

## LANGAGE OCaml – TP 3

### Tableaux et chaînes de caractères

#### Introduction

En OCaml, comme dans la plupart des langages de programmation, les types structurés usuels tels que les tableaux, les enregistrements ou les listes sont présents. Cependant, contrairement au langage C, l'allocation/libération mémoire est gérée par un ramasse-miettes (ou glaneur de cellules, ou garbage collector) : le langage OCaml est un langage de haut niveau. L'efficacité de cette allocation mémoire est supérieure à celle du malloc en C.

#### Chaînes de caractères

##### Caractères

En Caml, les caractères sont de type `char` et sont spécifiées entre des apostrophes. Elles peuvent être déclarées comme mutables grâce aux références et sont affichées à l'écran grâce à l'instruction `print_char( 'a' )`.

```
let a = 'a';;
a = 'b';;
a := 'b';;
```

```
let c = ref 'a';;
c := 'd';;
print_char (c);;
```

```
print_char (!c);;
```

Sorties écran :

```
# val a : char = 'a'
# - : bool = false
# Characters 1-2:
  a := 'b';;
  ^
```

Error: This expression has type char but an expression was expected of type  
          'a ref

```
# val c : char ref = {contents = 'a'}
# - : unit = ()
```

```
# Characters 12-15:
  print_char (c);;
  ^^^
```

Error: This expression has type char ref  
          but an expression was expected of type char

```
# d- : unit = ()
```

Les `char` sont codés sur 8 bits et regroupent tous les caractères alphanumériques, ainsi que d'autres caractères spéciaux (voir table ASCII en annexe).

Pour passer du caractère à l'entier associé, on utilise la fonction `int_of_char`. Inversement, si on souhaite passer de l'entier à son équivalent en `char`, on utilise la fonction `char_of_int`.

<pre>int_of_char ('c');; int_of_char (!c);; char_of_int (97);;</pre>	<pre># - : int = 99 # - : int = 100 # - : char = 'a'</pre>
<pre>for i = 97 to 101 do   print_char (char_of_int(i)); done;;</pre>	<pre>abcde- : unit = ()</pre>

Les caractères peuvent être comparés entre eux au moyen des opérateurs de comparaison : <, <=, =, >, >=, >. Les entiers associés aux caractères par la table ASCII sont alors comparés.

### Chaînes de caractères

Les chaînes de caractères sont des séquences finies de caractères. Elles sont de type string et sont spécifiées entre guillemets. En Caml, elles sont limitées à  $2^{24} - 6 = 16\,777\,210$  caractères. L'accès à un élément (type char) se fait avec un point suivi d'un indice entre crochets. Les indices sont numérotés à partir de 0. Enfin, elles sont mutables.

<pre>"Bonjour les MP2I";; let chaine = "Bonjour les MP2I";;  chaine;;  print_string (chaine);; (*Les parenthèses sont facultatives Mais aident à la compréhension *)  print_string (chaine.[0]);;  print_char (chaine.[0]);;  chaine.[14]&lt;- 'I';; chaine.[15]&lt;- ' ';; chaine;;  print_char (chaine.[16]);;</pre>	<pre>Sortie écran :  # - : string = "Bonjour les MP2I" # val chaine : string = "Bonjour les MP2I" # - : string = "Bonjour les MP2I" # Bonjour les MP2I- : unit = ()  # Characters 14-24:   print_string chaine.[0];;       ^^^^^^^^^^^ Error: This expression has type char but an expression was expected of type string  # B- : unit = ()  # - : unit = () # - : unit = () # - : string = "Bonjour les MPI "  Exception: Invalid_argument "index out of bounds".</pre>
--	--

Les chaînes de caractères peuvent aussi être déclarées comme référence.

<pre>let s = ref "";;  s := 'a';; (* Attention au type : Un caractère n'est pas une chaîne de caractères comportant un unique caractère *)  s := "a";;  s;;</pre>	<pre># val s : string ref = {contents = ""}  # Characters 6-9:   s := 'a';;     ^^^ Error: This expression has type char but an expression was expected of type string  # - : unit = ()  # - : string ref = {contents = "a"}</pre>
---	--

### Opération sur les chaînes de caractères

<pre>let chaine_1 = "Bon";; let chaine_2 = "jour";; let chaine_3 = chaine_1 ^ chaine_2;;</pre> <p>Sortie écran :</p> <pre># val chaine_1 : string = "Bon" # val chaine_2 : string = "jour" # val chaine_3 : string = "Bonjour"</pre>	<p>La concaténation de deux chaînes de caractères s1 et s2 est l'opération consistant à mettre bout à bout ces deux chaînes.</p> <p>L'opérateur <code>^</code> qui permet d'exprimer la concaténation de deux chaînes de caractères.</p>
<pre>let chaine_1 = "aa";; let chaine_2 = "ab";; chaine_1 = chaine_2;; chaine_1 &lt;&gt; chaine_2;; chaine_1 &lt;= chaine_2;;</pre> <p>Sortie écran :</p> <pre># val chaine_1 : string = "aa" # val chaine_2 : string = "ab" # - : bool = false # - : bool = true # - : bool = true</pre>	<p>Les opérateurs <code>=</code>, <code>&lt;</code>, <code>&lt;=</code>, <code>&gt;</code> et <code>&gt;=</code> permettent de comparer deux chaînes de caractères.</p> <p>Si s1 et s2 sont deux expressions de type string, alors :</p> <ul style="list-style-type: none"> <li>• <code>s1 = s2</code> est vrai si et seulement si les deux chaînes de caractères s1 et s2 sont égales ;</li> <li>• <code>s1 &lt; s2</code> est vrai si et seulement si la chaîne de caractères s1 vient strictement avant s2 dans l'ordre lexicographique ;</li> <li>• <code>s1 &lt;= s2</code> est vrai si et seulement si la chaîne de caractères s1 vient avant s2 dans l'ordre lexicographique ou est égale à s2 ;</li> </ul>

### Module String

Les modules en OCaml commencent par une majuscule. Le module String contient plusieurs fonctions à connaître :

<pre>let chaine = "Bonjour";; let l = String.length (chaine);;</pre> <p>Sortie écran</p> <pre># val chaine : string = "Bonjour" # val l : int = 7:</pre>	Longueur d'une chaîne de caractères.
<pre>let chaine = "Bonjour";; let l = String.length (chaine);; let copie = String.copy chaine;; let bis = copie;; copie.[0] &lt;- 'H';; copie;; bis;; chaine;;</pre> <p>Sortie écran :</p> <pre># val chaine : string = "Bonjour" # val copie : string = "Bonjour" # val bis : string = "Bonjour" # - : unit = () # - : string = "Honjour" # - : string = "Honjour" # - : string = "Bonjour"</pre>	<p>Attention aux problèmes de valeurs partagées.</p> <p>L'utilisation de la fonction <code>String.copy</code> permet de dupliquer une chaîne de caractères en créant une nouvelle variable.</p> <p>Une simple création avec un <code>let</code> ne permet pas l'indépendance des deux variables.</p>

A noter : il n'existe pas de fonction `string_of_char` ou `char_of_string`.

### Tableaux (Array)

Comme en C, un tableau est une structure séquentielle de données de même type. Les indices, numérotés à partir de 0, permettent un accès direct à un élément du tableau. Une fois le tableau créé, il est impossible d'en modifier la taille. Par contre, un tableau est une structure de donnée mutable : la valeur de chacun de ses éléments peut être modifiée. Il est donc inutile d'utiliser le `ref` lors de sa définition.

#### Création d'un tableau

Il existe plusieurs syntaxes permettant de créer un tableau : création directe ou en faisant appel au module `Array`. Lors de sa création, des valeurs doivent impérativement être affectées aux différentes cases du tableau. OCaml n'autorise pas les valeurs qui seraient incomplètes ou mal formées ; toute expression bien typée s'évalue en une valeur de ce type.

Déclaration directe d'un tableau	
<pre>[ 3; 2; 1 ];;</pre>	<p>Définit un tableau de 3 éléments. Le séparateur est le point-virgule.</p> <p>La sortie dans l'interpréteur est alors :</p> <pre># - : int array = [ 3; 2; 1 ]</pre> <p>Le type des éléments contenu dans le tableau est inféré : <code>int</code> dans ce cas.</p>

<pre>[ 3. ; 2; 1 ];;</pre> <pre>[   ] ;;</pre>	<p>Le tableau est de taille 3 et les éléments sont numérotés de 0 à 2. La taille du tableau est définitive mais les éléments pourront être modifiés.</p> <p>Sortie écran :</p> <p>Sortie écran :</p> <pre># - : 'a array = [   ]</pre> <p>Rappel : 'a : type quelconque</p>
Déclaration d'un tableau comme variable	
<pre>let mon_tableau = [ 0;2;4;6;8 ];;</pre> <pre>let tableau_chaine = [ "super" ; "tableau" ];;</pre>	<p>Une variable de type int array est créée.</p> <pre># val mon_tableau : int array = [ 0; 2; 4; 6; 8 ]</pre> <pre># val tableau_chaine : string array = [ "super"; "tableau" ]</pre>

Cette technique peut être utilisée sur des tableaux de petites tailles mais devient laborieuse pour des tableaux de grande taille. Le module Array (en OCaml, les modules commencent par une majuscule) contient de nombreuses fonctions permettant de manipuler les tableaux.

Utilisation du module Array	
<pre>let tableau = Array.make 5 8;;</pre>	<p>Syntaxe : <code>Array.make <i>taille valeur</i> ;;</code>  Création d'un tableau à partir de sa longueur et d'une valeur par défaut.</p> <p>Sortie écran :</p> <pre>val tableau : int array = [ 8; 8; 8; 8; 8 ]</pre> <p>Toutes les cases contiennent la même valeur.</p>
<pre>let carre_0 x = x*x in   Array.init 11 carre_0;;</pre>	<p>La fonction <code>Array.init</code> de type <code>int → (int → 'a) → 'a array</code> permet de créer des tableaux d'une longueur donnée par le premier argument et dont les éléments s'expriment en fonction de l'indice par la fonction donnée par le second argument.</p> <p>Sortie écran :</p> <pre># - : int array = [ 0; 1; 4; 9; 16; 25; 36; 49; 64; 81; 100 ]</pre> <p>En utilisant le <code>Array.init</code>, proposer des instructions permettant d'obtenir les variables et sorties écran ci-</p>

<pre> let carre_1 x = x*x;;  let tableau_1 = Array.init 4 carre_1;;  tableau_1;;  let tableau_2 = let carre x = x*x in Array.init 3 carre;;  tableau_2;; </pre>	<pre> dessous :  # val carre_1 : int -&gt; int = &lt;fun&gt;  # val tableau_1 : int array = [ 0; 1; 4; 9 ]  # - : int array = [ 0; 1; 4; 9 ]  # val tableau_2 : int array = [ 0; 1; 4 ]  # - : int array = [ 0; 1; 4 ] </pre>
---	---

### Accès et modification d'un élément

L'accès à l'élément situé à l'indice i d'un tableau se fait avec la syntaxe suivante :

```
nom_tableau.(indice) ;;
```

De la même manière, l'élément situé à l'indice i du tableau peut être modifié en utilisant la syntaxe :

```
nom_tableau.(indice)<- valeur ;;
```

### Exemple

<pre>let rire = Array.make 7 "ha";;</pre>	<p>Créer le tableau :</p> <pre># val rire : string array = [ "ha"; "ha"; "ha" ; "ha"; "ha"; "ha"; "ha" ]</pre>
<pre>for i=0 to 6 do   print_string rire.(i);   print_newline() done;;</pre>	<p>Instructions permettant l'affichage :</p> <pre># ha ha ha ha ha ha ha</pre> <p>- : unit = () Une boucle for sera utilisée.</p>
<pre>rire.(3)&lt;-"ho";; for i=0 to 6</pre>	<p>Instructions permettant la modification et l'affichage :</p>

<pre>do   print_string rire.(i);   print_newline() done;;</pre>	<pre># ha ha ha ho ha ha ha - : unit = ()</pre>
---	---

Comme n'importe quel type de valeur, un tableau peut être passé en paramètre à une fonction ou retourné.

Analyser et commenter les sorties écran des instructions suivantes :

<pre>let affiche_premier tab =   print_int tab.(0) ;   print_newline () ; in let t1 = [  4; 6; -3  ] in affiche_premier t1 ; let t2 = Array . make 23 (-2) in affiche_premier t2 ;;</pre>	<p>La fonction <code>affiche_premier</code> affiche le premier élément d'un tableau d'entiers. Cette fonction est ensuite appelée avec une tableau t1 dont le premier élément est 4 puis une seconde fois avec t2, dont le premier élément est -2.</p> <pre># 4 -2 - : unit = ()</pre>
---	--

Quel est le type de la fonction `affiche_premier` ?

Cette fonction ne renvoie rien ; son type est donc `unit()`.

Que se passe-t-il si les parenthèses délimitant le -2 sont omises ?

Message d'erreur :

# Characters 138-153:

```
let t2 = Array . make 23 -2 in
      ^^^^^^^^^^^^^^^^^
```

Error: This expression has type 'a -> 'a array  
but an expression was expected of type int

Les tableaux sont des objets mutables : les valeurs qu'ils contiennent peuvent être modifiées (mais pas les types).

Tester les instructions suivantes ; commenter le résultat obtenu.

<pre>let t3 = Array.make 3 2;; t3;; let t4 = t3;; t3.(0)&lt;-0;; t3;; t4;; let a = Array.length t3;; a;;</pre>	<p>Sortie écran :</p> <pre># val t3 : int array = [ 2; 2; 2 ] # - : int array = [ 2; 2; 2 ] # val t4 : int array = [ 2; 2; 2 ] # - : unit = () # - : int array = [ 0; 2; 2 ] # - : int array = [ 0; 2; 2 ] # val a : int = 3 # - : int = 3</pre>
--	--

<pre>(* On dit que les valeurs sont partagées *)  (*      Array.length :      à connaître ! *)</pre>	<p>Rappel :</p> <p>t3 ;; permet l'affichage de la variable</p> <p>Stockage sous la même forme que python ; donc, si on modifie l'un, on modifie l'autre.</p> <p>Array.length permet de récupérer le nombre d'éléments.</p>
<pre>let t5 = Array.copy t4;; t4;; t5;; t4.(1)&lt;-4;; t3;; t4;; t5;;</pre>	<p>Array.copy : copie profonde</p> <pre># val t5 : int array = [ 0; 2; 2 ] # - : int array = [ 0; 2; 2 ] # - : int array = [ 0; 2; 2 ] # - : unit = () (* Affectation *) # - : int array = [ 0; 4; 2 ] (* t3 *) # - : int array = [ 0; 4; 2 ] (* t4 *) # - : int array = [ 0; 2; 2 ] (* t5 *)</pre>

De même, tester et commenter les effets des instructions suivantes :

<pre>let ts1 = Array.make 3 "baba";; let ts2 = Array.make 3 ts1.(0);; ts1;; ts2;; ts2.(0)&lt;-"bibi";; ts1.(0).[3]='a';; ts1.(0).[3]&lt;-'o';; (* Guillemets simples pour un caractère*) (* Guillemets doubles pour une chaîne de caractères *) ts1;; ts2;;</pre>	<p>Sortie écran :</p> <pre># val ts1 : string array = [ "baba"; "baba"; "baba" ] # val ts2 : string array = [ "baba"; "baba"; "baba" ] # - : string array = [ "baba"; "baba"; "baba" ] # - : string array = [ "baba"; "baba"; "baba" ] # - : unit = () # - : bool = true # - : unit = () # - : string array = [ "babo"; "babo"; "babo" ]  # - : string array = [ "bibi"; "babo"; "babo" ]</pre> <p>Attention : le Array.make ne duplique pas la chaîne de caractères ; là encore, les valeurs sont partagées.</p>
---	---

### Comparaison de deux tableaux

Deux tableaux sont égaux s'ils ont même longueur et si pour chaque indice, les éléments correspondants sont égaux. Sur cette question d'égalité, en informatique, il est parfois nécessaire d'être plus précis en distinguant l'égalité logique de l'égalité physique. Deux tableaux sont **logiquement** égaux s'ils sont égaux au sens signifié plus haut (même longueur, et valeurs contenues égales). Ils sont dits **physiquement** égaux s'ils sont tous deux stockés au même endroit en mémoire. L'opérateur usuel = permet de tester l'égalité logique ; l'opérateur == permet de tester l'égalité physique.



Commenter les sorties obtenues

<pre>[ 1; 2; 3 ] = [ 1; 2; 3 ];; [ 1; 2; 3 ] = [ 3; 2; 1 ];;  let t = [ 1; 2; 3 ] in t = Array.copy t;;</pre>	<pre># - : bool = true (* contenus identiques *) # - : bool = false (* contenus différents *)  # - : bool = true (* contenus identiques *)</pre>
<pre>[ 1; 2; 3 ] == [ 1; 2; 3 ];;  let t = [ 1; 2; 3 ] in t == Array.copy t;;  let t = [ 1; 2; 3 ] in let s = t in t == s;;</pre>	<pre># - : bool = false (* Adresses mémoire différentes *)  # - : bool = false (* Adresses mémoire différentes *)  # - : bool = true (* Contenus identiques *)</pre>

### Tableaux en paramètres de fonctions

On souhaite écrire une fonction répondant à l'algorithme suivant :

<p>Algorithme : doublement des valeurs d'un tableau</p> <p>Entrée : tableau t de n entiers</p> <p>Sortie : tableau t' de n entiers valant le double des valeurs du tableau passé en argument</p> <p>Début</p> <p>Pour chaque indice i du tableau faire</p> <p style="padding-left: 40px;">t'[ i ] := 2 x t[ i ]</p> <p>finpour</p> <p>retourner t'</p> <p>fin</p>
---

La spécification du problème et l'algorithme n'indiquent pas s'il faut conserver intact le tableau original et produire un nouveau tableau avec les valeurs doublées, ou bien si le tableau d'origine doit être modifié pour contenir les valeurs doubles de celles de départ.

On peut alors définir les deux fonctions :

<pre>(* fonction double_elements : int array -&gt; int array  parametre t : int array = tableau dont on veut doubler les elts  valeur renvoyee : int array = nouveau tableau dont les elts sont double de ceux de t *)</pre>	Sortie écran :
--	----------------

<pre> let double_elements t = let n = Array.length t in let t1 = Array.make n 0 in for i = 0 to n - 1 do   t1.(i) &lt;- 2 * t.(i) done ; t1 ;; (* Pour renvoyer t1 *)  let tab = Array.make 3 2;; let tab_bis = double_elements tab;; </pre>	<pre> # val double_elements : int array -&gt; int array = &lt;fun&gt;  # val tab : int array = [ 2; 2; 2 ] # val tab_bis : int array = [ 4; 4; 4 ] </pre>
<pre> (* Fonction doubler_elements : int array -&gt; unit  Parametre t : int array = tableau dont on veut doubler les elts  action : doubler les elts de t (t est modifié)  *)  let doubler_elements t = let n = Array.length t in for i = 0 to n - 1 do   t.(i) &lt;- 2 * t.(i) done ;; (* pas de variable renvoyée : passage par adresse du tableau *)  let table = Array.make 3 4;; doubler_elements table;;  table;; </pre>	<pre> # val doubler_elements : int array -&gt; unit = &lt;fun&gt;  # val table : int array = [ 4; 4; 4 ] # - : unit = ()  # - : int array = [ 8; 8; 8 ] </pre>

### Tableau et filtrage

Analyser et commenter la sortie écran des instructions suivantes :

<pre> let t = [ 4;2 ];; match t with   [ _ ] -&gt; "Vide"   [ 1;_ ] -&gt; "Commence par 1"   [ 2;_ ] -&gt; "Commence par 2"   [ x;y ] -&gt; "t commence par " ^ string_of_int(x)   _ -&gt; "Autre cas";; </pre>	<p>Sortie écran :</p> <pre> # - : string = "t commence par 4" </pre>
---	--

Tableaux à plusieurs dimensions

Un tableau à deux dimensions est un tableau dont chaque élément est lui-même un tableau. Il peut être créé directement ou en utilisant les fonctions du module Array. Son type est donc 'a array array.

## Création directe

```
let matrice_1 =
[| [| (0,0);(0,1);(0,2)|];
  [| (1,0);(1,1);(1,2)|];
  [| (2,0);(2,1);(2,2)|];
  [| (3,0);(3,1);(3,2)|] |];;
```

Sortie écran :

```
# val matrice_1 : (int * int) array array =
[| [| (0, 0); (0, 1); (0, 2) |];
  [| (1, 0); (1, 1); (1, 2) |];
  [| (2, 0); (2, 1); (2, 2) |];
  [| (3, 0); (3, 1); (3, 2) |] |]
```

Le tableau matrice\_1 est un tableau à deux dimensions contenant des paires d'éléments de type int.

Les paires sont un type particulier, défini entre parenthèses, permettant de définir un produit cartésien de deux types différents : si x est un élément de type 'a et y un élément de 'b, alors (x, y) est un élément de type 'a \* 'b.

On peut généraliser cette définition pour définir un n-uplet, type permettant de définir un produit cartésien de n variables de types différents (ou non). Ces n-uplets ne sont pas mutables et peuvent être utilisés pour la curryfication des fonctions (voir TP complémentaires sur les fonctions).

```
let triplet_1 = (1,2,"cou");;
triplet_1;;
let a,b,c = triplet_1;;
(*let est destructurant *)
a;;
b;;
c;;
```

Donnera la sortie écran :

```
# val triplet_1 : int * int * string =
(1, 2, "cou")
# - : int * int * string = (1, 2, "cou")
# val a : int = 1
val b : int = 2
val c : string = "cou"
# - : int = 1
# - : int = 2
# - : string = "cou"
```

Comme pour la création des tableaux, la création de matrices de grande taille en utilisant ces méthodes (énumération de tous les éléments, lignes par lignes) peut vite devenir fastidieuse.

On peut alors recourir aux fonctions Array.make et Array.init ou Array.make\_matrix.

## Array.make

```
let mat_1 =
```

Sortie écran :

```
let t = Array.make 4 [| |] in
for i = 0 to 3 do
  let ligne = Array.make 3 (0,0) in
  for j = 0 to 2 do
    ligne.(j) <- (i,j)
  done;
  t.(i) <- ligne
done;
t;;
```

```
val mat_1 : (int * int) array array =
[[|(0, 0); (0, 1); (0, 2)|];
 |(1, 0); (1, 1); (1, 2)|];
 |(2, 0); (2, 1); (2, 2)|];
 |(3, 0); (3, 1); (3, 2)|]]
```

La méthode consiste à construire le tableau qui contiendra les lignes, puis à construire une à une chacune des lignes, et à les affecter au tableau de lignes

### Array.init

```
let mat_2 =
let elt i j = (i, j) in
let ligne i = Array.init 3 (elt i) in
Array.init 4 ligne;;
```

```
# val mat_2 : (int * int) array array =
[[|(0, 0); (0, 1); (0, 2)|]; |(1, 0);
 (1, 1); (1, 2)|];
 |(2, 0); (2, 1); (2, 2)|]; |(3, 0);
 (3, 1); (3, 2)|]]
```

Array.init 4 ligne crée un array de 4 lignes, auxquelles sont associées les indices 0, 1, 2, 3 (voir Array.init). Donc, ligne va être appelée 4 fois et recevoir i valant 0, 1, 2, 3.

Ligne de i est aussi un Array.init, avec comme premier paramètre 3 : une ligne contient 3 éléments, associé chacun aux valeurs 0, 1, 2. Pour créer l'élément, on fait appel à la fonction elt, qui prend 2 entiers en paramètre. Le premier sera le i passé à ligne, puisque l'on a (elt i). Le second, j, sera la valeur associée au Array.init 3, donc, 0, 1 puis 2.

L'expression permettant de construire cette matrice repose sur la définition locale de deux fonctions :

1. La fonction elt, qui construit un couple de deux valeurs
2. La fonction ligne qui construit des tableaux de longueur 3 à l'aide de la fonction elt i par application partielle de la fonction elt.

### Array.make\_matrix

```
let mat_3 =
Array.make_matrix 2 3 (1,1);;

(* make_matrix : pas à connaître par coeur *)
```

Sortie écran :

```
val mat_3 : (int * int) array array =
[[|(1, 1); (1, 1); (1, 1)|];
 |(1, 1) ; (1, 1) ; (1, 1)|]]
```

L'accès à un élément d'un tableau à deux dimensions et sa modification suivent des syntaxes similaires à celles vues pour les tableaux à une dimension :

<code>mat_2.(1);;</code>	Sortie écran : <code># - : (int * int) array = [  (1, 0); (1, 1); (1, 2)  ]</code>
<code>mat_2.(1)&lt;- [  (0, 0); (0, 0); (0, 0)  ];;</code> <code>mat_2.(1);;</code> <code>mat_2;;</code>	Sortie écran : <code># - : unit = ()</code>  <code># - : (int * int) array =</code> <code> [  (0, 0); (0, 0); (0, 0)  ]</code>  <code># - : (int * int) array array =</code> <code> [  [  (0, 0); (0, 1); (0, 2)  ];</code> <code> [  (0, 0); (0, 0); (0, 0)  ];</code> <code> [  (2, 0); (2, 1); (2, 2)  ];</code> <code> [  (3, 0); (3, 1); (3, 2)  ]  ]</code>

Comme pour les tableaux à une dimension, un problème de valeurs partagées peut se poser.

Analyser et expliquer les résultats obtenus.

```
let mat_4 =
let t = Array.make 3 (Array.make 4 (0,0)) in
for i = 0 to 2 do
  for j = 0 to 3 do
    t.(i).(j)<-(i,j)
  done
done;
t;;

mat_4.(0).(0)<-(0,0);;

mat_4;;

mat_4.(0) == mat_4.(1) ;;
```

Sortie écran :

```
# val mat_4 : (int * int) array array =
  [| [| (2, 0); (2, 1); (2, 2); (2, 3) |];
    [| (2, 0); (2, 1); (2, 2); (2, 3) |];
    [| (2, 0); (2, 1); (2, 2); (2, 3) |] |]

# - : unit = ()

# - : (int * int) array array =
  [| [| (0, 0); (2, 1); (2, 2); (2, 3) |];
    [| (0, 0); (2, 1); (2, 2); (2, 3) |];
```

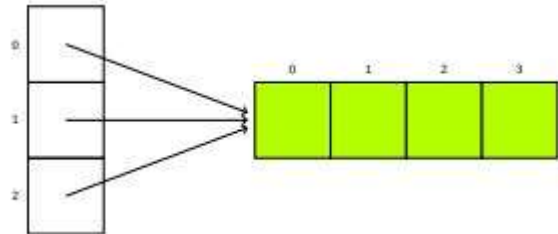
```
[|(0, 0); (2, 1); (2, 2); (2, 3)|]]]
```

Partage de valeur : les lignes sont égales en termes d'adresse physique. En fait, dans cette construction l'initialisation du tableau sous la forme

```
let t = Array.make 3 (Array.make 4 (0,0))
```

crée bien un tableau contenant trois lignes chacune étant un tableau de couples d'entiers de longueur 4, mais ces lignes ne sont qu'un seul et même tableau.

On a alors : `mat_4.(0) == mat_4.(1)` qui renvoie `true`



## Exercices

### Exercice 1

Ecrire une fonction de type `# val inverse : string -> string = <fun>` recevant une chaîne de caractères et retournant le miroir de cette chaîne de caractères (par exemple, « repus » devient « super »). Une fonction récursive auxiliaire sera définie.

Remarque : la fonction `string_of_char` n'existant pas, il faut l'écrire. Pour cela, vous pourrez utiliser le `String.make 1 c`, de signature `string_of_char : char -> string = <fun>`, qui constitue une chaîne de caractères de longueur 1, contenant le paramètre `c` passé en argument.

```
let string_of_char c = String.make 1 c;;
let inverse chaine =
  let len = String.length (chaine) and miroir = "" in
  let rec aux miroir i = match i with
    | 0 -> miroir
    | _ -> aux (miroir^string_of_char chaine.[i-1])(i-1)
  in aux miroir len;;
let ch = "repus";;
let suivant = inverse ch;;
```

### Exercice 2

- 1) Ecrire une fonction `crochets` de type `val crochets : string -> string = <fun>` recevant une chaîne de caractères et renvoyant une chaîne de caractères pour laquelle chacun des caractères de la chaîne reçue est entre crochets : `"chaîne" -> "[c][h][a][i][n][e]"`

```
let crochets chaine =
let nouvelle = String.create(String.length(chaine)*3)
```

```

and l = String.length(chaine)-1 in
for i = 0 to l do
  nouvelle.[3*i]<- '[';
  nouvelle.[3*i+1]<- chaine.[i];
  nouvelle.[3*i+2]<- ']';
done;
nouvelle;;
let c1 = "super";;
let c2 = crochets c1;;

(* Passage des caractères entre crochets -> chaines non mutables*)
let string_of_char c = String.make 1 c;;

let string_of_char c = String.make 1 c;;

let crochets_rec chaine =
  let nouvelle = "" and l = String.length(chaine) in
  let rec aux nouvelle i = match i with
    | i when i = l -> nouvelle (* l seul ne fonctionne pas *)
    | _ -> aux (nouvelle^string_of_char '[' ^ string_of_char
chaine.[i]^ string_of_char ']') (i+1)
  in aux nouvelle 0;;
let ch1 = "super";;
let cr_ch = crochets_rec ch1;;

```

- 2) Ecrire la fonction decrochets de type val crochets : string -> string = <fun> effectuant l'opération inverse, à savoir transformant "[c][h][a][i][n][e]" en "chaine".

```

let decrochets chaine =
let nouvelle = String.create(String.length(chaine)/3)
and l = String.length(chaine)-1
and k = ref 0 in
for i = 1 to l do
  if (i mod 3) = 1 then begin
    nouvelle.[!k]<- chaine.[i];
    k := !k+1;
  end;
done;
nouvelle;;

let c1 = "super";;
let c2 = crochets c1;;
let c3 = decrochets c2;;

```

Sortie écran :

```
# val decrochets : string -> string = <fun>
```

```
# val c1 : string = "super"
# val c2 : string = "[s][u][p][e][r]"
# val c3 : string = "super"
# val decrochets : string -> string = <fun>
# val c1 : string = "super"
# val c2 : string = "[s][u][p][e][r]"
# val c3 : string = "super"

(* La même en récursif et chaîne non mutable *)
let string_of_char c = String.make 1 c;;

let decrochet_rec chaine =
  let l = String.length(chaine)-1 and nouvelle = "" in
  let rec aux nouvelle i = match i with
    | i when i > l-1 -> nouvelle
    | _ -> aux (nouvelle^string_of_char chaine.[i+1]) (i+3)
  in aux nouvelle 0;;
let c1 = "super";;
let c2 = crochets_rec c1;;
let c3 = decrochet_rec c2;;
```

### Exercice 3 : sous-chaînes

Une sous-chaîne d'une chaîne de caractères *s* est une chaîne composée de caractères consécutifs de *s*. Elles peuvent être caractérisées par l'indice de début et leur longueur.

Ecrire une fonction `sous_chaine` de type `val sous_chaine : string -> int -> int -> string = <fun>` recevant une chaîne de caractères, `chaine`, un entier, `debut`, correspondant à l'indice de début de la sous-chaîne à créer et un entier, `longueur`, correspondant à la longueur de la sous-chaîne à construire et la renvoyant. Cette fonction devra s'assurer, avant de constituer cette sous-chaîne, que les longueurs sont compatibles.

L'instruction `failwith` (« message d'erreur ») sera utilisée : elle permet de faire avorter le programme en cas d'erreur irrattrapable.

```
let sous_chaine chaine debut longueur =
  let petite = String.create (longueur)
  and k = ref 0 in
  if (longueur + debut > String.length (chaine)) then begin
    failwith ("Longueurs non compatibles");
  end else begin
    for i = debut to longueur+debut-1 do
      petite.[!k]<- chaine.[i];
      k := !k + 1;
    done;
  end;
  petite;;

let c1 = "coucou";;
let c2 = sous_chaine c1 2 5;;
```



Sortie en # Exception: Failure "Longueurs non compatibles".

```
(* La même en récursif et chaîne non mutable *)
let sous_ch chaîne debut longueur =
  let petite = "" and k = longueur+debut in
  match k with
  | k when k > String.length (chaîne) -> failwith "longueurs
incompatibles"
  | _ -> let sous_chaine = "" in
        let rec aux ch i = match i with
          | i when i > (longueur-1) -> ch
          | _ -> aux (ch^string_of_char (chaîne.[debut + i]))
        (i+1);
  in aux petite 0;;
let c1 = "coucou";;
let c2 = sous_ch c1 2 3;;
```

#### Exercice 4

Le nombre de jours dans un mois, pour une année non bissextile, peut être décrit par un tableau de longueur 12 :

```
let nb_jours = [| 31 ; 28 ; 31 ; 30 ;
31 ; 30 ; 31 ; 31 ;
30 ; 31 ; 30 ; 31 |]
```

Définir une fonction `nb_jours` de type `int -> int` qui donne le nombre de jours d'un mois dont le numéro est passé en paramètre. Cette fonction devra tester le numéro du mois pris en paramètre, renvoyer le message d'erreur si le numéro du mois est strictement inférieur à 1 ou strictement supérieur à 12 ou renvoyer le nombre de jours du mois correspondant si le numéro du mois est valide.

Pour déclencher un tel comportement, utiliser l'instruction `failwith(« mois incorrect »)`. Le `failwith` permet de lever une exception irrattrapable.

La sortie écran attendue est la suivante :

```
# val nb_jours : int -> int = <fun>
# val mars : int = 31
# Exception: Failure "mois incorrect".

let nb_jours m =
  let nb_jours = [| 31 ; 28 ; 31 ; 30 ;
31 ; 30 ; 31 ; 31 ;
30 ; 31 ; 30 ; 31 |]
  in
  if (m < 1) || (m > 12) then
    failwith ("mois incorrect")
  else
    nb_jours.(m-1);;
```

```
let mars = nb_jours 3;;  
let autre = nb_jours 15;;
```

### Exercice 5

Ecrire une fonction prenant une chaîne de caractères quelconque en argument et renvoyant un tableau de 256 valeurs permettant de dénombrer le nombre d'occurrences de chaque caractères (codé sur 8 bits, donc 256 valeurs possibles pour les char) dans cette chaîne.

La fonction `int_of_string` sera utilisée pour convertir un caractère de la chaîne étudiée en entier.

Rappel : il est possible de définir deux variables dans un `let` :

```
let compteurs = ... and n = ... in ...
```

```
let chaine = "Bonjour les MP2I !!!!!";;  
let compteurs = Array.make 256 0 and n = String.length chaine in  
for i=0 to n - 1 do  
  compteurs.(int_of_char chaine.[i]) <- compteurs.(int_of_char chaine.[i])  
+ 1  
done ;  
compteurs;;
```

(\* Le même mais en récursif \*)

```
let compte chaine =  
  let compteurs = Array.make 256 0 and n = String.length chaine in  
  let rec aux i = match i with  
    | i when i > (n-1) -> compteurs  
    | _ -> compteurs.(int_of_char chaine.[i]) <-  
compteurs.(int_of_char chaine.[i]) + 1;  
aux (i+1)  
in aux 0;;
```

```
let chaine = "Bonjour les MP2I !!!!!";;  
let c = compte chaine;;  
c = compteurs;;
```

**TABLE ASCII**

NUL	0	ESP	32	@	64	`	96
SOH	1	!	33	A	65	a	97
STX	2	"	34	B	66	b	98
ETX	3	#	35	C	67	c	99
EOT	4	\$	36	D	68	d	100
ENQ	5	%	37	E	69	e	101
ACK	6	&	38	F	70	f	102
BEL	7	'	39	G	71	g	103
BS	8	(	40	H	72	h	104
HT	9	)	41	I	73	i	105
LF	10	*	42	J	74	j	106
VT	11	+	43	K	75	k	107
FF	12	,	44	L	76	l	108
CR	13	-	45	M	77	m	109
SO	14	.	46	N	78	n	110
SI	15	/	47	O	79	o	111
DLE	16	0	48	P	80	p	112
DC1	17	1	49	Q	81	q	113
DC2	18	2	50	R	82	r	114
DC3	19	3	51	S	83	s	115
DC4	20	4	52	T	84	t	116
NAK	21	5	53	U	85	u	117
SYN	22	6	54	V	86	v	118
ETB	23	7	55	W	87	w	119
CAN	24	8	56	X	88	x	120
EM	25	9	57	Y	89	y	121
SUB	26	:	58	Z	90	z	122
ESC	27	;	59	[	91	{	123
FS	28	<	60	\	92		124
GS	29	=	61	]	93	}	125
RS	30	>	62	^	94	~	126
US	31	?	63	_	95	DEL	127