TP n° 6 : Tableaux

On rappelle qu'en CAML un tableau, appelé aussi *vecteur*, est une collection modifiable d'éléments de même type, est de type `'a vect` et que sa valeur est représentée entre crochets `[|...|]`.

- le tableau vide s'obtient avec `[||]` :

```
[||];;
- : 'a vect = [||]
```

- pour les petits tableaux, on peut directement rentrer les éléments à la main :

```
[|1; 2; 42|];;
- : int vect = [|1; 2; 42|]
```

- la fonction `make_vect : int -> 'a -> 'a vect` permet de construire et initialiser un tableau :

```
let love_caml = make_vect 5 "Caml";;
love_caml : string vect = [|"Caml"; "Caml"; "Caml"; "Caml"; "Caml"|]
```

- on peut accéder à la longueur en temps constant à l'aide de la fonction `vect_length` :

```
vect_length love_caml;;
- : int = 5
```

- l'accès à l'élément `i` se fait par `tableau.(i)` :

```
love_caml.(0);;
- : string = "Caml"
```

- la modification se fait par l'opérateur `<-` :

```
love_caml.(1) <- "Python";;
- : unit = ()

love_caml;;
 - : string vect = [|"Caml"; "Python"; "Caml"; "Caml"; "Caml"|]
```

- on peut copier un tableau avec la fonction `copy_vect : 'a vect -> 'a vect`.

1 Quelques exercices sur les vecteurs

EXERCICE 1

1. Écrire une fonction `init_vect : int -> (int -> 'a) -> 'a vect` tel que `init_vect n f` initialise un tableau de taille `n` tel que la case d'indice `i` est `(f i)`.
2. En déduire une fonction `range : int -> int vect` « équivalente » à ce que donne `list(range(n))` en PYTHON.

EXERCICE 2 *Encore et toujours Fibonacci*

Écrire une fonction `fibonacci_tab : int -> int vect` renvoyant le tableau des n premiers termes de la suite de Fibonacci.

EXERCICE 3 *Map version tableau*

Deviner ce que doit faire une fonction `map_vect : ('a -> 'b) -> 'a vect -> 'b vect` et l'écrire.

EXERCICE 4 *Miroir d'un tableau*

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

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

1. Qu'en pensez-vous ?
2. Implémenter une version correcte.

EXERCICE 5 *Des listes aux tableaux*

Écrire les fonctions `list_of_vect` et `vect_of_list` qui permettent de passer d'un tableau à une liste et inversement. On n'utilisera pas de références de listes.

2 Quelques exercices sur les matrices

EXERCICE 6 *Création d'une matrice*

On souhaite créer notre propre version de la fonction intégrée `make_matrix`. Un élève propose la solution suivante :

```
let make_matrix_perso nb_lignes nb_cols val_init =
  make_vect nb_lignes (make_vect nb_cols val_int)
;;
```

1. Qu'en pensez-vous ? Justifier.
2. Écrire votre propre version de `make_matrix_perso`.

Sauf mention explicite du contraire, la fonction `make_matrix : int -> int -> 'a -> 'a vect vect` est à votre disposition en CAML.

3. Écrire une fonction `make_tenseur : int -> int -> int -> 'a -> 'a vect vect vect` qui créée un vecteur à trois dimensions.

EXERCICE 7 *Copie d'une matrice*

On souhaite écrire une fonction `copy_matrix : 'a vect vect -> 'a vect vect` qui copie une matrice. Un élève propose :

```
let copy_matrix m = copy_vect m;;
```

1. Qu'en pensez-vous ? Justifier.
2. Écrire correctement la fonction `copy_matrix`.

EXERCICE 8 *Dimension d'une matrice*

Écrire une fonction `dimensions : 'a vect vect -> (int * int)` calculant le coupe (n, p) des dimensions d'une matrice et renvoyant une erreur si le tableau de tableaux n'est pas bien formé, *i.e.* ne représente pas une matrice.

EXERCICE 9 *Addition de deux matrices*

Écrire une fonction `add_matrix : float vect vect -> float vect vect -> float vect vect` réalisant la somme de deux matrices.

EXERCICE 10 *Transposée d'une matrice*

Écrire une fonction `transposee : 'a vect vect -> 'a vect vect` calculant la transposée d'une matrice carrée. Écrire une fonction `transposer : 'a vect vect -> unit` qui transpose une matrice carrée. Avez-vous compris la différence ?

3 Autour des permutations

Une permutation σ de $[\![0, n-1]\!]$ est une bijection de cet ensemble dans lui-même. Nous représenterons une permutation σ de $[\![0, n-1]\!]$ par un tableau `sigma` de taille n tel que pour tout $i \in [\![0, n-1]\!]$, l'entier `sigma.(i)` soit égal à $\sigma(i)$.

QUESTION 1 Écrire une fonction `est_permutation : int vect -> bool` qui teste si un tableau représente une permutation. On cherchera une solution de complexité linéaire.

Le support d'une permutation σ est l'ensemble des points non fixes de σ, *i.e.* l'ensemble des i tels que $\sigma(i) \neq i$.

QUESTION 2 Écrire une fonction `support : int vect -> int list` qui calcule le support d'une permutation sous forme d'une liste d'entiers.

La composée de deux permutations σ et σ' de $[\![0, n-1]\!]$ est l'application $\sigma \circ \sigma'$, au sens habituel des fonctions. On vérifie facilement que la composée de deux permutations de $[\![0, n-1]\!]$ est une permutation de $[\![0, n-1]\!]$.

QUESTION 3 Écrire une fonction `compose : int vect -> int vect -> int vect` qui compose deux permutations.

QUESTION 4 Écrire une fonction `inverse : int vect -> int vect` qui calcule l'inverse d'une permutation, c'est-à-dire la fonction de $[\![0, n-1]\!]$ réciproque.