemann);
  print_string "Fin des tests\n" (* Pas de point virgule pour la dernière ligne *)
;;
```

Pour réaliser ces tests, imaginez que vous les écrivez dans le but de tester le code d'un ou d'une de vos camarades, et que vous voulez à tout pris mettre le mettre en échec: il faut tester les cas limites, bien couvrir tout le code, bref, vérifier que les fonctions respectent leur spécification ***à la lettre***. Le TP est fait de telle sorte que les noms des fonctions et des types sont fixés. Donc, les tests que vous écrivez peuvent directement être appliqués pour tester le code d'autres élèves. Vous êtes invités à partager vos tests (par exemple en les mettant dans des fichiers à part), en revanche le code que vous rendez doit être personnel.

# Exercice 1: Récursivité Mutuelle

En OCaml, on peut définir plusieurs fonctions récursives en même temps, de telle sorte que chacune peut référencer les autres. Pour cela, on utilise Par exemple, si l'on veut écrire des fonctions qui calculent les suites:

$$u_n = \begin{cases} 1 & \text{si } n = 0 \\ u_{n-1} + v_{n-1} - 1 & \text{sinon} \end{cases} \qquad v_n = \begin{cases} 2 & \text{si } n = 0 \\ v_{n-1} + u_n & \text{sinon} \end{cases}$$

on écrira:

```
let rec u n =
  if n = 0 then 1
  else u (n-1) + v (n-1) - 1
(* le mot-clé and sert à lier récursivement les deux fonctions définies *)
and v n =
  if n = 0 then 2
  else v (n-1) + u n
;;
```

**Question 1.** Écrire une fonction `diviseurs: int -> int list` qui renvoie la liste des diviseurs stricts d'un entier non nul.

**Question 2.** On définit la suite $(w_n)_{n\in\mathbb{N}^*}$ comme suit:

$$\begin{aligned} w_1 &= 1 \\ \forall n \in \mathbb{N}, w_n &= 1 + \sum_{d|n, d\neq n} w_d \end{aligned}$$

Écrire deux fonctions mutuellement récursives `w: int -> int` et `list_w: int list -> list` telles que `w n` calcule $w_n$, et `list_w l` calcule la somme des $w_d$ pour $d \in l$.

**Question 3.** On considère $(p_n)_{n\in\mathbb{N}^*}$ la suite des nombres premiers dans l'ordre croissant. Pour $n \in \mathbb{N}$, on note $b_n = w_{p_1\times\cdots\times p_n}$. Calculez les quelques premiers termes de $(b_n)_{n\in\mathbb{N}}$ et trouvez sur internet le nom d'une suite qui semble correspondre.

On s'intéresse maintenant aux expressions arithmétiques et booléennes. On veut pouvoir comparer les expressions arithmétiques, et avoir une construction équivalente au "if-then-else" d'OCaml. On souhaite souhaite donc avoir deux types, un pour les expressions booléennes et un pour les expressions arithmétiques, et les deux types doivent se référencer mutuellement. Ici aussi, on peut utiliser le mot-clé ***and***:

```
type bool_expr =
  | Or of bool_expr * bool_expr
  | And of bool_expr * bool_expr
  | Not of bool_expr
  | Equal of arith_expr * arith_expr
and arith_expr =
  | Int of int
  | Variable of string
  | Add of arith_expr * arith_expr
  | Mult of arith_expr * arith_expr
  | IfTE of bool_expr * arith_expr * arith_expr (*if b then e1 else e2 *)
;;
```

Les listes de type `(string * int) list` serviront à représenter un ***contexte***, c'est à dire une association entre les variables et les entiers:

```
type context = (string * int) list ;;
```

**Question 4.** Écrire une fonction `get_var: (string -> context -> int` telle que `get_var s l` cherche dans $l$ un couple $(s, n)$ et renvoie l'entier $n$ correspondant.

**Question 5.** Écrire deux fonctions d'évaluation `eval_bool: bool_expr -> context -> bool` et `eval_arith: arith_expr -> context -> int` qui permettent les expressions booléennes et arithmétiques dans un contexte donné.

# Exercice 2: Arbres binaires

On propose le type suivant pour les arbres binaires:

```
type 'a ab =
  | V (* Vide *)
  | N of 'a * 'a ab * 'a ab  (* Noeud: elem, gauche, droite *)
;;

(* Exemple *)
let t = N (3, N (5, V, N (8, V, V)), N (7, V, V)) ;;
```

**Question 6.** Écrire des fonctions `hauteur` et `taille` permettant de calculer la hauteur et la taille d'un arbre binaire.

**Question 7.** Écrire une fonction `est_feuille` qui détermine si un arbre binaire est une feuille, puis une fonction `feuilles` comptant le nombre de feuilles d'un arbre, ainsi qu'une fonction `internes` comptant le nombre de noeuds internes d'un arbre.

On représente un chemin dans un arbre binaire par une liste de booléens: true indique que l'on part à droite, et false que l'on part à gauche.

**Question 8.** Écrire une fonction `etiquette: 'a ab -> bool list -> 'a` qui renvoie l'étiquette du noeud correspondant à un chemin dans un arbre. Si le chemin n'est pas valide, une exception sera levée.

**Question 9.** Si $a, b$ sont deux arbres binaires, on dit qu'ils sont **miroir** l'un de l'autre si $a$ est le symétrique de $b$ par rapport à un axe vertical. Écrire une fonction `est_miroir` permettant de calculer si deux arbres binaires sont miroir l'un de l'autre.

**Question 10.** Écrire une fonction `miroir` qui calcule, pour un arbre binaire $a$ donné, un arbre $b$ qui soit son miroir.

**Question 11.** On dit qu'un arbre $a$ est symétrique s'il est miroir de lui même. Écrire une fonction `est_symetrique` qui calcule si un arbre est symétrique.

**Question 12.** Écrire une fonction `liste_prefixe: 'a ab -> 'a list` calculant la liste des éléments d'un arbre binaire, dans l'ordre préfixe.

**Question 13.** Écrire des fonctions `liste_postfixe` et `liste_infixe` permettant d'implémenter les deux parcours correspondants.

**Question 14.** (*Bonus*) Réécrire ces fonctions sans utiliser l'opérateur de concaténation `@`.

**Question 15.** Écrire une fonction `tree_map: ('a -> 'b)-> 'a ab -> 'b ab` qui applique une fonction $f$ sur chaque noeud d'un arbre $a$.

**Question 16.** Écrire une fonction `tree_sum: int ab -> int` calculant la somme des entiers contenus dans un arbre binaire.

**Question 17.** Écrire une fonction `appartient : 'a -> 'a ab -> bool` déterminant si un élément se trouve dans un arbre binaire.

**Question 18.** (*Bonus*) Déterminer un schéma commun dans les deux dernières fonctions, et proposer une fonction `tree_fold` équivalente de `fold` sur les listes.

# Exercice 3: Arbres quelconques

On souhaite étudier maintenant des arbres d'arité quelconque. Un noeud ne stockera plus un tuple d'enfants mais une liste. On ne représente plus l'arbre vide.

```
type 'a tree =
  Node of 'a * ('a tree list)  (* Noeud: étiquette, liste des enfants *)
;;
```

Dans cette partie, nous allons implémenter plusieurs opérations sur les arbres. Certaines questions demandent de coder des fonctions auxiliaires, et d'utiliser les fonctions des listes déjà implémentées dans OCaml: map, fold, filter, etc... Vous pouvez trouver la documentation des différentes fonctions utiles sur les listes ici: v2.ocaml.org/api/List.html. Cette page contient de nombreuses fonctions permettant de manipuler les listes, je vous conseille de l'explorer en long et en large, et même d'essayer de recoder certaines de ces fonctions avec vos propres moyens, pour vous entraîner.

Pour vous guider, voici deux versions de la fonction qui calcule le nombre de mots contenus dans un arbre préfixe:

```
(* Calcule la taille de l'arbre Node(b, l) *)
let rec taille (Node(x, l) : 'a tree) : int =
  1 + taille_liste l

(* Calcule la somme des tailles des arbres
   dans l *)
and taille_liste (l: 'a tree list) : int =
  match l with
  | [] -> 0
  | d :: q -> taille d + taille_liste q
;;
```

```
(* Calcule la taille de l'arbre Node(b, l) *)
let rec taille2 (Node(x, l) : 'a tree) : int =
  1 + List.fold_left (+) 0 (List.map taille2 l)
;;
```

La première utilise la syntaxe `and` pour définir deux fonctions mutuellement récursives qui calculent la taille d'un arbre et la somme des tailles des arbres d'une liste, la deuxième utilise les fonctions `List.fold` et `List.map` pour appliquer directement la fonction récursivement sur la liste des enfants et faire la somme.

**Question 19.** Implémenter une fonction `hauteur` qui calcule la hauteur d'un arbre

**Question 20.** Implémenter une fonction `etiquette` qui prend en entrée un arbre et une liste d'entiers $[a_1; \ldots; a_n]$ représentant un chemin, et qui renvoie l'étiquette du noeud correspondant. Si le chemin n'est pas valide dans l'arbre, la fonction lèvera une exception.

**Question 21.** Écrire les fonctions `liste_prefixe` et `liste_postfixe` de parcours préfixe et postfixe.

On introduit un nouveau type de parcours: les ***parcours en largeur***. Un tel parcours énumère les noeuds d'un arbre par ordre croissant de profondeur. Il est plus facile de décrire ces parcours de manière itérative, avec une boucle, qu'avec de la récursivité. On utilise une file pour effectuer des parcours en profondeur:

**Algorithme 1 :** Parcours en profondeur

**Entrée(s) :** $A$ un arbre

$r \leftarrow$ la racine de $A$ ;
$F \leftarrow$ une file vide ;
$F$.**enfiler**($r$);
**tant que** $F \neq \emptyset$ **faire**
    $u \leftarrow F$.**defiler**();
    Traiter $u$;
    **pour** $v$ ***enfant de*** $u$ **faire**
        $F$.**enfiler**($v$);

**Remarque 1.** Si l'on remplace la file par une pile, on retrouve les parcours en profondeur.

Voyons comment implémenter récursivement ces parcours. Nous allons utiliser deux listes, qui en OCaml se comportent comme des piles, pour simuler une file. Plus précisément, on utilisera une contion auxiliaire comme suit:

```
let liste_largeur (a: 'a tree) =
  let rec aux (l_actuel: 'a tree list) (l_suivant: 'a tree list) -> 'a list =
    ...
  in aux [a] [] ;;
```

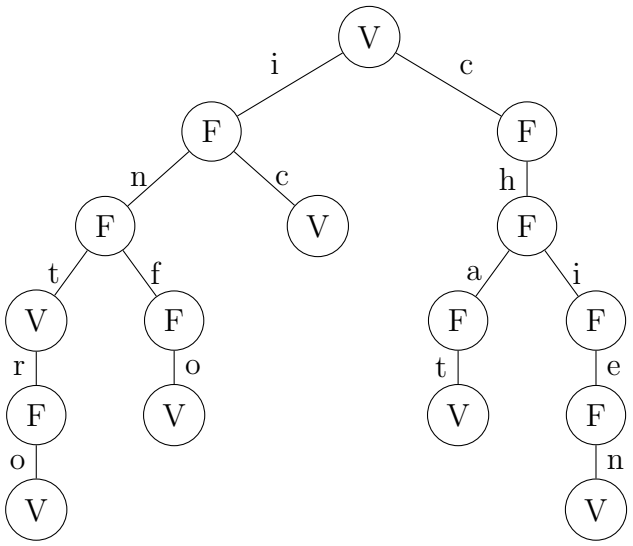
La spécification de la fonction auxiliaire est qu'elle calcule la liste des éléments dans les arbres des listes `l_actuel` et `l_suivant`. La liste `l_actuel` correspondra aux arbres du niveau actuel regardé, et la liste `l_suivant` correspondra aux arbres du niveau suivant.

Ainsi, la fonction auxiliaire devra vérifier l'invariant suivant: `l_suivant` contient des noeuds de l'arbre initial d'un profondeur d'exactement 1 de plus que les noeuds de `l_actuel`. En particulier, chacune des deux listes contiendra des noeuds d'une seule profondeur à la fois.

**Question 22.** Implémentez le parcours en largeur par une fonction `liste_largeur`.

# Exercice 4: Arbres préfixes

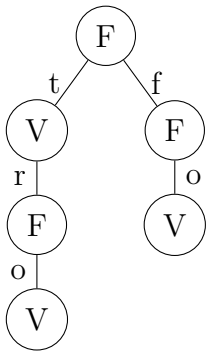
Les arbres préfixes servent à stocker des ensembles de mots. L'idée est de stocker sur chaque arête entre un noeud et son père une lettre. Un noeud représentera le mot formé de toutes les lettres entre lui et la racine. Par exemple:



Les noeuds contenant des V (comme vrai) correspondent à des mots. Dans l'exemple au dessus, l'arbre représente l'ensemble $\{\varepsilon$, int, intro, info, chat, chien $\}$. On propose le type suivant pour représenter ces ensembles:

```
type pre_tree = Node of (bool * (char * pre_tree) list ) ;;
```

Par exemple, voici un arbre préfixe et sa représentation sous ce type en OCaml:




```
let t =
Node(false, [
  ('t', Node(true, [
    ('r', Node(false, [
      ('o', Node(true, [])) ])) ])) ;
  ('f', Node(false, [
    ('o', Node(true, [])) ])) ])) ;;
```

On peut transformer une liste de caractères en un string comme suit en OCaml:

```
let string_of_list l = String.of_seq (List.to_seq l) ;;
```

**Question 23.** Écrire une fonction `taille` qui calcule le nombre de mots contenus dans un arbre préfixe.

**Question 24.** Écrire une fonction `plus_long` qui calcule la longueur du plus long mot contenu dans un arbre préfixe.

**Question 25.** Écrire une fonction `rechercher: string -> pre_tree -> bool` qui détermine si un mot est dans un arbre préfixe.

**Question 26.** Écrire une fonction `ajouter: string -> pre_tree -> pre_tree` qui ajoute un mot à un arbre préfixe.

**Question 27.** En vous renseignant sur la fonction `read_line` ainsi que sur le module `String` (v2.ocaml.org/api/String.html), implémentez une fonction `construire: pre_tree -> pre_tree` qui lit une ligne dans le terminal, et rajoute tous les mots (séparés par des espaces) dans l'arbre préfixe donné en argument.

**Question 28.** Écrire une fonction `enumerer` qui renvoie la liste des mots contenus dans un arbre préfixe, par ordre alphabétique.

**Question 29.** (*Bonus*) Rendre la fonction précédente linéaire en la taille de l'arbre (donc interdit d'utiliser une fonction de tri ou de concaténer des listes à tout va). Il pourra être utile de modifier les fonctions précédentes pour obliger les enfants d'un noeud à être stockés dans un ordre précis.

**Question 30.** Écrire une fonction `enumerer_prefixe: string -> pre_tree -> string list` telle que `enumerer_prefixe s t` renvoie la liste des mots de $t$ qui commencent par $s$.

**Question 31.** Écrire une fonction `fusion` permettant de faire l'union de deux arbres préfixes. Attention à ne pas créer de doublons.