

## Boucles for

**Syntaxe générale** Les boucles for s'écrivent comme suit :

```
1 for i = DEBUT to FIN do
2   CODE
3 done
```

où `i` est un identifiant quelconque, `DEBUT`, `FIN` et `CODE` des expressions OCaml.

**Typage** Une boucle for de la forme précédente est bien typée si `DEBUT` et `FIN` sont de type `int` et `CODE` de type `unit`. Alors, la boucle elle-même est une expression de type `unit`.

**Sémantique** Une boucle for exécute `CODE` avec `i` valant toutes les valeurs entre `DEBUT` et `FIN` **inclus**.

**Exemple** Le code suivant affiche tous les entiers de 1 à  $n$  :

```
1 let compter (n: int) : unit =
2   for i = 1 to n do
3     print_int i;
4     print_newline ()
5   done
```

## Références

Une référence est un objet **mutable** : c'est une case contenant une unique valeur, que l'on peut lire **et modifier**. Le type `'a ref` représente les références dont le contenu est de type `'a`. Les trois opérations de base sont :

```
1 (* ref x crée une référence contenant x *)
2 ref : 'a -> 'a ref
3
4 (* r := x modifie la référence r pour y écrire x *)
5 (:=) : 'a ref -> 'a -> unit
6
7 (* !r renvoie le contenu de r *)
8 (!) : 'a ref -> 'a
```

Par exemple :

```
1 let r = ref 3 in
2 r := !r + 1;
3 print_int !r; (* affiche 4 *)
4 r := !r * !r;
5 print_int !r (* affiche 16*)
```

Les références permettent de servir de variables modifiables dans les boucles.

**Exemple** Voici une fonction simple renvoyant le nombre de diviseurs d'un entier  $n$  :

```
1 let nombre_div (n: int) : int =
2   let d = ref 0 in
3   for i = 1 to n do
4     if n mod i = 0 then
5       d := !d + 1;
6   done;
7   !d
```

**Tableaux** Les tableaux OCaml fonctionnent comme en C. On peut créer un tableau à la main avec `[| ... |]` en séparant les éléments d'un point-virgule (comme les listes), accéder au contenu de la  $i$ -ème case d'un tableau `t` avec `t.(i)`, et écrire  $x$  dans cette case avec `t.(i) <- x`.

Le module `Array` contient des fonctions utiles sur les tableaux, notamment :

```
1 (* Array.make n x renvoie un tableau de n cases, toutes égales à x *)
2 Array.make : int -> 'a -> 'a array
3
4 (* Array.length t renvoie la longueur de t *)
5 Array.length : 'a array -> int
```

**Exemple** Voici deux fonctions sur les tableaux :

```
1 (* Renvoie la somme des éléments de t *)
2 let somme (t: int array) : int =
3   let s = ref 0 in
4   let n = Array.length t in
5   for i = 0 to n-1 do
6     s := !s + t.(i)
7   done;
8   !s
9
10 (* Renvoie un tableau de taille n >= 2 contenant
11    les n premiers nombres de fibonacci, avec F0 = 0, F1 = 1 *)
12 let fibo_tab (n: int) : int array =
13   let t = Array.make n 0 in
14   t.(1) <- 1;
15   for i = 2 to n-1 do
16     t.(i) <- t.(i-1) + t.(i-2)
17   done;
18   t
```

## Boucles while

**Syntaxe générale** Les boucles while s'écrivent de la manière suivante :

```
1 while CONDITION do
2   CODE
3 done
```

où `CONDITION` est une expression supposée de type `bool`, et `CODE` une expression supposée de type `unit`. La boucle est elle-même de type `unit`.

Une boucle while exécute `CODE` jusqu'à ce que `CONDITION` devienne fausse.

**Exemple** Implémentation de l'exponentiation rapide :

```
1 let exp (x: float) (n: int) : float =
2   let r = ref 1. in
3   let px = ref x in
4   let nn = ref n in
5   while !nn > 0 do
6     if !nn mod 2 = 1 then
7       r := !r *. !px;
8       px := !px *. !px;
9       nn := !nn / 2
10  done;
11  !r
```

## Parenthésage

Lorsque l'on fait une suite d'instructions dans un if-then-else, on les entoure par les mots-clés `begin` et `end`, qui agissent exactement comme les parenthèses :

```
1 if bla then begin
2   print_string "bli";
3   a := !a + 1;
4   ...
5   assert blo
6 end
```

Sans la paire `begin` / `end`, seul le premier `print_string` serait considéré comme faisant partie du `if`. On utilise la même syntaxe pour faire des suites d'instructions dans un cas de match with :

```
1 match ... with
2 | ... -> begin
3   ...;
4   ...
5 end
6 | -> ...
```