

Programmation Impérative

Alan Schmitt

10 octobre 2017

1 / 64

Le style impératif

Inconvénient sémantique difficile à décrire (car on s'éloigne de la logique ...)

Avantage programmation plus efficace de certains problèmes

Et surtout l'interaction est **impérative**

Attention : le danger est dans le mélange *alias* et *mutation*. Être efficace et correct est un sujet de recherche.

2 / 64

Ce qui va changer

- ▶ Au lieu de composer des expressions on va **juxtaposer** des instructions.
- ▶ Le caractère ; est le séparateur d'instructions
- ▶ Une instruction est vue comme un appel de fonction réalisant un effet de bord.
- ▶ Les fonctions utilisées comme instructions sont généralement à valeur dans `unit`. En effet la valeur qu'elles pourraient calculer est perdue.

Les exceptions

Les exceptions

Pourquoi les exceptions ?

- ▶ Pour indiquer une situation exceptionnelle (erreur dynamique).
- ▶ Pour modifier le flot de contrôle du programme.

5 / 64

Les exceptions: une nécessité (1)

Exemple: fonction qui rend le plus grand élément d'une liste polymorphe non vide

```
let rec plgrd l = match l with
| [a] -> a
| (a::r) -> max a (plgrd r)
```

Characters 18-69:

```
.....match l with
| [a] -> a
| (a::r) -> max a (plgrd r)..
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val plgrd : 'a list -> 'a = <fun>
```

Cette fonction n'a en effet pas de sens si la liste est vide.

6 / 64

Les exceptions: une nécessité (2)

- ▶ **Attitude 1:** On ne tient pas compte de l'avertissement. On promet qu'on sera "sage" (pas d'appel `plgrd []`). Mais ce contrat n'est pas reflété dans le type: problème si `plgrd` est utilisée par d'autres
- ▶ **Attitude 2:** on prévoit le cas de la liste vide !

```
let rec plgrd l = match l with
| [] -> ???
| [a] -> a
| (a::r) -> max a (plgrd r)
```

Problème: que mettre à la place de ???

- ▶ On ne connaît pas le type des éléments de la liste.
- ▶ Même si c'est `int`, que retourner ? `0` ? `min_int` ?

7 / 64

Les exceptions: une nécessité (3)

En C, les fonctions systèmes rendent une valeur spéciale (`malloc`, `fopen`, `printf`, ...)

La fonction appelante devrait tester et traiter cet valeur car il peut y avoir des situations exceptionnelles ! Au cas ou elle ne peut rien faire, cette fonction appelante devrait transmettre une valeur spéciale au niveau supérieur et ainsi de suite ...

```
struct list * x1 = malloc(sizeof(struct list));
struct list * x2 = malloc(sizeof(struct list));
if (x1 == NULL || x2 == NULL) return NULL;
x1->hd = 1; x1->t1 = x2; x2->hd = 2; x2->t1 = NULL;
return x1;
```

Combien de programmeurs le font réellement ? Cela peut doubler la taille du source ? C'est pourtant le prix à payer pour la qualité.

8 / 64

Un exemple concret

Février 2014, bug concernant iOS et Mac OS X.

```
{
    ...
    if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
        goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
    goto fail; // <-----
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    ...
fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

9 / 64

Un autre exemple concret

Février 2014, bug concernant Linux.

Deux conventions sur la valeur de retour:

- ▶ une valeur négative signale une erreur et 0 un succès
- ▶ la valeur de retour est un booléen (0 est faux, tout autre entier est vrai)

10 / 64

Un autre exemple concret

```
static int /* Returns true or false, if the issuer is a CA, or not. */
check_if_ca (gnutls_x509_cert_t cert, gnutls_x509_cert_t issuer,
             unsigned int flags) {
    int result;
    result = _gnutls_x509_get_signed_data (issuer->cert, "tbsCertificate",
                                           &issuer_signed_data);

    if (result < 0) {
        gnutls_assert ();
        goto cleanup; }

    result = _gnutls_x509_get_signed_data (cert->cert, "tbsCertificate",
                                           &cert_signed_data);

    if (result < 0) {
        gnutls_assert ();
        goto cleanup;}
    // snip
    result = 0;

cleanup:
    // cleanup type stuff
    return result;
}
```

11 / 64

Un autre exemple concret, corrigé

```
static int
check_if_ca (gnutls_x509_cert_t cert, gnutls_x509_cert_t issuer,
             unsigned int flags) {
    int result;
    result = _gnutls_x509_get_signed_data (issuer->cert, "tbsCertificate",
                                           &issuer_signed_data);

    if (result < 0) {
        gnutls_assert ();
        goto fail; // CHANGED
    }
    // snip
    result = 1;

fail: // ADDED
    result = 0;

cleanup:
    // cleanup type stuff
    return result;
}
```

12 / 64

Un nouveau type: `exn`

La solution adoptée par CAML et d'autres (Java, Ada...)
Une exception ou "valeur exceptionnelle" est une valeur appartenant au type particulier `exn`.

```
Division_by_zero
```

```
- : exn = Division_by_zero
```

```
Failure "hd"
```

```
- : exn = Failure "hd"
```

```
Failure
```

```
Characters 0-7:
```

```
Failure;;  
~~~~~
```

```
Error: The constructor Failure expects 1 argument(s),  
but is applied here to 0 argument(s)
```

Ce type prédéfini est particulier car "extensible" : le programmeur peut y ajouter de nouveaux constructeurs.

13 / 64

Exception: déclarer

Déclarer une exception = ajouter un nouveau constructeur à `exn`.

```
exception Echec of string
```

```
exception Echec of string
```

```
Echec "pas de chance"
```

```
- : exn = Echec "pas de chance"
```

Une valeur exceptionnelle est donc une valeur.

14 / 64

Exception: lever

Lever une exception = appliquer `raise` à une exception.

```
raise (Echec "pas de chance")
```

Exception: Echec "pas de chance".

Ceci interrompt le programme en affichant l'exception. Sauf si ...

15 / 64

Exception: rattraper (1)

Rattraper une exception = utiliser l'expression

```
try <expr> with <filtrage>
```

16 / 64

Exception: rattraper (2)

La valeur de

```
try expr with
| m1 -> e1
| m2 -> e2
| ...
```

est la valeur de `expr` si aucune exception n'est déclenchée durant son évaluation.

Si l'exception `E` est levée durant l'évaluation, elle est alors filtrée par les motifs `m1`, `m2`, ...

- ▶ si `mi` est le premier motif acceptant `E`, la valeur rendue est la valeur de `ei`
- ▶ si aucun motif n'accepte `E`, l'exception continue à être levée

17 / 64

Exception: exemple (1)

```
exception Liste of string

let rec plgrd l = match l with
| [] -> raise (Liste "plgrd sur []")
| [a] -> a
| (a::r) -> max a (plgrd r)
```

```
exception Liste of string
val plgrd : 'a list -> 'a = <fun>
```

```
plgrd []
```

Exception: Liste "plgrd sur []".

```
try plgrd [] with
| (Liste "plgrd sur []") -> 0
```

```
- : int = 0
```

18 / 64

Exception: exemple (2)

Avantage des exceptions: ne pas avoir à manuellement tester et propager l'aspect exceptionnel à chaque appel de fonction. Le système propage l'exception tant qu'elle n'est pas rattrapée.

19 / 64

Exception: exercice (1)

Soit `ll` de type `int list list` et `p` un prédicat `int -> bool`. Définir une fonction qui prend dans chaque liste le premier entier vérifiant `p` et calcule leur produit.

Si dans une liste aucun entier ne vérifie `p` nous utiliserons 1.

```
mult (fun x -> x < 10) [[15;12;7;11;5]; [14]; [11;2]]
```

```
- : int = 14
```

20 / 64

Exception: exercice (2)

```
let rec cherch p l = match l with
| []      -> raise (Liste "cherch sur []")
| (a::r) -> if p a then a else cherch p r
```

```
val cherch : ('a -> bool) -> 'a list -> 'a = <fun>
```

```
let rec mult p l = match l with
| []      -> 1
| (l::rl) -> (try cherch p l with Liste "cherch sur []" -> 1)
              * (mult p rl)
```

```
val mult : (int -> bool) -> int list list -> int = <fun>
```

On récupère l'exception de cherch qui ne peut savoir quelle valeur rendre si elle ne trouve pas d'élément

21 / 64

Exception: failwith

- ▶ Failure est un constructeur prédéfini, d'arité 1, prenant une chaîne de caractères. Il peut souvent éviter de définir un nouveau constructeur
- ▶ failwith "zzz" est alors un raccourci pour raise (Failure "zzz")

22 / 64

Exception et efficacité (1)

Exemple final: produits des entiers d'une liste

```
let rec mult l = match l with
| []      -> 1
| 0::_   -> 0
| x::r1  -> x * mult r1
```

```
val mult : int list -> int = <fun>
```

Dès que l'on rencontre un 0 il est inutile de regarder les éléments suivants car on connaît résultat.

23 / 64

Exception et efficacité (2)

Mais non seulement il est inutile d'aller voir les éléments suivants, il est aussi inutile de réaliser les produits en attente.

```
exception Prod_zero
```

```
exception Prod_zero
```

```
let rec mul l = match l with
| []      -> 1
| 0::_   -> raise Prod_zero
| x::r1  -> x * mul r1
```

```
val mul : int list -> int = <fun>
```

```
let mult l = try mul l with Prod_zero -> 0
```

```
val mult : int list -> int = <fun>
```

24 / 64

Exception et efficacité (3)

Version 1

Dès que l'on trouve un 0, *on abandonne le parcours de la liste*, car on sait que les valeurs restantes sont sans influence sur le résultat, mais les produits en attente sont effectués.

Version 2

Dès que l'on trouve un 0, *on abandonne le parcours de la liste*, car on sait que les valeurs restantes sont sans influence sur le résultat, *et les produits en attente* sont eux aussi abandonnés.

Les entrées/sorties

Entrées / Sorties

Le système d'entrée/sortie de Ocaml adopte le style impératif : effet de bord, séquentialité.

Il est basé sur la notion de canal.

Deux types prédéfinis, `in_channel` et `out_channel`, dont la représentation est inaccessible, et une exception.

```
End_of_file
```

```
- : exn = End_of_file
```

27 / 64

Les canaux

les valeurs appartenant à ces types :

```
stdin
```

```
- : in_channel = <abstr>
```

```
stdout
```

```
- : out_channel = <abstr>
```

```
stderr
```

```
- : out_channel = <abstr>
```

ces canaux sont généralement associés au clavier et à l'écran

28 / 64

Ouverture des canaux

```
open_in
```

```
- : string -> in_channel = <fun>
```

```
open_out
```

```
- : string -> out_channel = <fun>
```

```
let mychin = open_in "../caml/lesnotes"
```

```
val mychin : in_channel = <abstr>
```

teste l'existence du fichier `lesnotes` du répertoire `caml` et rend le canal ou déclenche une exception

```
let mychout = open_out "../caml/moyennes"
```

```
val mychout : out_channel = <abstr>
```

crée ou recrée le fichier `moyennes` du répertoire `caml` et rend le canal ou déclenche une exception si les droits sont insuffisants

29 / 64

Fermeture des canaux

```
close_in
```

```
- : in_channel -> unit = <fun>
```

```
close_out
```

```
- : out_channel -> unit = <fun>
```

```
close_in mychin
```

```
- : unit = ()
```

```
close_out mychout
```

```
- : unit = ()
```

Comportement non spécifié si tentative de fermeture d'un canal standard ou d'un canal déjà fermé.

30 / 64

Les opérations en sortie

écriture sur canaux

```
output_char
```

```
- : out_channel -> char -> unit = <fun>
```

```
output_string
```

```
- : out_channel -> string -> unit = <fun>
```

```
output_value
```

```
- : out_channel -> 'a -> unit = <fun>
```

```
output
```

```
- : out_channel -> bytes -> int -> int -> unit = <fun>
```

`output oc s p l` écrit sur `oc` les `l` caractères de la chaîne `s` commençant à la position `p`

31 / 64

Les opérations en sortie

positionnement sur canaux

```
pos_out
```

```
- : out_channel -> int = <fun>
```

```
seek_out
```

```
- : out_channel -> int -> unit = <fun>
```

32 / 64

Les opérations en entrée

lecture sur canaux

```
input_char
```

```
- : in_channel -> char = <fun>
```

```
input_line
```

```
- : in_channel -> string = <fun>
```

```
input_value
```

```
- : in_channel -> 'a = <fun>
```

```
input
```

```
- : in_channel -> bytes -> int -> int -> int = <fun>
```

`input ic s p l` tente de lire sur `ic` un nombre `l` de caractères et les stocke dans la chaîne `s` commençant à partir de la position `p`.
Retourne le nombre de caractères effectivement lus.

33 / 64

Les opérations en entrée

positionnement sur canaux

```
pos_in
```

```
- : in_channel -> int = <fun>
```

```
seek_in
```

```
- : in_channel -> int -> unit = <fun>
```

34 / 64

Opérations sur les canaux standards

- ▶ **sortie standard**, pas de canal à ouvrir

```
print_char, print_string, print_int, print_float,  
print_endline, print_newline
```

- ▶ **entrée standard**, pas de canal à ouvrir

```
read_int, read_float, read_line
```

35 / 64

Exercice: entrées sorties clavier écran (1)

Exercice: lecture d'une liste d'entiers.

Le comportement souhaité est :

```
litliste()  
    Nombre d'éléments : 4  
    Entre tes 4 entiers :  
    4  
    5  
    4  
    7  
- : int list = [7; 4; 5; 4]
```

36 / 64

Exercice: entrées sorties clavier écran (2)

```
let litliste () =
  print_string "Nombre d'éléments : ";
  let nbel = read_int() in
  print_string "Entre tes "; print_int nbel;
  print_endline " entiers :";
  let rec litelem e = match e with
    | 0 -> []
    | n -> read_int() :: (litelem (n-1)) in
  litelem nbel
```

```
val litliste : unit -> int list = <fun>
```

Les éléments se retrouvent en ordre inverse dans la liste car l'opération `::` est faite après l'appel récursif.

37 / 64

Ordre des opérations

Il faut rester vigilant quant à l'ordre d'évaluation.

```
read_int() + read_int() * read_int()
  4
  7
  5
- : int = 33
```

Les `read_int` ne sont pas évalués dans l'ordre du texte mais dans l'ordre de l'évaluation. Or le produit est plus prioritaire que la somme.

38 / 64

Exercice: lecture fichier (1)

Exercice : affichage du fichier de nom ../caml/reussite.ml

```
let c_in = open_in "../caml/reussite.ml"
```

```
val c_in : in_channel = <abstr>
```

```
let rec litecr c =  
  print_char(input_char c);  
  try litecr c  
  with End_of_file -> ()
```

```
val litecr : in_channel -> unit = <fun>
```

```
litecr c_in
```

```
type couleur = Trefle | Carreau | Coeur | Pique  
type hauteur = As | Roi | Dame | Valet  
              | Dix | Neuf | Huit | Sept  
type carte   = Carte of hauteur * couleur  
- : unit = ()
```

39 / 64

Exercice: lecture fichier (2)

La version précédente pose problème : si le fichier est vide le premier `input_char` déclenche une exception non récupérée.

```
let rec litecr c =  
  try print_char(input_char c); litecr c  
  with End_of_file -> ()
```

```
val litecr : in_channel -> unit = <fun>
```

ou plus élégant

```
let litecr c =  
  let rec aux () = print_char (input_char c); aux () in  
  try aux () with End_of_file -> ()
```

```
val litecr : in_channel -> unit = <fun>
```

40 / 64

Boucles

Boucle for

```
for i = 1 to 5 do  
  print_endline (string_of_int i)  
done
```

```
1  
2  
3  
4  
5  
- : unit = ()
```

Boucle for

```
for i = 5 downto 1 do
  print_endline (string_of_int i)
done
```

```
5
4
3
2
1
- : unit = ()
```

43 / 64

Boucle while

```
let i = ref 0 in
while !i < 5 do
  print_endline (string_of_int (!i));
  i := !i + 1
done
```

```
0
1
2
3
4
- : unit = ()
```

44 / 64

Références et affectations

Enregistrements mutables

Le champ d'un enregistrement peut être déclaré **mutable**

```
type personne = { nom : string; mutable age : int }
```

```
type personne = { nom : string; mutable age : int; }
```

On crée les enregistrements comme d'habitude

```
let moi = { nom = "Alan"; age = 42 }
```

```
val moi : personne = {nom = "Alan"; age = 42}
```

Enregistrements mutables

On peut modifier les champs mutables (opérateur <-)

```
let anniversaire r = r.age <- r.age + 1
```

```
val anniversaire : personne -> unit = <fun>
```

```
anniversaire moi; moi
```

```
- : personne = {nom = "Alan"; age = 43}
```

On ne peut pas modifier les champs non mutables

```
moi.nom <- "Schmitt"
```

Characters 0-20:

```
moi.nom <- "Schmitt";;  
~~~~~
```

Error: The record field nom is not mutable

47 / 64

Les références

OCaml permet l'utilisation de valeurs qui sont des adresses en mémoire: ce sont les références. Leur type est 'a ref.

Si x est de type 'a ref, x est une valeur **référence** (un pointeur) sur une valeur référencée de type 'a

Constructeur:

```
ref
```

```
- : 'a -> 'a ref = <fun>
```

48 / 64

Les références

```
let x = ref (1,2)
```

```
val x : (int * int) ref = {contents = (1, 2)}
```

```
ref (fun x -> 2*x)
```

```
- : (int -> int) ref = {contents = <fun>}
```

Une référence n'est qu'un enregistrement à un seul champ, contents, qui est mutable

Une référence est toujours initialisée

49 / 64

Les références: accès

la fonction préfixe (!) permet d'accéder au contenu d'une référence

```
(!)
```

```
- : 'a ref -> 'a = <fun>
```

```
fst !x
```

```
- : int = 1
```

```
fst x
```

Characters 4-5:

```
fst x;;  
^
```

```
Error: This expression has type (int * int) ref  
      but an expression was expected of type 'a * 'b
```

50 / 64

Le retour des affectations

Et que fait-on avec ces données mutables ? Des affectations.

```
(:=)
```

```
- : 'a ref -> 'a -> unit = <fun>
```

```
let x = ref (1,2)
```

```
val x : (int * int) ref = {contents = (1, 2)}
```

```
x := (fst(!x)+1,3)
```

```
- : unit = ()
```

```
x
```

```
- : (int * int) ref = {contents = (2, 3)}
```

51 / 64

Tableaux

Les tableaux se notent

```
[|e1;e2;...;en|]
```

où les e_i sont des expressions du même type.
Ce sont des valeurs modifiables.

```
let a = [|11;3;5|]
```

```
val a : int array = [|11; 3; 5|]
```

52 / 64

Accès à un tableau: `Array.get`

Accès direct à la valeur d'un élément (2 possibilités):

1. `expr1.(expr2)` où `expr1` retourne un tableau, `expr2` retourne un entier (l'indice, commençant à 0)

```
[| 1; 2; 3 |].(1)
```

```
- : int = 2
```

2. par la fonction

```
Array.get
```

```
- : 'a array -> int -> 'a = <fun>
```

53 / 64

Modification de tableau: `Array.set`

Modification d'un élément (2 possibilités):

1. `expr1.(expr2) <- expr3` où `expr1` retourne un tableau, `expr2` retourne un entier (l'indice) et `expr3` retourne un élément de même type que les éléments du tableau

```
let a = [| 1; 2; 3 |] in a.(1) <- 42; a
```

```
- : int array = [|1; 42; 3|]
```

2. par la fonction

```
Array.set
```

```
- : 'a array -> int -> 'a -> unit = <fun>
```

54 / 64

Exemple : Quicksort

ALGORITHM 64

QUICKSORT

C. A. R. HOARE

Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
procedure quicksort (A,M,N); value M,N;  
           array A; integer M,N;  
comment Quicksort is a very fast and convenient method of  
sorting an array in the random-access store of a computer. The  
entire contents of the store may be sorted, since no extra space is  
required. The average number of comparisons made is  $2(M-N) \ln$   
 $(N-M)$ , and the average number of exchanges is one sixth this  
amount. Suitable refinements of this method will be desirable for  
its implementation on any actual computer;  
begin           integer I,J;  
                if M < N then begin partition (A,M,N,I,J);  
                    quicksort (A,M,J);  
                    quicksort (A, I, N)  
                end  
end           quicksort
```

55 / 64

Swap

On va travailler dans le tableau directement. Il nous faut un moyen d'échanger deux éléments

```
let swap a i j =  
  let tmp = a.(i) in  
  a.(i) <- a.(j);  
  a.(j) <- tmp
```

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

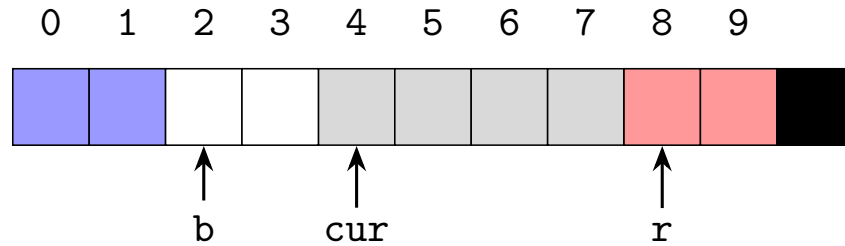
56 / 64

Interlude : le drapeau hollandais

On veut trier un tableau de trois valeurs Bleu, Blanc et Rouge de telle sorte que le Bleu soit en premier, puis le Blanc, puis le Rouge.

```
type couleur = Bleu | Blanc | Rouge
```

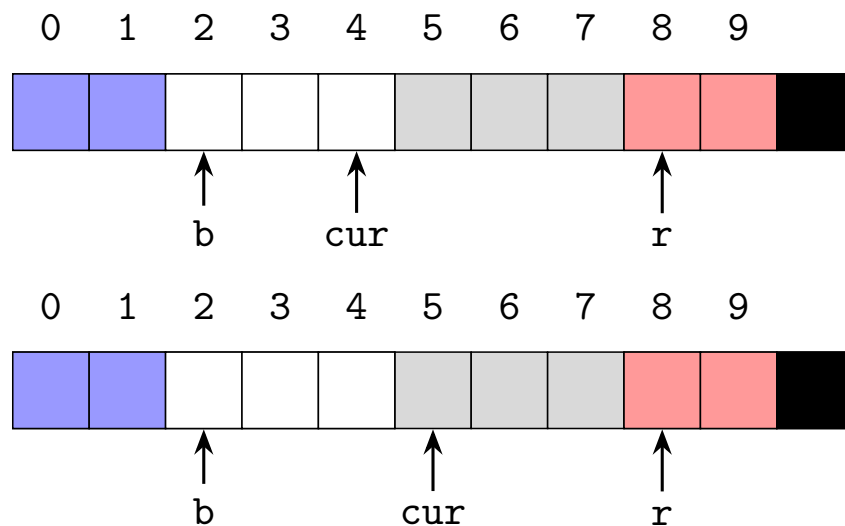
```
type couleur = Bleu | Blanc | Rouge
```



```
let rec flag a b cur r =  
  failwith "todo"
```

57 / 64

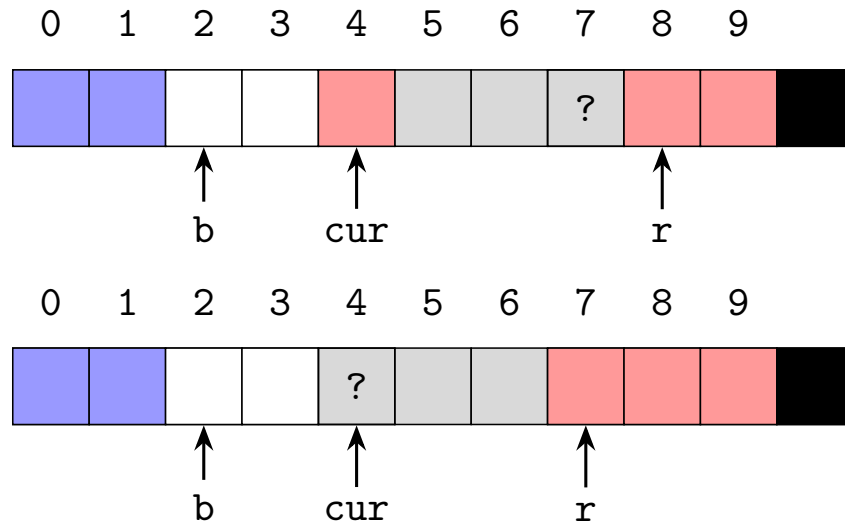
Interlude : le drapeau hollandais



```
let rec flag a b cur r =  
  match a.(cur) with  
  | Blanc -> flag a b (cur+1) r  
  ...
```

58 / 64

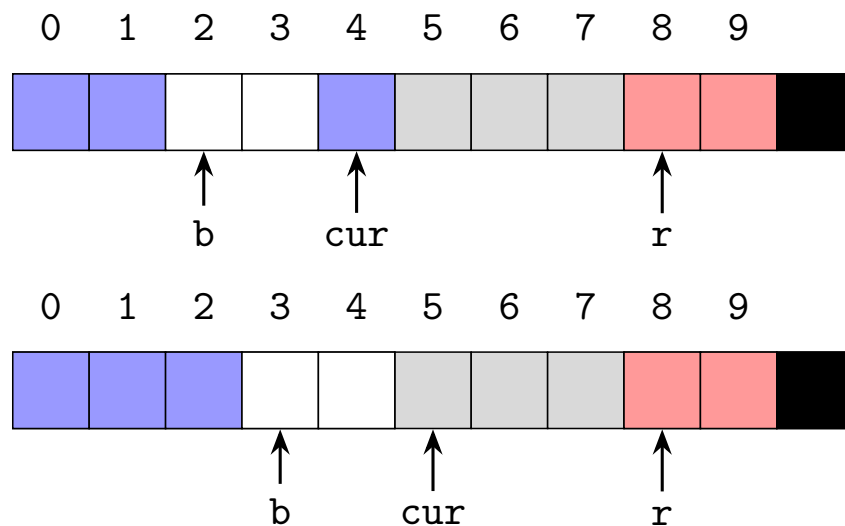
Interlude : le drapeau hollandais



```
let rec flag a b cur r =  
  match a.(cur) with  
  | Red -> swap a cur (r-1); flag a b cur (r-1)  
  ...
```

59 / 64

Interlude : le drapeau hollandais

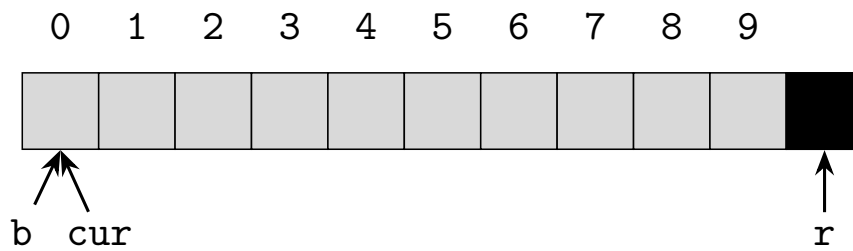


```
let rec flag a b cur r =  
  match a.(cur) with  
  | Blue -> swap a b cur; flag a (b+1) (cur+1) r  
  ...
```

60 / 64

Interlude : le drapeau hollandais

Début :



Fin : quand $cur = r$, retourner b et r

```
let flag a =
  let rec aux b cur r =
    if cur = r then b,r else
    match a.(cur) with
    | Bleu -> swap a b cur; aux (b+1) (cur+1) r
    | Blanc -> aux b (cur+1) r
    | Rouge -> swap a cur (r-1); aux b cur (r-1)
  in aux 0 0 (Array.length a)
```

```
val flag : couleur array -> int * int = <fun>
```

61 / 64

Partition

Pour partitionner autour du pivot v , on utilise le drapeau hollandais avec les valeurs plus petites que v , celles égales à v , et celles plus grandes.

```
let partition a m n =
  let v = a.(m) in
  let rec flag b i r =
    if i = r then b,r else
    let e = a.(i) in
    if e < v then begin swap a b i; flag (b+1) (i+1) r end
    else if e = v then flag b (i+1) r
    else begin swap a i (r-1); flag b i (r-1) end
  in
  flag m m n
```

```
val partition : 'a array -> int -> int -> int * int = <fun>
```

62 / 64

Quicksort

```
let quicksort a =  
  let rec qs a m n =  
    if m + 1 < n then begin  
      let b,r = partition a m n in  
      qs a m b;  
      qs a r n  
    end  
  in qs a 0 (Array.length a)
```

```
val quicksort : 'a array -> unit = <fun>
```

```
let res =  
  let a = [|4; 12; 27; -12; 7; 8; 1; 3; 6; 12; 42|] in  
  quicksort a; a
```

```
val res : int array = [| -12; 1; 3; 4; 6; 7; 8; 12; 12; 27; 42 |]
```

63 / 64

Pour le plaisir !

```
let rec qs l = match l with  
| [] -> []  
| x::xs -> (qs (List.filter (fun y -> y < x) xs)) @  
           (x :: (qs (List.filter (fun y -> y >= x) xs)))
```

```
val qs : 'a list -> 'a list = <fun>
```

Concis, mais gourmand.

64 / 64