# TP n°11 – Programmation impérative en OCaml

## 1. Exercices avec des boucles

- **Q1.** Écrire une fonction qui prend en entrée $n$ et calcule $\sum_{k=1}^{n} 3k^3 + \ln(k) + 5$ en utilisant une boucle et une référence.
- **Q2.** Écrire une fonction impérative `fibonacci : int -> int` calculant le terme d'indice n de la suite de Fibonacci définie par $F_0 = 1, F_1 = 1$ et pour tout $n \geqslant 0$, $F_{n+2} = F_n + F_{n+1}$, en utilisant deux références.
- **Q3.** Déterminer à partir de quel rang la suite définie par $u_0 = 1$, $u_1 = 1$ et $u_{n+1} = 3 * u_n + u_{n-1} + 1$ dépasse 250 en utilisant une boucle et des références.

## 2. Algorithme d'Euclide

- **Q4.** Écrire une fonction `euclide : int -> int -> int` programmant la version impérative de l'algorithme d'Euclide.
  Rappel : l'algorithme d'Euclide sur $a \geq b$ consiste à trouver leur PGCD en remarquant que $pgcd(a, b) = pgcd(b, r)$ où $r$ est le reste de la division euclidienne de $a$ par $b$.
- **Q5.** Modifier la fonction pour qu'elle affiche les divisions euclidiennes successives. Par exemple :

**OCaml**

```
euclide_affiche (fibonacci 11) (fibonacci 10);;
89 = 55 x 1 + 34
55 = 34 x 1 + 21
34 = 21 x 1 + 13
21 = 13 x 1 + 8
13 = 8 x 1 + 5
8 = 5 x 1 + 3
5 = 3 x 1 + 2
3 = 2 x 1 + 1
2 = 1 x 2 + 0
- : int = 1
```

Indication : on pourra utiliser la fonction suivante qui imprime les choses joliment :

**OCaml**

```
let affiche_division a b =
  print_int a;
  print_string " = ";
  print_int b;
  print_string " * ";
  print_int (a / b);
  print_string " + ";
  print_int (a mod b);
  print_newline ()
;;
```

## 3. Tableaux

- **Q6.** Ecrire une fonction `range : int -> int array` qui prend en entrée $n$ et renvoie le tableau `[|0;1;...;n-1|]`.
- **Q7.** Écrire une fonction `init: int -> (int-> 'a) -> 'a array` tel que `init n f` initialise un tableau de taille `n` telle que la case d'indice `i` est `(f i)`.
  Par exemple si on définit `let f x = x*3`, alors `init f 4` renvoie `[|0;3;6;9|]`.
- **Q8.** Écrire une fonction `fibonacci_tab : int-> int array` renvoyant le tableau des $n$ premiers éléments de la suite de Fibonacci.
- **Q9.** Écrire une fonction `map: ('a-> 'b)-> 'a array -> 'b array` qui prend en entrée une fonction `f` et un tableau `t` et renvoie un tableau dont l'élément $i$ est `f tab[i]`.

On cherche à écrire une fonction `miroir : 'a array -> unit` qui **renverse l'ordre des éléments d'un tableau**. Un élève propose le code suivant :

**OCaml**

```
let miroir tableau =
 let n = Array.length tableau in
 for i=0 to n-1 do
   tableau.(i) <- tableau.(n-i-1)
 done
;;
```

- **Q10.** Essayez ce code sur deux ou trois exemples de taille 5. Qu'en pensez-vous ? Écrire une version qui marche.
  Remarque : si vous ne voyez pas en quoi ça ne marche pas, essayez sur un tableau qui n'a aucun doublon.
- **Q11.** Écrire une fonction `copy : 'a array -> 'a array` qui copie un tableau.
- **Q12.** Tester les lignes suivantes :

```
let t = [|0;1;2;3|];;
let t' = copy t;;
t'.(0) <- 100;;
t.(0)=t'.(0);;
```

  Si la dernière ligne renvoie **`False`**, passez à la suite. Sinon votre fonction est incorrecte, retournez à la question précédente.

## 4. Quelques exercices sur les matrices

On cherche à créer notre propre version de la fonction **`make_matrix : int -> int -> 'a -> 'a array array`** qui construit une matrice de la taille donnée en entrée.

**OCaml**

```
let make_matrix nb_lignes nb_colonnes val_init =
 Array.make nb_lignes (Array.make nb_colonnes val_init)
;;
```

- **Q13.** Tester

```
let m = make_matrix 3 3 0;;
m.(0).(0) <- 3;;
m;;
```

- **Q14.** Pourquoi la fonction est-elle incorrecte ? Écrire une version correcte.
- **Q15.** Écrire une fonction `make_tenseur: int -> int -> int-> 'a -> 'a array array array` qui initialise un tableau à 3 dimensions.
- **Q16.** Écrire une fonction `dimensions : 'a array array -> (int*int)` qui calcule le couple $(n, p)$ des dimensions d'un matrice et qui lève une exception si le tableau de tableaux n'est pas bien formé, c'est à dire s'il ne représente pas une matrice.
  Un exemple de tableau mal formé est **`[|[|0;1|];[|2;3;4|]|]`** où la première ligne a 2 colonnes et la deuxième ligne a 3 colonnes.
- **Q17.** Écrire une fonction `transposee : 'a array array -> 'a array array` qui calcule la transposée d'une matrice carrée. C'est à dire qu'elle renvoie une nouvelle matrice qui est la transposée de celle d'entrée.
- **Q18.** Écrire une fonction `transpose : 'a array array -> unit` qui transpose une matrice carrée, c'est à dire la modifie par effet de bord pour qu'elle devienne sa transposée.

## 5. Jouons au morpion

Dans cette section on va programmer un petit jeu de morpion dans le terminal, pour deux joueurs humains.

Pour cette section vous pouvez utiliser l'interpréteur pour débugguer mais vous ne pourrez le tester qu'en compilant.

On définit le type suivant :

```
type case =
| Vide
| Rond
| Croix
```

On représente le plateau du jeu de morpion par une matrice 3*3 de type **`case array array`**.

On rappelle qu'on peut lire une entrée donnée au clavier par l'utilisateur grâce à **`read_line()`**. La fonction **`read_line()`** renvoie toujours une chaine de caractères.

- **Q19.** Écrire une fonction qui étant donné un plateau de jeu, l'affiche joliment dans le terminal. Par exemple :


- **Q20.** Écrire une fonction qui crée un plateau vide
- **Q21.** Écrire une fonction `modifie : carte array array -> carte -> int -> int -> unit` telle que **`modifie plateau c x y`** met un jeton du type **`c`** aux coordonnées **`x,y`** du plateau. Si les deux entiers ne sont pas des coordonnées valides ou si la case n'est pas vide, on levera une exception **`Coordonees`**.
- **Q22.** Écrire une fonction `tour : carte array array -> carte -> unit` qui prend en entrée l'état actuel du plateau et **`Rond`** ou **`Croix`** selon quel joueur doit jouer. Cette fonction demande au joueur correspondant de rentrer deux entiers pour les coordonnées où il veut jouer et utilise la fonction `modifie`.

  Si l'exception **`Coordonees`** est levée, on demandera au joueur de rentrer à nouveau deux coordonnées.
- **Q23.** Écrire une fonction `fini : carte array array -> carte` qui prend en entrée un plateau et détermine si le jeu est fini, c'est à dire si 3 jetons du même type sont alignés dans le plateau. Elle renvoie **`Vide`** si le jeu n'est pas fini et sinon **`Rond`** ou **`Croix`** selon quel joueur a gagné.
- **Q24.** Écrire une fonction `morpion : unit -> unit` qui permet à deux joueurs de jouer au morpion et affiche le gagnant à la fin.

## 6. Lecture/écriture de fichiers

- **Q25.** On vous donne un fichier **`a_sommer.txt`** de valeurs séparées par des virgules. Calculer la somme des valeurs.

Pour lire tout le fichier on pourra utiliser la structure de code suivante :

```
(*ouvrir le fichier et initialiser des variables*)
...

let continuer = ref true in
while !continuer do
 try
   let line = input_line le_fichier in
   ... (*Faire des choses*)
 with (*S'il n'y a plus rien a lire, une exception est levée, on la rattrape*)
   |End_of_file -> continuer := false (*On change cette variable pour finir la boucle*)
done;;
```

Si vous avez terminé écrivez le tri d'un tableau par insertion, par sélection et par le tri à bulles.