Programmation fonctionnelle
TP n° 4 : Programmer de manière impérative

Exercice 1 (Mélange)

1. Sans regarder le cours, redéfinissez la fonction `swap` telle que `swap tab i j` échange les valeurs du tableau `tab` aux indices `i` et `j`.
2. Pour mélanger un tableau, il suffit d'échanger un nombre suffisant de fois (par exemple 10 fois la taille du tableau) des valeurs à des indices aléatoires. Définissez une fonction `shuffle` qui réalise le mélange d'un tableau.

Exercice 2 (Matrices)

On représente une matrice de dimension $m \times n$ par un tableau de longueur m dont les éléments sont des tableaux de longueur n. Pour accéder à l'élément de la i^{e} ligne et j^{e} colonne d'une matrice représentée par le tableau `mat`, on écrit donc `mat.(i).(j)`.

1. Définissez une fonction `dimension` qui renvoie la dimension d'une matrice sous la forme d'un couple.
2. On cherche à définir une fonction `make_matrix` telle que `make_matrix m n e` renvoie une matrice $m \times n$ dont tous les éléments sont égaux à `e`. Quel est le problème de l'implémentation suivante ?

   ```
   let make_matrix m n e = Array.make m (Array.make n e)
   ```

   Si vous ne voyez pas, interprétez le code suivant au top-level.

   ```
   let m = make_matrix 2 2 0;;
   m.(0).(0) <- 1;;
   m;;
   ```

3. Donnez une implémentation correcte de la fonction `make_matrix`.
4. Définissez des opérateurs `+~` et `*~` qui réalisent la somme et le produit de deux matrices.

Exercice 3 (Files)

Une file est une structure de donnée FIFO (*first in, first out*). On peut ajouter des éléments en fin de file ou au contraire sortir les éléments du début de la file. On se donne ici une représentation mutable d'une file d'attente.

Une file est un chaînage composé de cellules. Une cellule stocke une valeur et connaît la cellule qui la suit. La file elle-même connaît la première et la dernière cellule de la chaîne.

```
type 'a cell = {
  content : 'a;
  mutable next : 'a cell option;
}

type 'a queue = {
  mutable first : 'a cell option;
  mutable last : 'a cell option;
}
```

On voit que les champs `next`, `first` et `last` sont à la fois mutables et optionnels. En effet, la dernière cellule du chaînage n'a pas de suivant et une file vide n'a ni première ni dernière cellule (elle n'en a pas du tout). Par contre, ceci peut évoluer lors de l'ajout ou du retrait de nouveaux éléments.

1. Définissez une fonction sans argument `make_queue` qui crée une nouvelle file vide.
2. Définissez une fonction `is_empty_queue` qui indique si une file est vide.
3. Définissez une fonction `enqueue` telle que `enqueue q e` ajoute `e` à la file `q`.
4. Définissez une fonction `dequeue` telle que `dequeue q` retire et renvoie le premier élément de la file `q`.
5. Définissez une fonction `queue_to_list` qui renvoie une liste contenant les mêmes éléments et dans le même ordre que la file passée en paramètre. Après l'appel à cette fonction, la file est vide.
6. Définissez une fonction `list_to_queue` qui renvoie une file contenant les mêmes éléments et dans le même ordre que la liste passée en paramètre.

Exercice 4 (Générateurs)

Un générateur est une fonction sans argument qui renvoie une nouvelle valeur à chaque appel. On cherche à créer des générateurs qui renvoient des identifiants uniques. Un identifiant est une chaîne de caractères composée d'un préfixe fixe et suivi d'un nombre qui augmente de 1 à chaque appel du générateur.

Définissez une fonction `make_generator` qui prend en paramètre la partie fixe de l'identifiant et qui renvoie le générateur associé.

Exemple d'utilisation :

```
let g1 = make_generator "foo";;
let g2 = make_generator "bar";;
g1 ();; (* renvoie "foo0" *)
g1 ();; (* renvoie "foo1" *)
g2 ();; (* renvoie "bar0" *)
g2 ();; (* renvoie "bar1" *)
g1 ();; (* renvoie "foo2" *)
```

Indication : Clôture lexicale.