L'objectif de cet exercice était de vous faire manipuler des tableaux, et notamment d'observer les différences entre les tableaux et les listes. Cependant, l'exercice contenait quelques pièges et je dois admettre qu'il est un peu difficile pour un premier exercice. J'ai notamment oublié de vous parler des fonctions d'entrées/sorties afin que vous puissiez tester vos fonctions.

1 Entrées/sorties en Ocaml

Contrairement à l'interpréteur, lorsque vous écrivez un programme en Ocaml puis que vous le compilez, afin de voir quelque chose sur la console, il vous faut écrire une fonction qui affiche explicitement quelque chose à l'écran.

La *sortie standard* est un terme pour désigner ce vers quoi vous voulez afficher. En Ocaml, la variable qui permet d'afficher sur la sortie standard s'appelle `stdout` (comme *standard output*. Lorsque vous lancez votre programme depuis le terminal, la sortie standard correspond au terminal. Donc afficher du texte sur la sortie standard revient à écrire du texte dans le terminal. De façon symmétrique, on parle d'*entrée standard* pour parler de ce que votre programme peut récupérer en entrée depuis *l'extérieur*. Évidemment, l'éxtérieur peut désigner beaucoup de chose, même si dans notre cas ce sera principalement des valeurs saisies au clavier depuis le terminal. Vous verrez dans le cours d'*Architectures et systèmes* d'autres façon de récupérer des valeurs du monde extérieur.

Le module qui permet d'afficher du texte en Ocaml s'appelle `Printf`. C'est celui que nous avons utilisé lorsqu'on a affiché notre premier texte : "hello world". La fonction que l'on a utilisé s'appelle aussi `printf`. Notez bien la différence entre le nom de module qui commence par une majuscule et celui de la fonction qui commence par une minuscule.[1]

Le module `Printf` déclare plusieurs fonctions dans le but est toujours le même : afficher du texte!

Dans notre cas, les fonctions qui nous intéressent sont `fprintf`, `printf` et `sprintf`. Les deux dernières étant des cas particuliers de la première.

Avec `fprintf` vous pouvez choisir le *channel* de sortie. On a vu qu'il existait un channel particulier qui s'appelle la sortie standard, mais il en existe d'autre, par exemple la sortie d'erreur (`stderr`). `printf` est un cas particulier de `fprintf` où le channel est `stdout`. `sprintf` est un autre cas particulier où le *channel* est finalement une autre chaîne de caractère. Ainsi, les trois instructions suivantes produisent la même chose :

```
let _ = Printf.fprintf stdout ``Hello world!\n''

let _ = Printf.printf ``Hello world!\n''
```

[1] D'ailleurs il y a une différence sémantique entre les noms qui commencent par une majuscule et ceux qui commencent par une minuscules. Vos noms de **variables/fonctions** (même si sensiblement la même chose en Ocaml) doivent toujours commencer par une **miniscule**. Les noms commençant par une majuscule sont réservés pour les constructeurs de type ainsi que pour les noms de module

```
let _ = Printf.printf ``%s" (Printf.sprintf ``Hello world!\n")
```

Le dernier exemple introduit la fonctionnalité majeur des fonctions d'affichage d'Ocaml que sont les *format*.

Si vous regardez le type de `printf` dans la doc, vous pouvez voir que son type est :

```
val printf : ('a, out_channel, unit) format -> 'a
```

Les formats sont des types hautement polymorphes. Cependant pour le moment, on considerera que les formats ne sont ni plus ni moins des chaînes de caractères dans lesquelles ont peut rajouter des *arguments* sous forme de format justement. Un format commence toujours par le symbole "%" et est suivi d'une lettre, cette dernière indiquant le **type** de l'argument attendu.

Dans l'exemple avec `sprintf`, la lettre qui suit le "%" est un `s` ce qui veut dire que l'argument attendu est de type `string`. Les différents formats possibles sont décrit dans la documentation du module `Printf` mais voici les plus utiles :

- `%d` pour un entier
- `%s` pour une chaîne
- `%f` pour un nombre floattant
- `%b` pour un booléen

Le truc chouette avec les format, c'est qu'on peut les enchaîner. Ainsi, on peut écrire

```
let _ = Printf.printf ``%d + %d <> %f" 42 42 85
```

Notez bien qu'après la chaîne de caractère formatée, il y a autant d'argument que de format dans la chaîne.

Les saisies utilisateur fonctionnent de façon similaire sauf que l'argument attendu est une fonction qui qui est appliquée à la valeur lue. Ces fonctions se trouvent dans le module `Scanf`. Par exemple

```
let _ = Scanf.scanf ``%d'' (fun x -> Printf.printf ``%d'' x)
```

affiche l'entier entrée par l'utilisateur.

2 Afficher vos types

Un des problèmes qui a été soulevé lors de cette séance, c'est comment afficher un tableau ? Les formats ne sont utilisés que pour les types simples (`int`, `bool`, `string`, ...). Si vous voulez afficher des objets plus compliqués comme des listes ou des tableaux, c'est à vous de le faire !

Pour cela, un des idiomes du langage c'est de créer des fonctions

```
val string_of_mytype : mytype -> string
```

Ensuite pour afficher un objet de votre type vous pouvez faire

```
let _ = Printf.printf ``%s\n" (string_of_mytype myobject)
```

Dans notre cas, ce qui nous intéressait, c'était de créer une fonction `string_of_array`. Certains parmi vous ont tenté une implémentation qui donnait à la fonction `string_of_array` le type :

```
val string_of_array : int array -> string
```

ce qui veut dire que votre fonction ne fonctionne que pour des tableaux d'entiers. Or, c'est plutôt problématique puisque les tableaux étant un type polymorphe, on voudrait pouvoir réutiliser notre fonction pour différent type de tableaux. Pour cela on a pas le choix, il faut que la fonction prenne un autre argument en entrée qui dise comment afficher des éléments de type α. Le type de `string_of_array` devient alors

```
val string_of_array : ('a -> string) -> 'a array -> string
```

Une implémentation possible pour `string_of_array` est la suivante :

```
let string_of_array f a =
  let (pre,sep,post) = ("[|",",","|]") in
  let s = Array.fold_left (fun s e -> s^(f e)^sep) pre a in
  let s' = if Array.length a <> 0 then
             String.sub s 0 (String.length s - 1)
           else
             s
  in
  s'^post
```

3 Correction de l'exercice 1

3.1 Question 1

La première question vous demandait d'implémenter une fonction `array_of_list` de type

```
val array_of_list : 'a list -> 'a arrow
```

qui convertissait une liste en un tableau. Cette première question comportait un premier piège à savoir, que se passe-t-il si ma liste est vide ? Vous voulez retournez un tableau vide qui soit polymorphe. Or, la fonction `make` proposée par la librairie standard ne vous propose pas de créer un tel tableau. En effet, le type de l'expression

```
Array.make 0 0
```

est `int array`. D'ailleurs, vous pouvez tester

```
let _ = Printf.printf "%b\n" (Array.make 0 0 = Array.make 0 "")
```

mais le booléen retourné par Ocaml est `false`. Pour cela, je rajoutais une indication dans le TP qui vous indiquait de façon un peu *magique* que la syntaxe

```
[||]
```

construit un tableau vide polymorphe.

Le problème étant reglé pour la liste vide, il reste à traiter le cas récursif. Cette fois, sachant que la liste n'est pas vide, on peut initialisé le tableau avec le premier élément de la liste. Voici un exemple d'implémentation d'`array_of_list`:

```
let array_of_list l =
  match l with
    | [] -> [||]
    | x::t ->
       let a = Array.make (List.length l) x in
       let rec fill i l =
         match l with
         | [] -> a
         | x::t -> Array.set a i x; fill (i+1) t
       in
       fill 1 t
```

ce qui peut aussi s'écrire en utilisant la fonction `iteri` du module `List` :

```
let array_of_list l =
  match l with
    | [] -> [||]
    | x::t ->
       let a = Array.make (List.length l) x in
       List.iteri (fun i e -> a.(i) <- e) l; a
```

Cette version utilise de la récursivité. Je vous laisse le soin de le refaire avec une boucle `for` par exemple.

3.2 Question 3

Fort heureusement, il se trouve que cette question est beaucoup plus simple que la précédente. Pour ma part, j'ai opté pour une solution utilisant `fold_left` :

```
let list_of_array a = Array.fold_left (fun l x -> x::l) [] a
```

3.3 Question 5 & Question 6

La question 5 de prime abord ne semble pas très compliqué. Il suffit de créer un tableau dont tous les éléments seront eux aussi des tableaux. Cela peut se faire ainsi :

```
let make_matrix n m =
  Array.make n (Array.make m 0)
```

Mais si vous faites la question 6, vous allez vous rendre compte que cette solution ne fonctionne pas. En effet, créons une matrice de taille 4×4 et affichons-là :

```
let m = make_matrix 4 4

let _ = m.(2).(1) <- 1

let _ = Printf.printf "%s\n" (string_of_array (string_of_array string_of_int) m)
```

Le résultat affiché par la console est :

```
[|[|0,1,0,0|],[|0,1,0,0|],[|0,1,0,0|],[|0,1,0,0|]|]
```

C'est un peu décevant n'est-ce pas ? Le soucis vient du fait que le tableau `Array.make m 0` se retrouve *dupliqué*. L'explication de ce phénomène, vous le comprendrez plus aisément lors des TP de C que vous ferez avec Lucca. Grosso-modo, contrairement à une liste, un tableau est assigné à une adresse particulière de la mémoire ce qui fait que la fonction `make_matrix` ci-dessus est équivalente à

```
let make_matrix n m =
  let tmp = Array.make m 0 in
  Array.make n tmp
```

Ca veut dire que le tableau *extérieur* est initialisé n fois avec la même adresse qui contient un tableau de taille m initialisé à 0. Le schéma de gauche de la figure 1 montre ce qui se passe réellement en mémoire. Les flèches sont des *pointeurs* : chaque case du tableau extérieur contient une adresse. La flèche montre à quoi correspond cette adresse.

Pour éviter ce problème, il faut tricher un peu de telle sorte à ce qu'Ocaml génère des adresse différentes à chaque fois. Pour cela, il faut se débrouiller pour que `Array.make m 0` soit réévalué pour chaque case du tableau. Heureusement, une fonction de la librairie standard va nous sauver : la fonction `init`. Cette fonction est similaire a `make` sauf qu'on peut paramétriser la valeur par défaut par un indice. Ainsi une correction consiste à faire

```
let make_matrix n m =
  Array.init n (fun _ -> Array.make m 0)
```

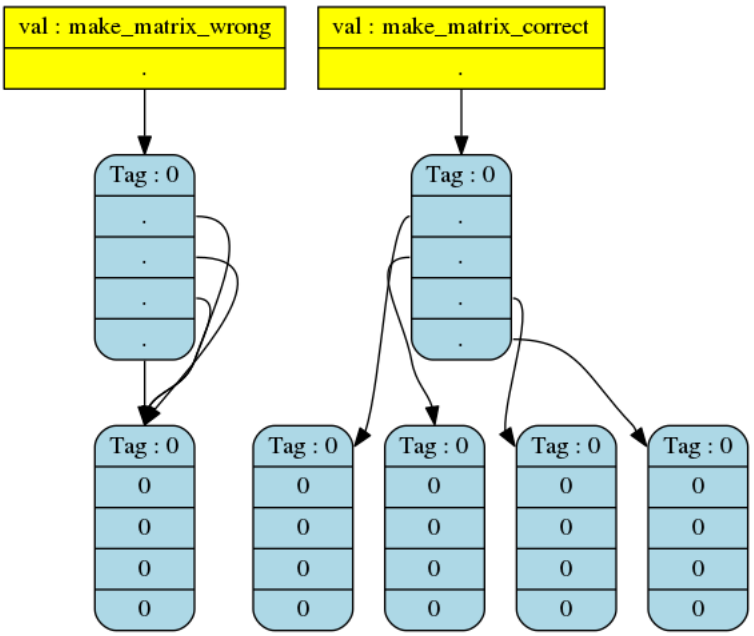

Figure 1: représentation mémoire des tableaux

Le schéma de droite montrent la représentation mémoire du tableau obtenu. Pourquoi ça fonctionne cette fois ? Cela se repose sur le fait que même lorsqu'une fonction (ici `fun _ -> Array.make m 0`) est passée en argument d'une autre fonction, le corps de cette fonction (`Array.make m 0`) n'est pas évalué directement. Il sera évalué seulement lorsque la fonction sera appliquée a un argument. Comme la fonction est appelé n fois avec les indices $0, 1, \ldots, n-1$, son corps est aussi évalué n fois. Et donc n adresse différentes sont générées.

Ce phénomène arrive tout le temps lorsque l'on fait de la programmation *impérative*, ce qui est typiquement le cas avec les *array* d'Ocaml. Avec les listes, vous n'aurez jamais ce genre de problème.