Boucles for

Syntaxe générale Les boucles for s'écrivent comme suit :

```
for i = DEBUT to FIN do
  CODE
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 :

```
let compter (n: int) : unit =
  for i = 1 to n do
    print_int i;
    print_newline ()
  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 :

```
(* ref x crée une référence contenant x *)
ref : 'a -> 'a ref

(* r := x modifie la référence r pour y écrire x *)
(:=) : 'a ref -> 'a -> unit

(* !r renvoie le contenu de r *)
(!) : 'a ref -> 'a
```

Par exemple :

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

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

```
(* Array.make n x renvoie un tableau de n cases, toutes égales àx *)
Array.make : int -> 'a -> 'a array

(* Array.length t renvoie la longueur de t *)
Array.length : 'a array -> int
```

Exemple Voici deux fonctions sur les tableaux :

```
(* Renvoie la somme des éléments de t *)
let somme (t: int array) : int =
  let s = ref 0 in
  let n = Array.length t in
  for i = 0 to n-1 do
    s := !s + t.(i)
  done;
  !s

(* Renvoie un tableau de taille n >= 2 contenant
   les n premiers nombres de fibonacci, avec F0 = 0, F1 = 1 *)
let fibo_tab (n: int) : int array =
  let t = Array.make n 0 in
  t.(1) <- 1;
  for i = 2 to n-1 do
    t.(i) <- t.(i-1) + t.(i-2)
  done;
  t
```

Boucles while

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

```
while CONDITION do
  CODE
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 :

```
let exp (x: float) (n: int) : float =
  let r = ref 1. in
  let px = ref x in
  let nn = ref n in
  while !nn > 0 do
    if !nn mod 2 = 1 then
      r := !r *. !px;
    px := !px *. !px;
    nn := !nn / 2
  done;
  !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 :

```
if bla then begin
  print_string "bli";
  a := !a + 1;
  ...
  assert blo
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 :

```
match ... with
| ... -> begin
  ...;
  ...
end
| -> ...
```

Points virgules

Le point-virgule simple `;` sert à **séparer** deux expressions OCaml. Précisément, si `e1` et `e2` sont des expressions OCaml, alors `e1; e2` est une expression, dont le type est celui de `e2`. Pour l'évaluer, on évalue `e1`, on en ignore le résultat, puis on évalue `e2`. Pour le type `unit`, ceci permet d'enchaîner des instructions impératives (affectations, boucles, print, assert...).

A retenir : **pas de point-virgule à la fin d'une expression**, seulement pour séparer deux instructions.

En particulier, il ne faut **JAMAIS** mettre de point virgule à la fin d'une fonction, sinon OCaml ne saura pas comment lire le code, et affichera généralement une erreur étrange située à la fin du fichier.

Par exemple, voici du code invalide :

```
let affichage () =
  print_string "MP";
  print_int 2;
  print_string "I";

let test () =
  affichage ()
```

Ce code cause une erreur de syntaxe, que l'on peut corriger en enlevant le point-virgule à la fin de la fonction `affichage`.

Parcours de tableau

Le code suivant est une base que l'on peut apprendre par cœur et qu'il faut pouvoir ressortir instantanément lorsque l'on veut faire une boucle qui parcourt tous les éléments d'un tableau :

```
let ma_fonction (t: 'a array) =
  let n = Array.length t in
  for i = 0 to n-1 do
    ...
  done
```

On peut ensuite adapter aux détails du problème : commencer ou terminer à d'autres indices, parcourir dans le sens inverse, etc...

Arrêt de boucle

Contrairement au C, OCaml ne permet pas de "return" au beau milieu d'une boucle. Si l'on veut arrêter une boucle, on peut utiliser une référence booléenne indiquant s'il faut continuer la boucle ou non. Par exemple, pour chercher l'indice d'apparition d'un élément dans un tableau :

```
(* Renvoie la première occurrence de x dans t, ou -1 si x n'apparaÃ®t pas *)
let indice (t: 'a array) (x: 'a) : int =
  let n = Array.length t in
  let trouve = ref false in
  let i = ref 0 in
  while !i < n && not !trouve do
    if t.(i) = x then begin
      trouve := true
    end;
    i := !i+1
  done;
  (* attention, le code a incrémenté i mÃªme si on a trouvé *)
  if !trouve then !i - 1 else -1
```

Une alternative aurait été d'utiliser des exceptions pour arrêter la boucle en cours d'exécution.

Boucle pour décroissante

Par défaut, les boucles pour OCaml sont forcément croissantes si l'on utilise le mot-clé `to`. On peut le remplacer par `downto` pour faire une boucle décroissante :

```
1 for i = 10 downto 1 do
2   print_int i;
3   print_newline()
4 done
```