

Impératif en OCaml

February 20, 2022

1 Références

Il est possible en OCaml de définir des variables modifiables comme en C. Pour définir une variable modifiable de type 'a on va passer par un autre type 'a ref appelé le type des *références sur le type 'a*. Une référence sur le type int peut être vue comme une boîte dans laquelle on a mis un entier. **Ce n'est donc pas un accès direct sur un entier** mais un intermédiaire. Pour le manipuler il faut trois opérations distinctes :

- Créer une référence contenant une valeur initiale : `ref v` est une expression qui s'évalue en une référence sur le type de `v` avec comme valeur initiale `v`. Par, exemple `ref 3` est une référence sur le type `int` avec comme valeur initiale 3. Le plus souvent, on va nommer cette référence par une variable avec un `let ... in` comme dans `let x = ref 3 in expr` où on pourra utiliser la référence dans `expr`.
- Obtenir la valeur actuelle d'une référence : si `r` est une référence, on écrit `!r` pour obtenir la valeur qu'elle contient. Ainsi si `r` est de type 'a ref alors `!r` est de type 'a.
- **Changer la valeur d'une référence** : On rajoute une nouvelle expression de la forme `exprref := exprval` où `exprref` est une expression de type 'a ref et `exprval` de type 'a et qui change la valeur de la référence pour y placer la valeur de `exprval`. On l'utilise le plus souvent directement avec un nom de variable comme dans `x := 3`.

Voilà un exemple utilisant des références :

```
[1]: let x = ref 0 in

Printf.printf "Ici x contient %d\n" !x; (* Printf c'est le module et printf la
↳ fonction dedans *)
x := !x + 1;
Printf.printf "Ici x contient %d\n" !x;
let y = ref !x in
Printf.printf "Nouvelle ref y contient %d\n" !y;
Printf.printf "x = y ? %s\n" (if x = y then "oui" else "non");
flush stdout
```

```
Ici x contient 0
Ici x contient 1
Nouvelle ref y contient 1
x = y ? oui
```

```
[1]: - : unit = ()
```

On remarque que l'égalité entre références porte uniquement sur leur contenu.

En fait, pour l'usage qu'on va en avoir, les références ressemblent aux pointeurs : `t * -> t ref`, `*t -> !t`, `*t = ... -> t := ...`. Mais le point très différent, c'est qu'en C on peut pointer sur une variable existante avec `&x` alors qu'ici il faut créer la référence avant.

On peut ainsi passer des références en paramètres pour modifier dans une fonction une valeur définie en dehors, comme en C, mais cette valeur aura été définie précédemment. On pourra donc faire par exemple :

```
let incr x =  
  x := !x + 1  
  
let r = ref 3  
  
incr r;  
incr r;  
print_int !r (* va afficher 5 *)
```

(d'ailleurs cette fonction `incr` d'incrementation est prédéfinie en OCaml.

```
[2]: let echange x y : unit = (* écrire une fonction qui échange le contenu des  
  ↪ références x et y *)  
  (* BEGIN SOLUTION *)  
  let z = !x in  
  x := !y;  
  y := z  
  (* END SOLUTION *)
```

```
[2]: val echange : 'a ref -> 'a ref -> unit = <fun>
```

```
[3]: let x = ref 3 in  
let y = ref 2 in  
echange x y;  
assert(!x = 2 && !y = 3);  
let x = ref 'a' in  
let y = ref 'b' in  
echange x y;  
assert(!x = 'b' && !y = 'a');  
"Tous les tests sont corrects"
```

```
[3]: - : string = "Tous les tests sont corrects"
```

Il découle de la nature des variables en OCaml des comportements qu'il faut avoir en tête quand

on programme avec des références. Ainsi, lorsqu'on écrit `let x = ref v` et plus loin `let z = x` alors `z` et `x` sont deux noms différents pour la même référence. Ainsi, modifier la valeur de `x` va naturellement modifier celle de `z`, car c'est exactement la même donnée.

2 Des boucles

Il y a en OCaml des boucles `for` et `while`, comme en C, mais contrairement à ce dernier, ce sont encore de simples expressions.

2.1 Rappel sur ; et la notion implicite de blocs

On a vu que si `e1` et `e2` sont deux expressions alors écrire `e1 ; e2` signifie effectuer `e1` puis `e2`, ce qui n'a de sens que si `e1` n'a pas de valeur signifiante, c'est-à-dire si `e1` est du type `unit`. On peut ainsi écrire

```
print_string "x = "; print_int x; x
```

qui aura le type `int` car c'est le type de `x`. Mais on ne peut pas écrire `3 ; 2` sans avoir d'avertissement sur le fait que la valeur `3` est ignorée.

`;` est un séparateur et non un terminateur. On n'écrira pas `print_string "Bonjour"; print_newline ();` car ce dernier `;` n'est pas nécessaire. Si c'est accepté par OCaml, c'est maladroit car il le transforme automatiquement en `print_string "Bonjour"; print_newline (); ()`.

On en déduit une notion de **bloc d'instructions** en OCaml qui sont des expressions ainsi séparées par des `;` :

```
e1;  
e2;  
(* ... *)  
en
```

Dans la suite, quand on parlera d'expression comme corps de boucle, on aura en tête que cela pourra très bien représenter un tel bloc.

2.2 Boucle for

On écrit

```
for var = debut to fin do  
    expression_corps  
done
```

où `var` est un nom de variable, `debut` et `fin` sont des entiers et `expression_corps` est une expression de type `unit`, souvent un bloc comme précédemment.

Cette boucle va évaluer `expression_corps` avec `var` prenant des valeurs de `debut` à `fin` en **comptant cette dernière valeur** par incrément de 1.

Ainsi, le programme

```

for i = 1 to 5 do
  print_int i
done

```

affichera 12345.

Avec des références, on peut ainsi calculer $\sum_{k=0}^n k$:

```

let somme n =
  let s = ref 0 in
  for i = 0 to n do
    s := !s + i
  done;
  !s (* attention à ne pas renvoyer s mais bien sa valeur !s *)

```

On retrouve ainsi une structure de programme impératif très proche de celle de Python ou du C : variables locales, boucles, valeur de retour.

Attention Un point critique est qu'on ne peut pas faire de `return` inopiné, et donc sortir d'une boucle comme on a pu le faire dans les autres langages. On verra plus tard comment le mécanisme d'exceptions permet de pallier ce problème.

```

[1]: let fact n =
      (* Calculer n! en impératif *)
      (* BEGIN SOLUTION *)
      let p = ref 1 in
      for i = 1 to n do
        p := !p * i
      done;
      !p
      (* END SOLUTION *)

```

```

[1]: val fact : int -> int = <fun>

```

```

[4]: assert(fact 0 = 1);
      assert(fact 1 = 1);
      assert(fact 5 = 120);
      "Tous les tests sont corrects"

```

```

[4]: - : string = "Tous les tests sont corrects"

```

```

[5]: let fibo n =
      (* Calculer fibonacci n en impératif
         attention il faut un peu réfléchir ;-*) *)
      (* BEGIN SOLUTION *)
      let f_p = ref 1 in
      let f_n = ref 0 in
      for i = 1 to n do

```

```

    let t = !f_p in
    f_p := !f_n;
    f_n := !f_n + t
done;
!f_n
(* END SOLUTION *)

```

```
[5]: val fibo : int -> int = <fun>
```

```
[13]: assert(fibo 4 = 3);
assert(fibo 0 = 0);
assert(fibo 1 = 1);
assert(fibo 10 = 55);
"Tous les tests sont corrects"
```

```
[13]: - : string = "Tous les tests sont corrects"
```

2.3 Boucle while

On écrit

```

while condition do
  expression_corps
done

```

où `condition` est du type `bool` et `expression_corps` est une expression de type `unit`.

Cette boucle va évaluer `expression_corps` tant que `condition` s'évalue à `true`. Le plus souvent, il y aura des déréférencement de références dans la condition pour qu'elle puisse changer. Car si l'expression ne peut pas changer, elle va s'évaluer à `true` ou `false` pour toutes les itérations.

```
[15]: let nombre_chiffres n =
  (* renvoie le nombre de chiffres minimal pour écrire
  l'entier naturel n en binaire. On dira ici que 0 a une écriture vide *)
  (* BEGIN SOLUTION *)
  let p = ref 0 in
  let x = ref n in
  while !x <> 0 do
    x := !x / 2;
    p := !p + 1
  done;
  !p
  (* END SOLUTION *)

```

```
[15]: val nombre_chiffres : int -> int = <fun>
```

```
[20]: assert(nombre_chiffres 0 = 0);
      assert(nombre_chiffres 1 = 1);
      assert(nombre_chiffres 16 = 5);
      "Tous les tests sont corrects"
```

```
[20]: - : string = "Tous les tests sont corrects"
```

3 Les tableaux

Les tableaux de valeurs de type 'a en OCaml ont le type 'a array.

Le tableau vide est []. Pour créer un tableau contenant des valeurs initiales, on le déclare comme une liste mais avec les | comme dans [1; 2; 3].

Si t est un tableau, on accède à la case d'indice i avec t.(i). Pour **changer** la valeur à cet indice, on écrit t.(i) <- v qui place la valeur v. C'est à comparer avec r := v mais **attention** aux symboles différents.

On peut faire des tableaux de références et écrire t.(i) := 2 mais on évitera de mélanger les syntaxes. Intrinséquement, t.(i) est déjà une sorte de références, donc ce serait inutile.

Les fonctions de la bibliothèque de tableaux sont dans le module **Array** (documentation ici <https://ocaml.org/api/Array.html>).

De manière non exhaustive :

- Array.length permet d'obtenir le nombre d'élément d'un tableau
- Array.make n x crée un tableau de n éléments contenant la valeur x partout
- Array.init n f crée un tableau de n éléments avec une fonction f ainsi [f 0; f 1; ...; f n]. Très pratique pour faire comme les [2 * i for i in range(n)] de Python (ici, on ferait Array.init n (fun i -> 2*i))

Un exemple qui montre l'usage classique qu'on va faire des tableaux en OCaml :

```
[58]: let somme t =
      let s = ref 0 in
      for i = 0 to Array.length t - 1 do (* ATTENTION au -1 car c'est inclusif *)
        s := !s + t.(i)
      done;
      !s

      let t = Array.init 10 (fun i -> i)

      let s = somme t (* calcule 0 + 1 + 2 + ... + 9 *)
```

```
[58]: val somme : int array -> int = <fun>
```

```
[58]: val t : int array = [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|]
```

```
[58]: val s : int = 45
```

```
[60]: let recherche t x : int option =  
  (* renvoie l'indice de la première valeur de t égale à x.  
    On utilisera un type optionnel pour le résultat *)  
  (* BEGIN SOLUTION *)  
  let v = ref None in  
  for i = 0 to Array.length t - 1 do  
    if t.(i) = x && !v = None  
    then v := Some i  
  done;  
  !v  
  (* END SOLUTION *)
```

```
[60]: val recherche : 'a array -> 'a -> int option = <fun>
```

```
[64]: assert(recherche [|1;2;3|] 2 = Some 1);  
assert(recherche [|1;2;3|] 0 = None);  
assert(recherche [|'a';'b';'c'|] 'c' = Some 2);  
assert(recherche [|]| () = None);  
assert(recherche [|1;1;1;1;1|] 1 = Some 0);  
"Tous les tests sont corrects."
```

```
[64]: - : string = "Tous les tests sont corrects."
```

```
[65]: let indice_maximum t =  
  (* Renvoie le plus petit indice d'un maximum de t  
    On suppose t non vide *)  
  (* BEGIN SOLUTION *)  
  let v = ref 0 in  
  for i = 1 to Array.length t - 1 do  
    if t.(i) > t.(!v)  
    then v := i  
  done;  
  !v  
  (* END SOLUTION *)
```

```
[65]: val indice_maximum : 'a array -> int = <fun>
```

```
[68]: assert(indice_maximum [|1;2;3|] = 2);
      assert(indice_maximum [|'a';'b';'c'|] = 2);
      assert(indice_maximum [|5;6;2|] = 1);
      assert(indice_maximum [|5;6;2;6|] = 1);
      "Tous les tests sont corrects."
```

```
[68]: - : string = "Tous les tests sont corrects."
```

4 Tris

4.1 Par sélection

```
[70]: let swap t i j =
      (* Echange les valeurs situées dans t aux indices
         i et j *)
      (* BEGIN SOLUTION *)
      let z = t.(i) in
      t.(i) <- t.(j);
      t.(j) <- z
      (* END SOLUTION *)
```

```
[70]: val swap : 'a array -> int -> int -> unit = <fun>
```

```
[74]: let t = [|1;2;3|] in
      swap t 0 2;
      assert(t = [|3;2;1|]);
      swap t 1 2;
      assert(t = [|3;1;2|]);
      let t = [|'a';'b';'c'|] in
      swap t 0 2;
      assert(t = [|'c';'b';'a'|]);
      "Tous les test sont corrects"
```

```
[74]: - : string = "Tous les test sont corrects"
```

On rappelle le principe du tri par selection : on cherche le minimum des non triés, on le place a la bonne position, on itère.

```
[78]: let tri_par_selection t : unit =
      (* Trier t par selection *)
      (* BEGIN SOLUTION *)
      let minimum t i =
        let m = ref i in
        for j = i+1 to Array.length t - 1 do
```

```

        if t.(j) < t.(!m)
        then m := j
    done;
    !m
in
for i = 0 to Array.length t - 2 do
    let j = minimum t i in
    swap t i j
done
(* END SOLUTION *)

```

```
[78]: val tri_par_selection : 'a array -> unit = <fun>
```

```
[83]: let t = Array.init 10 (fun i -> 20 - i/2) in
tri_par_selection t;
assert(t = [|16; 16; 17; 17; 18; 18; 19; 19; 20; 20|]);
let s = [|'d'; 'e'; 'a'; 'd'; 'b'; 'e'; 'e'; 'f'|] in
tri_par_selection s;
assert(s = [|'a'; 'b'; 'd'; 'd'; 'e'; 'e'; 'e'; 'f'|]);
"Tous les tests sont corrects"
```

```
[83]: - : string = "Tous les tests sont corrects"
```

4.2 Par insertion

Le tri par insertion d'un tableau t de n éléments consiste à rechercher itérativement la place d'insertion de l'élément d'indice i dans les éléments d'indice 0 à i sachant que les éléments d'indice 0 à $i - 1$ sont triés. Une fois cette place déterminée, on procède par échanges successifs pour que le tableau des éléments d'indice 0 à i soient triés.

On commence en considérant que l'élément d'indice 0 est, à lui tout seul, un tableau trié, et on procède ainsi pour insérer les éléments d'indice 1 à $n-1$.

```
[98]: let position_insertion t i x =
  (* renvoie la position d'insertion de x dans le tableau t
   considéré trié entre les indices 0 et i-1.
   Si x est plus grand que toutes les autres valeurs, on renverra i *)
  (* On fera en sorte que cette fonction ait une complexité logarithmique en
  ↪ |t| *)
  (* BEGIN SOLUTION *)
  let a = ref 0 in
  let b = ref i in
  while !b - !a > 0 do
    let m = (!a + !b) / 2 in
    if t.(m) < x
    then a := m+1

```

```

        else b := m
    done;
    !a
    (* END SOLUTION *)

```

```
[98]: val position_insertion : 'a array -> int -> 'a -> int = <fun>
```

```
[114]: assert(position_insertion [|1;2;5;8;7;9;10|] 4 3 = 2);
assert(position_insertion [|1;2;5;8;7;9;10|] 4 0 = 0);
assert(position_insertion [|1;2;5;8;7;9;10|] 4 1 = 0);
assert(position_insertion [|1;2;5;8;7;9;10|] 4 9 = 4);
"Tous les tests sont corrects"
```

```
[114]: - : string = "Tous les tests sont corrects"
```

Pour la question suivante, on aimerait faire un `for` en sens inverse, c'est possible en écrivant `downto` à la place de `to`.

```

for i = 5 downto 1 do
    Printf.printf "%d...\n" i
done;
Printf.printf "Partez !"

```

```
[129]: let decale t i j : unit =
    (* place la valeur à l'indice i à l'indice i + 1, celle
    d'indice i + 1 à l'indice i+2, .., celle d'indice j à l'indice j+1.
    Cela revient à décaler vers la droite les valeurs entre les indices i et j
    ce qui permet de libérer la place i *)
    (* BEGIN SOLUTION *)
    for k = j downto i do
        t.(k+1) <- t.(k)
    done
    (* END SOLUTION *)
```

```
[129]: val decale : 'a array -> int -> int -> unit = <fun>
```

```
[126]: let t = [|0;1;2;3;4;5;6|] in
decale t 2 4;
assert(t = [|0; 1; 2; 2; 3; 4; 6|]);
"Tous les tests sont corrects."
```

```
[126]: - : string = "Tous les tests sont corrects."
```

```
[127]: let tri_par_insertion t : unit =
  (* tri t par insertion *)
  (* BEGIN SOLUTION *)
  for i = 1 to Array.length t - 1 do
    let j = position_insertion t i t.(i) in
    let v = t.(i) in
    decale t j (i-1);
    t.(j) <- v
  done
  (* END SOLUTION *)
```

```
[127]: val tri_par_insertion : 'a array -> unit = <fun>
```

```
[128]: let t = Array.init 10 (fun i -> 20 - i/2) in
tri_par_insertion t;
assert(t = [|16; 16; 17; 17; 18; 18; 19; 19; 20; 20|]);
let s = [|'d'; 'e'; 'a'; 'd'; 'b'; 'e'; 'e'; 'f'|] in
tri_par_insertion s;
assert(s = [|'a'; 'b'; 'd'; 'd'; 'e'; 'e'; 'e'; 'f'|]);
"Tous les tests sont corrects"
```

```
[128]: - : string = "Tous les tests sont corrects"
```

4.3 Tri à bulles

On va étudier ici un algorithme de tri très simple : le tri à bulles. Ce tri consiste à identifier des éléments consécutifs dans un tableau qui ne sont pas rangés et à les inverser. C'est-à-dire que si $t.(i) > t.(i+1)$ on échange les valeurs dans $t.(i)$ et $t.(i+1)$.

Résoudre une telle inversion peut être vue comme une bulle qui remonterait progressivement dans un verre d'une boisson gazeuse. C'est cela qui a donné son nom à ce tri. Pour trier, on va donc réaliser des inversions en faisant progresser l'indice i . A la suite d'un balayage on n'a aucune raison d'avoir trié le tableau, on va donc itérer cela jusqu'à ce que le tableau soit trié.

Voici en animation un tri par tri à bulle. Les éléments du tableau sont représentées par des points du plan de coordonnées $(i, t.(i))$.

```
[132]: let tri_a_bulles t : unit =
  (* trier t avec un tri a bulles *)
  (* BEGIN SOLUTION *)
  for i = 0 to Array.length t - 1 do
    for j = 0 to Array.length t - 2 do
      if t.(j) > t.(j+1)
      then swap t j (j+1)
    done
  done
```

```
(* END SOLUTION *)
```

```
[132]: val tri_a_bulles : 'a array -> unit = <fun>
```

```
[133]: let t = Array.init 10 (fun i -> 20 - i/2) in
tri_a_bulles t;
assert(t = [|16; 16; 17; 17; 18; 18; 19; 19; 20; 20|]);
let s = [|'d'; 'e'; 'a'; 'd'; 'b'; 'e'; 'e'; 'f'|] in
tri_a_bulles s;
assert(s = [|'a'; 'b'; 'd'; 'd'; 'e'; 'e'; 'e'; 'f'|]);
"Tous les tests sont corrects"
```

```
[133]: - : string = "Tous les tests sont corrects"
```

5 Démineur

Il est possible de créer un tableau de tableau en utilisant `Array.init` ainsi :

```
let a = Array.init 5 (fun _ -> Array.make 3 0)
```

va donc être un tableau de 5 tableaux de 3 éléments contenant la valeur 0 initialement.

```
a : int array array =
[|
  [| 0; 0; 0 |];
  [| 0; 0; 0 |];
  [| 0; 0; 0 |]
|]
```

Cela permet donc de représenter des matrices. On accède à l'élément situé à la i ème ligne et j ème colonne avec `a.(i).(j)`. De même pour changer sa valeur.

Il est possible de le définir directement avec `Array.make_matrix` comme ici avec `let a = Array.make_matrix 5 3 0`.

On peut obtenir le nombre de ligne avec `Array.length a` et le nombre de colonnes avec `Array.length a.(0)` en supposant `a` non vide.

Have you ever played Minesweeper? It's a cute little game which comes within a certain Operating System which name we can't really remember. Well, the goal of the game is to find where are all the mines within a $M \times N$ field. To help you, the game shows a number in a square which tells you how many mines there are adjacent to that square. For instance, suppose the following 4×4 field with 2 mines (which are represented by an '*' character):

```
*...
....
.*..
....
```

If we would represent the same field placing the hint numbers described above, we would end up with:

```
*100
2210
1*10
1110
```

As you may have already noticed, each square may have at most 8 adjacent squares.

Pour représenter un tel tableau, on va utiliser une matrice de `case` où ‘case est le type suivant :

```
[134]: type case = Mine | Inconnu | Compte of int
```

```
[134]: type case = Mine | Inconnu | Compte of int
```

```
[137]: let complete (t : case array array) : unit =
  (* t est une representation du champ à déminer sous forme de matrice de
  ↪ cases *)
  (* il ne contient que des case Mine ou Inconnu. On demande de déterminer
  ↪ toutes les
  cases Inconnu avec le nombre de mines dans le voisinage direct de la case *)
  (* BEGIN SOLUTION *)
  let n = Array.length t in
  let m = Array.length t.(0) in
  let compte_mine t i j =
    let mines = ref 0 in
    for di = i-1 to i+1 do
      for dj = j-1 to j+1 do
        if di >= 0 && di < n
          && dj >= 0 && dj < m
          && t.(di).(dj) = Mine
        then incr mines
      done
    done;
    !mines
  in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      if t.(i).(j) = Inconnu
      then t.(i).(j) <- Compte (compte_mine t i j)
    done
  done
  (* END SOLUTION *)
```

```
[137]: val complete : case array array -> unit = <fun>
```

```
[139]: let t = Array.make_matrix 4 4 Inconnu in
t.(0).(0) <- Mine;
t.(2).(1) <- Mine;
complete t;
assert(t = [| [| Mine; Compte 1; Compte 0; Compte 0 |];
  [| Compte 2; Compte 2; Compte 1; Compte 0 |];
  [| Compte 1; Mine; Compte 1; Compte 0 |];
  [| Compte 1; Compte 1; Compte 1; Compte 0 |] |]);
"Tous les tests sont corrects."
```

```
[139]: - : string = "Tous les tests sont corrects."
```