LANGAGE OCaml – TP 5
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 :

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`.

```
int_of_char ('c');;
int_of_char (!c);;
char_of_int (97);;
```

```
# - : int = 99
# - : int = 100
# - : char = 'a'
```

```
let rec affiche n =
  match n with
   |n when n > 101 -> print_newline();
              print_string "Fini"
   | _ -> print_char (char_of_int (n));
              print_char (char_of_int (32));
              affiche (n+1);;

affiche 97;
```

```
a b c d e

Fini- : unit = ()
```

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.

```
"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]<-'I';;


print_char (chaine.[16]);;
```

Sortie écran :

Les chaînes de caractères peuvent aussi être déclarées comme référence.

```
let s = ref "";;

s := 'a';;
(* Attention au type :
Un caractère n'est pas
une chaîne de caractères
comportant un unique
caractère *)

s :="a";;

s;;
```

```
# 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"}
```

Opération sur les chaînes de caractères

```
let chaine_1 = "Bon";;
let chaine_2 = "jour";;
let chaine_3 = chaine_1 ^ chaine_2;;

Sortie écran :
# val chaine_1 : string = "Bon"
# val chaine_2 : string = "jour"
# val chaine_3 : string = "Bonjour"
```

La concaténation de deux chaînes de caractères `s1` et `s2` est l'opération consistant à mettre bout à bout ces deux chaînes.

L'opérateur `^` qui permet d'exprimer la concaténation de deux chaînes de caractères.

```
let chaine_1 = "aa";;
let chaine_2 = "ab";;
chaine_1 = chaine_2;;
chaine_1 <> chaine_2;;
chaine_1 <= chaine_2;;

Sortie écran :

# val chaine_1 : string = "aa"
# val chaine_2 : string = "ab"
# - : bool = false
# - : bool = true
# - : bool = true
```

Les opérateurs `=`, `<`, `<=`, `>` et `>=` permettent de comparer deux chaînes de caractères.

Si `s1` et `s2` sont deux expressions de type `string`, alors :

- `s1 = s2` est vrai si et seulement si les deux chaînes de caractères `s1` et `s2` sont égales ;
- `s1 < s2` est vrai si et seulement si la chaîne de caractères `s1` vient strictement avant `s2` dans l'ordre lexicographique ;
- `s1 <= s2` est vrai si et seulement si la chaîne de caractères `s1` vient avant `s2` dans l'ordre lexicographique ou est égale à `s2` ;

Module String

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

`let chaine = "Bonjour";;`
`let l = String.length (chaine);;`
 	
Sortie écran
`# val chaine : string = "Bonjour"`
`# val l : int = 7:`
	Longueur d'une chaîne de caractères.

`let chaine = "Bonjour";;`
`let l = String.length (chaine);;`
`let copie = String.copy chaine;;`
`let copie = copie^copie;;`
`copie;;`
`chaine;;`
 	
Sortie écran :
`# val chaine : string = "Bonjour"`
`# val l : l = 7`
`Strings now immutable: no need to copy`
`# val copie : string = "Bonjour"`
`# val copie : string = "BonjourBonjour"`
`# - : string = " BonjourBonjour "`
`# - : string = "Bonjour"`
	Attention aux problèmes de valeurs partagées.
	
	L'utilisation de la fonction `String.copy` permet de dupliquer une chaîne de caractères en créant une nouvelle variable.
	
	Attention : les chaînes de caractères ne sont pas mutables (elles ne peuvent être modifiées).

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	
`[|3; 2; 1|];;`	Définit un tableau de 3 éléments. Le séparateur est le point-virgule.

	La sortie dans l'interpréteur est alors : `# - : int array = [|3; 2; 1|]` Le type des éléments contenu dans le tableau est inféré : `int` dans ce cas. 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.
`[|3. ; 2; 1|];;`	Sortie écran :
`[| |] ;;`	Sortie écran : `# - : 'a array = [||]` Rappel : `'a` : type quelconque (type alpha)
Déclaration d'un tableau comme variable	
`let mon_tableau = [|0;2;4;6;8|];;`	Une variable de type `int array` est créée. `# val mon_tableau : int array = [|0; 2; 4; 6; 8|]`
`let tableau_chaine = [|"super" ; "tableau"|];;`	`# val tableau_chaine : string array = [|"super"; "tableau"|]`

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	
`let tableau = Array.make 5 8;;`	Syntaxe : `Array.make` *`taille valeur`* `;;` Création d'un tableau à partir de sa longueur et d'une valeur par défaut. Sortie écran : `val tableau : int array = [|8; 8; 8; 8; 8|]` Toutes les cases contiennent la même valeur.
	La fonction `Array.init` de type `int → (int → 'a) → 'a array` 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.

`let carre_0 x = x*x in`
`  Array.init 11 carre_0;;`
	Sortie écran :
	
	En utilisant le `Array.init`, proposer des instructions permettant d'obtenir les variables et sorties écran ci-dessous :
	`# val carre_1 : int -> int = <fun>`
	
	`# 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|]`

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

	Créer le tableau : `# val rire : string array = [|"ha"; "ha"; "ha" ; "ha"; "ha"; "ha"; "ha"|]`

	Instructions permettant l'affichage :
	`# ha`
	`ha`
	`ha`
	`ha`
	`ha`
	`ha`
	`ha`

	`- : unit = ()` Une boucle for sera exceptionnellement utilisée (voir memento Caml distribué au premier TP).

	Instructions permettant la modification et l'affichage :
	`# ha`
	`ha`
	`ha`
	`ho`
	`ha`
	`ha`
	`ha`
	`- : unit = ()`

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 :

`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 ;;`	

Quel est le type de la fonction affiche_premier ?

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

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.

`let t3 = Array.make 3 2;;`
`t3;;`
`let t4 = t3;;`
`t3.(0)<-0;;`
`t3;;`
`t4;;`
`let a = Array.length t3;;`
`a;;`
 	
`(* On dit que les valeurs sont partagées *)`
 	
`(* Array.length : à connaître ! *)`
	Sortie écran :

`let t5 = Array.copy t4;;`
`t4;;`
`t5;;`
`t4.(1)<-4;;`
`t3;;`
`t4;;`
`t5;;`	

Tester et commenter les effets des instructions suivantes :

`let ts1 = Array.make 3 "baba";;`
`let ts2 = Array.make 3 ts1.(0);;`
`ts1;;`
`ts2;;`
 	
`ts2.(0)<-"bibi";;`
 	
`ts1;;`
`ts2;;`
	Sortie écran :

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

`[|1; 2; 3|] = [|1; 2; 3|];;`
 	
`[|1; 2; 3|] = [|3; 2; 1|];;`
 	
`let t = [|1; 2; 3|] in`
`t = Array.copy t;;`	

`[|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;;`	

Tableaux en paramètres de fonctions

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

```
Algorithme : doublement des valeurs d'un tableau
Entrée : tableau t de n entiers
Sortie : tableau t' de n entiers valant le double des valeurs du tableau
passé en argument

Début
Pour chaque indice i du tableau faire
    t'[ i ] := 2 x t[ i ]
finpour
retourner t'
fin
```

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 :

`(* fonction double_elements :`
`int array -> 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  *)`
	Sortie écran :

```
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) <- 2 * t.(i)
done ;
t1 ;; (* Pour renvoyer t1 *)

let tab = Array.make 3 2;;

let tab_bis = double_elements tab;;
```

```
(* Fonction doubler_elements :
int array -> unit

Parametre t : int array = tableau
dont on veut doubler les elts

action : double 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) <- 2 * t.(i)
done ;;
(* pas de variable renvoyée :
   passage par adresse du tableau *)

let table = Array.make 3 4;;
doubler_elements table;;

table;;
```

Tableau et filtrage

Analyser et commenter la sortie écran des instructions suivantes :

```
let t = [|4;2|];;
match t with
| [| |]-> "Vide"
| [|1;_|]->"Commence par 1"
| [|2;_|]->"Commence par 2"
| [|x;y|]->"t commence par "^string_of_int(x)
| _->"Autre cas";;
```

Sortie écran :

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 déstructurant *)
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 =
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;;
```

Sortie écran :

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;;
```

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 :

```
mat_2.(1);;
```

Sortie écran :

```
mat_2.(1)<-[|(0, 0); (0, 0);
(0, 0)|];;

mat_2.(1);;


mat_2;;
```

Sortie écran :

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 :

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.

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 : `"chaine" -> "[c][h][a][i][n][e]".` Une fonction auxiliaire récursive sera définie.
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".` Une fonction auxiliaire récursive devra être définie.

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. Une fonction récursive auxiliaire deva être écrite.

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".
```

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. Une fonction auxiliaire récursive devra être écrite.

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

TABLE ASCII

NUL	0
SOH	1
STX	2
ETX	3
EOT	4
ENQ	5
ACK	6
BEL	7
BS	8
HT	9
LF	10
VT	11
FF	12
CR	13
SO	14
SI	15
DLE	16
DC1	17
DC2	18
DC3	19
DC4	20
NAK	21
SYN	22
ETB	23
CAN	24
EM	25
SUB	26
ESC	27
FS	28
GS	29
RS	30
US	31

ESP	32
!	33
"	34
#	35
$	36
%	37
&	38
'	39
(	40
)	41
*	42
+	43
,	44
-	45
.	46
/	47
0	48
1	49
2	50
3	51
4	52
5	53
6	54
7	55
8	56
9	57
:	58
;	59
<	60
=	61
>	62
?	63

@	64
A	65
B	66
C	67
D	68
E	69
F	70
G	71
H	72
I	73
J	74
K	75
L	76
M	77
N	78
O	79
P	80
Q	81
R	82
S	83
T	84
U	85
V	86
W	87
X	88
Y	89
Z	90
[	91
\	92
]	93
^	94
_	95

`	96
a	97
b	98
c	99
d	100
e	101
f	102
g	103
h	104
i	105
j	106
k	107
l	108
m	109
n	110
o	111
p	112
q	113
r	114
s	115
t	116
u	117
v	118
w	119
x	120
y	121
z	122
{	123
|	124
}	125
~	126
DEL	127