TD #9 — Tableaux en OCaml, en C (avec solutions)

Exercice 1. *Échauffement sur des* `'a array`*s*

1. Écrivez une fonction OCaml :
   `rev_array : 'a array -> 'a array`
   telle que `rev_array a` s'évalue en un `'a array` nouvellement construit dont les éléments sont ceux de `a` « en sens inverse ».
   Vous *devez* pour cela utiliser `Array.init`

   On propose :
   ```
   let rev_array a =
       let n = Array.length a in
       Array.init n (fun i -> a.(n - 1 - i))
   ```

2. Commentez l'intérêt d'une telle fonction, notamment en comparaison avec son homologue sur les `'a list`.

   L'intérêt semble assez limité par rapport aux `'a list` : de par les possibilités d'accès aléatoire aux éléments d'un `'a array`, il est facile de lire ceux-ci « en sens inverse » si besoin. Il est aussi probablement plus facile de les écrire dans « le bon ordre » dès le début.

3. Écrivez une fonction OCaml :
   `is_palin : 'a array -> bool`
   qui s'évalue à `true` (resp. `false`) si son argument représente (resp. ne représente pas) un palindrome. Cette fonction *devra* utiliser `Array.for_all` (même si cela est un peu sous-optimal et en fait pas si pratique & joli).

   On propose :
   ```
   let is_palin a =
       let n = Array.length a in
       let i = ref (- 1) in
       Array.for_all (fun x -> i := !i + 1 ; x = a.(n - 1 - !i)) a
   ```

4. Écrivez une nouvelle version de votre fonction qui utilise toujours `Array.for_all` mais ne fait pas de vérifications inutiles. (Le résultat n'est pas forcément beau ; en tout cas le mien est assez moche.)

   On propose deux variantes ; la première utilise une exception *ad hoc* pour explicitement interrompre l'exécution d'`Array.for_all` quand on sait qu'elle ne peut plus s'évaluer qu'à `true`, et la seconde exploite le fait que cette fonction est paresseuse et termine dès qu'un élément du tableau ne satisfait plus le prédicat.
   ```
   let is_palin' a =
       let exception Okay in
       let n = Array.length a in
       let i = ref (- 1) in
       let check x =
           if !i >= n / 2 then raise Okay else
           (i := !i + 1 ; x = a.(n - 1 - !i))
       in
       try Array.for_all check a with Okay -> true

   let is_palin'' a =
       let n = Array.length a in
       let i = ref (- 1) in
       let check x =
           if !i >= n / 2 then false else
           (i := !i + 1 ; x = a.(n - 1 - !i))
       in
       Array.for_all check a || !i >= n / 2
   ```

Exercice 2. *Tableaux de tableaux* a la mano

On souhaite construire des tableaux à deux dimensions non rectangulaires : un tel tableau peut se représenter par un tableau de tableaux `a`, c'est à dire un tableau dont les éléments sont eux-mêmes des tableaux. On adopte pour l'instant une représentation *en ligne* (« *row major* ») : l'indice `i` de `a` contient le tableau de tous les éléments de la *i*ème « ligne » du tableau.

1. Écrivez une fonction OCaml :

   ```
   make_2D : int -> int list -> 'a -> 'a array array
   ```

   telle que `make2D nrows rowdims e` s'évalue en un `'a array array` nouvellement créé dont tous les éléments sont initialisés à `e`, et possédant `nrows` lignes dont les dimensions (le nombre d'éléments) est donné par `rowdims` (dans « le même ordre »).
   Par exemple, on a :

   ```
   utop # make_2D 3 [0;1;2] ();;
   - : unit array array = [|[||]; [|()|]; [|(); ()|]|]
   ```

   Vous *pouvez* par exemple utiliser `List.iteri` si vous le souhaitez.

   On propose deux versions, avec ou sans `List.iteri` :

   ```
   let make_2D nrows rowdims e =
       let rd = ref rowdims in
       let make_row _
           = match !rd with
           | [] -> failwith "nrows / rowdims mismatch"
           | x::xs -> rd := xs ; Array.make x e
       in
       Array.init nrows make_row

   let make_2D' nrows rowdims e =
       let a = Array.make nrows [||] in
       List.iteri
           (fun i j -> a.(i) <- Array.make i e)
           rowdims
       ; a
   ```

2. Écrivez une fonction C de signature :

   ```
   int **make_2D(size_t nrows, size_t rowdims[nrows], int e)
   ```

   qui fait de même *mutatis mutandis* pour un tableau de tableau d'`int` (de durée de stockage « allouée ») représenté par un pointeur de pointeur.
   On pourra supposer que toutes les allocations mémoire se déroulent avec succès (que faudrait-il faire sinon ?).

   On propose :

   ```
   int **make_2D(size_t nrows, size_t rowdims[nrows], int e)
   {
       int **a = malloc(nrows * sizeof(int *));
       for (size_t i = 0; i < nrows; i++)
       {
           a[i] = malloc(rowdims[i] * sizeof(int));
           for (size_t j = 0; j < rowdims[i]; j++)
           {
               a[i][j] = e;
           }
       }

       return a;
   }
   ```

   Si les allocations mémoire n'étaient pas toutes garanties de se dérouler avec succès, il faudrait faire échouer la fonction en cas d'erreur (par exemple en renvoyant un pointeur de valeur nulle), tout en libérant la mémoire éventuellement déjà allouée.

3. Quelle différence y a-t'il entre les objets créés par les fonctions OCaml & C ci-dessus, en terme de métadonnées ? Qu'est-ce que cela implique pour leurs utilisations respectives ?

   Le `'a array array` construit par la fonction OCaml « contient » toutes les informations sur ses dimensions (nombre de lignes et longueur de chaque ligne), ce qui n'est pas le cas pour le `int **` renvoyé par la fonction C. Il est donc nécessaire pour l'utilisation de ce dernier de fournir cette information séparément.

4. Proposez une fonction C `free_2D` permettant de libérer un objet créé par `make_2D`.

   Ici, seule la connaissance du nombre de lignes est nécessaire. On propose :

```c
void free_2D(size_t nrows, int *a[nrows])
{
    for (size_t i = 0; i < nrows; i++)
    {
        free(a[i]);
    }
    free(a);

    return;
}
```

Exercice 3. *Linéarisation de tableaux multidimensionnels*

Dans cet exercice, on souhaite représenter un tableau multidimensionnel (disons à deux dimensions) par un tableau à une seule dimension. On suppose plus précisément que le tableau à représenter est rectangulaire, de n lignes et m colonnes. Dans une représentation linéarisée « en ligne » (*row major*) d'un tel tableau, les m premiers éléments sont ceux de la première ligne, les m suivants ceux de la seconde, etc.
Par exemple, le tableau à deux dimensions (ici un `int array array`) `[|[|0; 1|]; [|2; 3|]|]` est linéarisé (« en ligne ») en `[|0; 1; 2; 3|]`
On définit le type :

```c
struct lin_2d_array
{
    size_t size;
    size_t colsize;
    int *dat;
};
```

dans le but de manipuler en C de tels tableaux linéarisés.

1. Écrivez une fonction C de signature :
   `int get(struct lin_2d_array arr, size_t i, size_t j)`
   qui renvoie la valeur de l'élément (i, j) du tableau à deux dimensions représenté (de façon linéarisée) par `arr`.

   On propose :

```c
int get(struct lin_2d_array arr, size_t i, size_t j)
{
    return arr.dat[i*arr.colsize + j];
}
```

2. De même pour une fonction `set` de signature & spécifications adaptées.

   On propose :

```c
void set(struct lin_2d_array arr, size_t i, size_t j, int v)
{
    arr.dat[i*arr.colsize + j] = v;
}
```

Une représentation linéarisée « en colonne » (*column major*) est similaire à une représentation linéarisée « en ligne », où l'on a simplement que les n premiers éléments sont ceux de la première colonne etc. On peut remarquer que calculer une représentation en colonne d'un tableau à partir de sa représentation en ligne est équivalent à représenter sa transposée en ligne.

3. Écrivez une fonction C de signature :
`struct lin_2d_array transpo(struct lin_2d_array arr)`
réalisant cette opération.

On propose :

```c
struct lin_2d_array transpo(struct lin_2d_array arr)
{
    struct lin_2d_array arrt;
    arrt.size = arr.size;
    arrt.dat = malloc(arrt.size * sizeof(int));
    arrt.colsize = arrt.size / arr.colsize;

    for (size_t i = 0; i < arr.colsize; i++)
    {
        for (size_t j = 0; j < arrt.colsize; j++)
        {
            // arrt.dat[i * arrt.colsize + j] = arr.dat[j * arr.colsize + i];
            set(arrt, i, j, get(arr, j, i));
        }
    }

    return arrt;
}
```