université de BORDEAUX	**Année universitaire :** 2024-2025 **Parcours :** Licence Informatique 3e année **UE :** Programmation fonctionnelle UE 4TIN514U **Épreuve :** Examen de Programmation fonctionnelle **Date :** mercredi 18 décembre 2024 9 :00 – 10 :30 **Durée :** 1h30 **Documents interdits.**	**Collège Sciences et Technologies**

Exercice 1 *(4pts)* Soit la fonction `mystere l` présentée par la figure 1.

Que retournent (type et valeur) les expressions suivantes ?

1. `mystere []`
2. `mystere [1; 1; 1]`
3. `mystere [1; 2; 1; 3; 2; 1; 3]`

```
let mystere l =
  List.fold_right (fun a b -> if List.mem a b then b else a :: b) l []
```

FIG. 1 : Fonction `mystere`

4. Expliquer en une phrase ce que fait la fonction `mystere`.
5. Réécrire la fonction `mystere` **sans utiliser** de fonction de type fold (`fold_left`, `fold_right`) et qui soit **récursive terminale**.

Exercice 2 *(11pts)*

6. Écrire une fonction **récursive terminale**, `count_fst x l` qui compte le nombre d'éléments `x` consécutifs en tête de la liste `l`.

```
# count_fst;;
- : 'a -> 'a list -> int = <fun>
# count_fst 5 [5; 5; 5; 1; 2; 3; 5];;
- : int = 3
# count_fst 5 [1; 5; 5];;
- : int = 0
```

7. Écrire une fonction **récursive terminale**, `remove_fst x l` qui *supprime* les éléments `x` consécutifs présents en tête de la liste `l`.

```
# remove_fst;;
- : 'a -> 'a list -> 'a list = <fun>
# remove_fst 5 [5; 5; 5; 1; 2; 3; 5];;
- : int list = [1; 2; 3; 5]
```

On considère des listes d'entiers naturels. On définit la fonction `suivant l` qui est définie de la manière suivante.

Si $l = \{l_0, \ldots, l_{n-1}\}$ alors $suivant(l) = \{c_0, e_0, c_1, e_1, \ldots c_{k-1}, e_{k-1}\}$ où k est le nombre de changements de valeurs dans la liste l, $\{e_0, \ldots, e_{k-1}\}$ sont dans l'ordre, les éléments de l qui sont différents de leur prédécesseur dans la liste lue de gauche à droite ($k < n$) et chaque c_i est égal au nombre de e_i consécutifs dans l. Exemples :

$\{20\}$ → `un 20`
$\{1, 20\}$ → `un 1, un 20`
$\{1, 1, 1, 20\}$ → `trois '1', un 20`
$\{3, 1, 1, 20\}$ → `un 3, deux 1, un 20`
$\{1, 3, 2, 1, 1, 20\}$ → `un 1, un 3, un 2, deux 1, un 20`

Exemple détaillé :

$$l = \{l_0, l_1, l_2, l_3, l_4, l_5\} = \{30, 30, 30, 10, 10, 30\}$$

$k = 3$ car trois changements en $l_0 = 30, l_3 = 10, l_5 = 30$.
$e_0 = l_0$ et $c_0 = 3$ car on a trois occurrences consécutives égales : $l_0 = l_1 = l_2 = 30$.
$e_1 = l_3$ et $c_1 = 2$ car on a deux occurrences consécutives égales : $l_3, l_4 = 10$.
$e_2 = l_5$ et $c_2 = 1$ car on a une seule occurrence $l_5 = 30$.

$$\mathit{suivant}(l) = \{c_0, e_0, c_1, e_1, c_2, e_2\} = \{3, 30, 2, 10, 1, 30\}$$

8. Écrire la fonction `suivant l` qui fait passer de la liste l à la liste *suivant(l)*. Exemples :

```
# suivant;;
- : int list -> int list = <fun>
# suivant [3; 3; 1; 1; 1; 3; 3];;
- : int list = [2; 3; 3; 1; 2; 3]
```

Soit la suite de listes d'entiers positifs définie de la façon suivante :

$$\begin{cases} u_0 &= [1] \\ u_n &= \mathit{suivant}(u_{n-1}) \texttt{ pour } n > 0 \end{cases}$$

9. Écrire la fonction `suite n` qui retourne le `n`-ième élément de cette suite. Exemples :

```
# suite;;
- : int -> int list = <fun>
# suite 0;;
- : int list = [1]
# suite 4;;
- : int list = [1; 1; 1; 2; 2; 1]
```

10. Écrire la fonction `suite_gen n first next` qui retourne le `n`-ième élément d'une suite récurrente dont le premier élément est `first` et `next` est la fonction qui fait passer d'un élément u_n au suivant u_{n+1}. Exemples :

```
# suite_gen;;
- : int -> 'a -> ('a -> 'a) -> 'a = <fun>
# suite_gen 3 [4] suivant;;
- : int list = [3; 1; 1; 4]
```

11. Réécrire la fonction `suite n` en utilisant `suite_gen`.

Exercice 3 *(5pts)*

Soit le type suivant défini pour représenter des arbres planaires dont les noeuds sont des entiers. On rappelle qu'un arbre planaire contient au moins un noeud et que chaque noeud a un nombre quelconque de noeuds fils.

```
type tree = T of int * tree list
```

L'entier `int` est la racine de l'arbre et `tree list` est la liste ordonnée des sous-arbres fils de la racine. Des exemples d'arbres sont présentés par la figure 2.

12. Écrire la fonction constructeur `make_tree root subtrees` qui fabrique un arbre à partir de la valeur de sa racine `root` et de la liste de ses sous-arbres `subtrees`. Exemples :

```
# let leaf1 = make_tree 1 [];;
val leaf1 : tree = T (1, [])
# let leaf2 = make_tree 2 [];;
val leaf2 : tree = T (2, [])
# let s2 = make_tree 3 [leaf1; leaf2];;
val s2 : tree = T (3, [T (1, []); T (2, [])])
# let s3 = make_tree 4 [leaf1; leaf2; leaf1];;
val s3 : tree = T (4, [T (1, []); T (2, []); T (1, [])])
# let tree = make_tree 5 [s3; s2; leaf1];;
val tree : tree =
  T (5,
   [T (4, [T (1, []); T (2, []); T (1, [])]); T (3, [T (1, []); T (2, [])]);
    T (1, [])])
```

Le code précédent produit les arbres présentés par la figure 2.

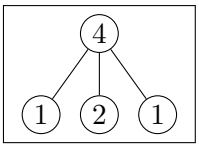
13. Écrire les fonction accesseurs `tree_root tree` et `tree_subtrees tree` qui retournent respectivement la racine et les sous-arbres de l'arbre `tree`.
14. Écrire une fonction `depth_first tree` qui retourne la liste des entiers contenus dans l'arbre `tree` obtenue par un parcours en profondeur et l'ordre préfixe[1]. Exemples :

```
# s2;;
- : tree = T (3, [T (1, []); T (2, [])])
# depth_first s2;;
- : int list = [3; 1; 2]
# s3;;
- : tree = T (4, [T (1, []); T (2, []); T (1, [])])
# depth_first s3;;
- : int list = [4; 1; 2; 1]
# tree;;
- : tree =
T (5,
 [T (4, [T (1, []); T (2, []); T (1, [])]); T (3, [T (1, []); T (2, [])]);
  T (1, [])])
# depth_first tree;;
- : int list = [5; 4; 1; 2; 1; 3; 1; 2; 1]
```

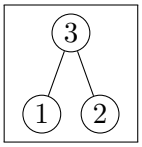
FIN

[1] Le parcours en profondeur dans l'ordre préfixe d'un arbre `T(i, [s0, ..., ])` est la liste dont le premier élément est `i` et la suite de la liste est la concaténation des parcours en profondeur de chacun des sous-arbres.

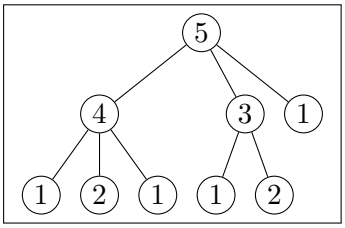
(a) `leaf1`



(b) `s3`



(c) `s2`



(d) `tree`

FIG. 2 : Arbres planaires