

Aspects impératif de OCaml

OCaml est un langage *fonctionnel*, et on peut tout faire avec des fonctions, ... et des constantes!

Mais on vous fournit aussi quelques constructions impératives pour les quelques programmes qui en ont besoin :

- entrée/sorties
- boucles for
- tableaux modifiables en place (Array)

Les entrées et sorties standard

Le "toplevel" OCaml permet de faire énormément de choses sans besoin de lire et écrire sur un terminal ou des fichiers, *mais* on vous donne une panoplie de fonctions qui écrivent sur le terminal et qui lisent sur le terminal.

Par exemple :

```
# print_string;;
- : string -> unit = <fun>
# print_string "Bonjour, tout le monde\n";
Bonjour, tout le monde
- : unit = ()
```

Notez bien le type `unit` : il est utilisé pour les fonctions qui ne rendent pas de valeurs intéressants... sa seule valeur possible est

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

Lire et écrire : les briques de base

```
# print_string;;
- : string -> unit = <fun>
# print_char;;
- : char -> unit = <fun>
# print_int;;
- : int -> unit = <fun>
# print_float;;
- : float -> unit = <fun>
```

1

```
#read_int;;
- : unit -> int = <fun>
# read_float;;
- : unit -> float = <fun>
# read_line;;
- : unit -> string = <fun>
```

Les boucles : for i = ... to ... do ... done

```
# for i = 65 to 90 do
  print_char (Char.chr i)
done;;

ABCDEFGHIJKLMNOPQRSTUVWXYZ : unit = ()
```

IMPORTANT :

- la "variable de boucle" `i` prend successivement les valeurs 65, ..., 90, et ne peut pas être modifiée dans le corps de la boucle
- la boucle toute entière est de type `unit`, donc ...
- le type du corps de la boucle devrait être du type spécial `unit`

Exercice

Qu'est-ce que cette boucle fait ?

```
# for j = 65 to 90 do
  let j = 65 in print_char (Char.chr j)
done;;

- [] elle ne termine jamais, et imprime une infinité de A
- [] elle imprime 25 A
```

Tableaux et matrices

Un tableau contient une suite de valeurs de même type, accessibles par leur *index*, qui est un entier compris entre 0 et la taille du tableau moins un.

En OCaml on peut créer un tableau en énumérant explicitement ses éléments :

```
# let v = [|1;4;5;10|];;
val v : int array = [|1; 4; 5; 10|]
```

et on peut accéder aux éléments du tableau par la notation :

2

```
# v.(0);;
- : int = 1
```

On peut *modifier en place* un tableau :

```
# v.(0);;
- : int = 1

# v.(0) <- 33;;
- : unit = ()

# v.(0);;
- : int = 33
```

Création de tableaux

On peut aussi créer des tableaux de taille arbitraire grâce à la fonction

```
# Array.make;;
- : int -> 'a -> 'a array = <fun>
```

le premier paramètre est la taille du tableau, le deuxième est la valeur utilisée pour l'initialiser.

```
# Array.make 3 'a';;
- : char array = [|'a'; 'a'; 'a'|]

# Array.make 2 0;;
- : int array = [|0; 0|]
```

Matrices

Une matrice est simplement un tableau à deux dimensions¹. En OCaml, une matrice est vue comme un tableau de tableaux, mais on dispose de fonctions d'initialisation spécialisées :

```
# Array.make_matrix;;
- : int -> int -> 'a -> 'a array array = <fun>

# let morpion = Array.make_matrix 3 3 ' ';;
val morpion : char array array =
  [|
    [|' ' ; ' ' ; ' ' |];
    [|' ' ; ' ' ; ' ' |];
    [|' ' ; ' ' ; ' ' |];
```

¹Chaque élément est déterminé par deux indices plutôt que un.

3

```
    [|' ' ; ' ' ; ' ' |];
    [|' ' ; ' ' ; ' ' |];
    [|' ' ; ' ' ; ' ' |];
# morpion.(1).(1) <- '0';;
- : unit = ()
# morpion;;
- : char array array =
  [|
    [|' ' ; ' ' ; ' ' |];
    [|' ' ; '0' ; ' ' |];
    [|' ' ; ' ' ; ' ' |];
```

Le projet

Dans la suite, nous allons apprendre à programmer des fonctions récur-sives, et à en prouver la correction par *induction*.

Ces connaissances vous seront utiles pour la réalisation d'un projet, qui cette année porte sur le vaste sujet des *labyrinthes* : vous aurez à en créer, à en dessiner, et à en résoudre.



4