PF chapître 7 : aspects impératifs

Jean-François Monin

POLYTECH GRENOBLE

Plan

Plan

Effets de bord

Exemples

- Entrées - sorties
- Modifications d'état (tableaux, mutables)
- Exceptions
- Flots

Effets de bord

Exemples

- Entrées - sorties
- Modifications d'état (tableaux, mutables)
- Exceptions
- Flots

Ce qui est observé dépend de l'ordre séquentiel des calculs

En fonctionnel : utilisation parcimonieuse !

Le type unit

Le type `unit` est un type somme à un seul élément noté `()`

Si on calcule une expression de type `unit`, la valeur obtenue ne nous apprend rien : elle est connue d'avance

Il n'y a donc pas d'information contenue dans une valeur de type `unit`

Subtilité

Quand un calcul est effectué, qu'a-t-on appris ?

- Information 1 : valeur du résultat
- Information 2 : le calcul est terminé

Avant le calcul que sait-on ?

- Le type du résultat
- Donc, pour un type somme : les différentes valeurs possibles

Pour le type unit

- On apprend simplement : "le calcul est fini"

Utilisation de `unit` en type de sortie

Type du résultat de fonctions purement à effet de bord

- écritures :
  `print_string` etc.
- modifications en place :
  `tab.(i) <- expr`
- clôtures de fichiers, de sockets, etc.

Rappel : évaluation d'un let

let $a = expr_1$ `in` $expr_2$

- évaluation de $expr_1$
- liaison de la valeur de $expr_1$ au nom a
- PUIS évaluation de $expr_2$
  (dans l'environnement augmenté de cette liaison)

Rappel : évaluation d'un let

let $a = expr_1$ `in` $expr_2$

- évaluation de $expr_1$
- liaison de la valeur de $expr_1$ au nom a
- PUIS évaluation de $expr_2$
  (dans l'environnement augmenté de cette liaison)

Dans le cas où

- $expr_2$ ne contient pas a
- on veut simplement évaluer successivement $expr_1$ et $expr_2$

il suffit d'écrire :

let _ = $expr_1$ `in` $expr_2$

Rappel : évaluation d'un let

let $a = expr_1$ `in` $expr_2$

- évaluation de $expr_1$
- liaison de la valeur de $expr_1$ au nom a
- PUIS évaluation de $expr_2$
  (dans l'environnement augmenté de cette liaison)

Dans le cas où

- $expr_2$ ne contient pas a
- on veut simplement évaluer successivement $expr_1$ et $expr_2$

il suffit d'écrire :

let _ = $expr_1$ `in` $expr_2$

Cas particulier intéressant : $expr_1$ est de type `unit`

Raccourci syntaxique

Notation

$expr_1$; $expr_2$
est une écriture simplifiée de
let () = $expr_1$ `in` $expr_2$

Exemple

```
let a = 3;;
let b =
  print_string "Bonjour";
  a+2;;

Bonjourval b : int = 5
```

Séquence

Syntaxe

$(e_1; e_2; \ldots; e_n)$ ou plutôt `begin` $e_1; e_2; \ldots; e_n$ `end`

Exemple

```
let x = 42
let main =
   print_string "Hello World! \n";
   print_int x;
   print_newline ()
```

Utilisation de `unit` en type d'entrée

Type du paramètre de fonctions de lecture

- lectures :
  `read_int` etc.
- ouvertures de fichiers, de sockets, etc.

Rappel

Le corps d'une fonction n'est évalué que lorsque cette fonction est appliquée à des arguments

Exemple

```
let n = read_int ()
```

Exercice

Écrire **une** expression qui pose une question, attend un entier au clavier et le stocke dans `x`.

Exercice

Écrire **une** expression qui pose une question, attend un entier au clavier et le stocke dans `x`.
Et si on veut un couple d'entiers ?

La fonction ignore

```
# 3;5 ;;
Warning : this expression should have type unit.
- : int = 5
```

La fonction ignore

```
# 3;5 ;;
Warning : this expression should have type unit.
- : int = 5
```

- Pour éviter cela on peut écrire
  `ignore 3 ; 5 ;;`
- Question subsidiaire : quel est le type de `ignore` ?

La fonction ignore

```
# 3;5 ;;
Warning : this expression should have type unit.
- : int = 5
```

- Pour éviter cela on peut écrire
  `ignore 3 ; 5 ;;`
- Question subsidiaire : quel est le type de `ignore` ?

```
# ignore;;
- : 'a -> unit = <fun>
```

Plan

Liste vs tableau

Écrire une fonction qui renvoie le *i*ème élément d'une liste (numérotée à partir de 0).

Liste vs tableau

Écrire une fonction qui renvoie le *i*ème élément d'une liste (numérotée à partir de 0).

```
let rec ieme = fun i l -> match l with
  | [] -> invalid_arg "Indice inexistant"
  | t :: q -> if i=0 then t else ieme (i-1) q;;
```

Écrire une fonction qui rend une liste dans laquelle le i^{eme} élément est remplacé par un élément donné.

Liste vs tableau

Écrire une fonction qui renvoie le *i*ème élément d'une liste (numérotée à partir de 0).

```
let rec ieme = fun i l -> match l with
  | [] -> invalid_arg "Indice inexistant"
  | t :: q -> if i=0 then t else ieme (i-1) q;;
```

Écrire une fonction qui rend une liste dans laquelle le i^{eme} élément est remplacé par un élément donné.

```
let rec remplace = fun x i l → match l with
  | [] -> invalid_arg "Indice inexistant"
  | t::q -> if i=0 then x::q else t::remplace x (i-1) q;;
```

Le type array

Construction

```
# let t = [| 12 ; 15 ; 0 ; 5 |];;
val t : int array = [|12; 15; 0; 5|]
# Array.make 3 'A';;
- : char array = [|'A'; 'A'; 'A'|]
```

Le type array

Construction

```
# let t = [| 12 ; 15 ; 0 ; 5 |];;
val t : int array = [|12; 15; 0; 5|]
# Array.make 3 'A';;
- : char array = [|'A'; 'A'; 'A'|]
```

Accès

```
# t.(1);;
- : int = 15
# t.(4);;
Exception: Invalid_argument "index out of bounds".
# Array.length t;;
- : int = 4
```

Le type array (suite)

Modification

```
# t;;
- : int array = [|12; 15; 0; 5|]
# t.(2) <- 10;;
- : unit = ()
# t;;
- : int array = [|12; 15; 10; 5|]
```

Redéfinition VS modification

Rappel : **let** x = A n'est pas l'affectation d'une variable.
C'est l'association d'un *identifiant* avec une *valeur* pour toute sa *portée*.

Redéfinition VS modification

Rappel : **let** x = A n'est pas l'affectation d'une variable.
C'est l'association d'un *identifiant* avec une *valeur* pour toute sa *portée*.

```
# let x = 0 in (let x = 1 in print_int x)
```

Redéfinition VS modification

Rappel : **let** *x* = A n'est pas l'affectation d'une variable.
C'est l'association d'un *identifiant* avec une *valeur* pour toute sa *portée*.

```
# let x = 0 in (let x = 1 in print_int x)
1- : unit = ()
```

Redéfinition VS modification

Rappel : **let** x = A n'est pas l'affectation d'une variable.
C'est l'association d'un *identifiant* avec une *valeur* pour toute sa *portée*.

```
# let x = 0 in (let x = 1 in print_int x)
1- : unit = ()
```

Redéfinition VS modification

Rappel : **let** x = A n'est pas l'affectation d'une variable.
C'est l'association d'un *identifiant* avec une *valeur* pour toute sa *portée*.

```
# let x = 0 in (let x = 1 in print_int x)
1- : unit = ()
```

Redéfinition VS modification

Rappel : **let** x = A n'est pas l'affectation d'une variable.
C'est l'association d'un *identifiant* avec une *valeur* pour toute sa *portée*.

```
# let x = 0 in (let x = 1 in print_int x)
1- : unit = ()
```

```
# let x = 0 in print_int (let x = x+1 in x+2) ; print_int x
```

Redéfinition VS modification

Rappel : **let** x = A n'est pas l'affectation d'une variable.
C'est l'association d'un *identifiant* avec une *valeur* pour toute sa *portée*.

```
# let x = 0 in (let x = 1 in print_int x)
1- : unit = ()
```

```
# let x = 0 in print_int (let x = x+1 in x+2) ; print_int x
30- : unit = ()
```

Redéfinition VS modification

Rappel : **let** x = A n'est pas l'affectation d'une variable.
C'est l'association d'un *identifiant* avec une *valeur* pour toute sa *portée*.

```
# let x = 0 in (let x = 1 in print_int x)
1- : unit = ()
```

```
# let x = 0 in print_int (let x = x+1 in x+2) ; print_int x
30- : unit = ()
```

Mais on ne peut pas écrire
let x = 0 ; **let** x = x + 1 ;;

Redéfinition VS modification

Rappel : **let** x = A n'est pas l'affectation d'une variable.
C'est l'association d'un *identifiant* avec une *valeur* pour toute sa *portée*.

```
# let x = 0 in (let x = 1 in print_int x)
1- : unit = ()
```

```
# let x = 0 in print_int (let x = x+1 in x+2) ; print_int x
30- : unit = ()
```

Mais on ne peut pas écrire
let x = 0 ; **let** x = x + 1 ;;

En revanche le symbole <- modifie l'identifiant à sa gauche si c'est autorisé.

Attention à l'aliasing

Une variable peut être modifiée sans être nommée explicitement.

```
# let t = [|1;2;3|];;
# let s = t;;
# t.(0) <- 42;;
# t;;
- : int array = [|42; 2; 3|]
# s;;
- : int array = [|42; 2; 3|]
```

Attention à l'aliasing

Une variable peut être modifiée sans être nommée explicitement.

```
# let t = [|1;2;3|];;
# let s = t;;
# t.(0) <- 42;;
# t;;
- : int array = [|42; 2; 3|]
# s;;
- : int array = [|42; 2; 3|]
```

Pire :

```
# let t = [|1;2;3|];;
# let l = [t];;
val l : int array list = [ [|1; 2; 3|] ]
# t.(0) <- 42;;
# l;;
- : int array list = [[|42; 2; 3|]]
```

Égalité logique vs physique

let x = 3 ;; **let** y = 3 ;;

```
x = y;;
```

Égalité logique vs physique

let x = 3 ;; **let** y = 3 ;;

```
x = y;;      - : bool = true
```

Égalité logique vs physique

let x = 3 ;; **let** y = 3 ;;

```
x = y;;      - : bool = true
x == y;;
```

Égalité logique vs physique

let x = 3 ;; **let** y = 3 ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = true
```

Égalité logique vs physique

let x = 3 ;; **let** y = 3 ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = true
```

let x = [| 1 ; 2 |] ;; **let** y = [| 1 ; 2 |] ;; **let** z = x ;;

```
x = y;;
```

Égalité logique vs physique

let x = 3 ;; **let** y = 3 ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = true
```

let x = [| 1 ; 2 |] ;; **let** y = [| 1 ; 2 |] ;; **let** z = x ;;

```
x = y;;      - : bool = true
```

Égalité logique vs physique

let x = 3 ;; **let** y = 3 ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = true
```

let x = [| 1 ; 2 |] ;; **let** y = [| 1 ; 2 |] ;; **let** z = x ;;

```
x = y;;      - : bool = true
x == y;;
```

Égalité logique vs physique

let x = 3 ;; **let** y = 3 ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = true
```

let x = [| 1 ; 2 |] ;; **let** y = [| 1 ; 2 |] ;; **let** z = x ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = false
```

Égalité logique vs physique

let x = 3 ;; **let** y = 3 ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = true
```

let x = [| 1 ; 2 |] ;; **let** y = [| 1 ; 2 |] ;; **let** z = x ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = false
x = z;;
```

Égalité logique vs physique

let x = 3 ;; **let** y = 3 ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = true
```

let x = [| 1 ; 2 |] ;; **let** y = [| 1 ; 2 |] ;; **let** z = x ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = false
x = z;;      - : bool = true
```

Égalité logique vs physique

let x = 3 ;; **let** y = 3 ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = true
```

let x = [| 1 ; 2 |] ;; **let** y = [| 1 ; 2 |] ;; **let** z = x ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = false
x = z;;      - : bool = true
x == z;;
```

Égalité logique vs physique

let x = 3 ;; **let** y = 3 ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = true
```

let x = [| 1 ; 2 |] ;; **let** y = [| 1 ; 2 |] ;; **let** z = x ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = false
x = z;;      - : bool = true
x == z;;     - : bool = true
```

Égalité logique vs physique

let x = 3 ;; **let** y = 3 ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = true
```

let x = [| 1 ; 2 |] ;; **let** y = [| 1 ; 2 |] ;; **let** z = x ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = false
x = z;;      - : bool = true
x == z;;     - : bool = true
```

let x = [| 1 ; 2 |] ;; **let** y = Array.copy x ;;

Égalité logique vs physique

let x = 3 ;; **let** y = 3 ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = true
```

let x = [| 1 ; 2 |] ;; **let** y = [| 1 ; 2 |] ;; **let** z = x ;;

```
x = y;;      - : bool = true
x == y;;     - : bool = false
x = z;;      - : bool = true
x == z;;     - : bool = true
```

let x = [| 1 ; 2 |] ;; **let** y = Array.copy x ;;

```
y = x;;      - : bool = true
y == x;;     - : bool = false
```

Tableau à 2 dimensions

Création et accès

```
# let x = Array.make_matrix 2 3 0;;
val x : int array array = [| [|0; 0; 0|]; [|0; 0; 0|] |]
```

Tableau à 2 dimensions

Création et accès

```
# let x = Array.make_matrix 2 3 0;;
val x : int array array = [| [|0; 0; 0|]; [|0; 0; 0|] |]
# x.(0).(0) <- 1;;
```

Tableau à 2 dimensions

Création et accès

```
# let x = Array.make_matrix 2 3 0;;
val x : int array array = [| [|0; 0; 0|]; [|0; 0; 0|] |]
# x.(0).(0) <- 1;;
# x;;
- : int array array = [| [|1; 0; 0|]; [|0; 0; 0|] |]
```

Tableau à 2 dimensions

Création et accès

```
# let x = Array.make_matrix 2 3 0;;
val x : int array array = [| [|0; 0; 0|]; [|0; 0; 0|] |]
# x.(0).(0) <- 1;;
# x;;
- : int array array = [| [|1; 0; 0|]; [|0; 0; 0|] |]
```

Attention !!

```
# let y = Array.make 2 (Array.make 3 0);;
val y : int array array = [| [|0; 0; 0|]; [|0; 0; 0|] |]
```

Tableau à 2 dimensions

Création et accès

```
# let x = Array.make_matrix 2 3 0;;
val x : int array array = [| [|0; 0; 0|]; [|0; 0; 0|] |]
# x.(0).(0) <- 1;;
# x;;
- : int array array = [| [|1; 0; 0|]; [|0; 0; 0|] |]
```

Attention !!

```
# let y = Array.make 2 (Array.make 3 0);;
val y : int array array = [| [|0; 0; 0|]; [|0; 0; 0|] |]
# y.(0).(0) <- 1;;
```

Tableau à 2 dimensions

Création et accès

```
# let x = Array.make_matrix 2 3 0;;
val x : int array array = [| [|0; 0; 0|]; [|0; 0; 0|] |]
# x.(0).(0) <- 1;;
# x;;
- : int array array = [| [|1; 0; 0|]; [|0; 0; 0|] |]
```

Attention !!

```
# let y = Array.make 2 (Array.make 3 0);;
val y : int array array = [| [|0; 0; 0|]; [|0; 0; 0|] |]
# y.(0).(0) <- 1;;
# y;;
- : int array array = [| [|1; 0; 0|]; [|1; 0; 0|] |]
```

Plan

Plan

Entrées / sorties standard

Sortie

`print_string` affiche une chaîne de caractères passée en argument.
Valeur de retour ?

Entrées / sorties standard

Sortie

`print_string` affiche une chaîne de caractères passée en argument.
Valeur de retour ?

```
# print_string "hello" ;;
hello- : unit = ()
```

unit est le type/la valeur de retour des fonctions

- qui ne renvoient rien
- mais qui peuvent avoir des effets de bord

Entrées / sorties standard

Entrée

`read_line` lit une ligne au clavier et renvoie la chaîne lue.
Quel est son type ?

Entrées / sorties standard

Entrée

`read_line` lit une ligne au clavier et renvoie la chaîne lue.
Quel est son type ?

- Un identifiant = une valeur, jamais réévalué

```
# read_line;;
- : unit -> string = <fun>
```

Entrées / sorties standard

Entrée

`read_line` lit une ligne au clavier et renvoie la chaîne lue.
Quel est son type ?

- Un identifiant = une valeur, jamais réévalué

```
# read_line;;
- : unit -> string = <fun>
```

- Seules sont évaluées les fonctions appliquées à un argument

```
# read_line ();;
hello
- : string = "hello"
```

Entrées / sorties standard

Entrée

`read_line` lit une ligne au clavier et renvoie la chaîne lue.
Quel est son type ?

- Un identifiant = une valeur, jamais réévalué

```
# read_line;;
- : unit -> string = <fun>
```

- Seules sont évaluées les fonctions appliquées à un argument

```
# read_line ();;
hello
- : string = "hello"
```

unit est le type/la valeur de l'argument des fonctions

- qui n'ont besoin d'aucun argument
- mais qui doivent être réévaluées à chaque fois

Lecture / écriture dans un fichier

Ouverture d'un fichier, lecture et fermeture

```
open_in : string -> in_channel
```

```
close_in : in_channel -> unit
```

```
# let cin = open_in ``nomdufichier''
# let sin = Stream.of_channel cin
  ⋮     (lecture et analyse)
# let () = close_in cin
```

Lecture / écriture dans un fichier

Ouverture d'un fichier, lecture et fermeture

```
open_in : string -> in_channel
```

```
close_in : in_channel -> unit
```

```
# let cin = open_in ``nomdufichier''
# let sin = Stream.of_channel cin
  ⋮     (lecture et analyse)
# let () = close_in cin
```

Écriture dans un fichier : utilisation similaire de

```
open_out : string -> out_channel
close_out : out_channel -> unit
```

et de fonctions comme

```
output_char : out_channel -> char -> unit
```

Plan

Notions d'enregistrements

```
# type personne =  prenom : string ; nom : string ;
                    age : int ;;
# let quelquun =  prenom = "Bob" ; nom = "Dylan" ;
                   age = 77 ;;

#  quelquun with age = quelquun.age + 1  ;;
- : personne = prenom = "Bob"; nom = "Dylan";
               age = 78

# quelquun ;;
- : personne = prenom = "Bob"; nom = "Dylan";
               age = 77

# quelquun.age <- quelquun.age + 1;;
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: The record field age is not mutable
```

Plus général : les enregistrements mutables

```
# type personne = { prenom : string ; nom : string ;
                    mutable age : int} ;;
# let quelquun = { prenom = "Bob" ; nom = "Dylan" ;
                   age = 43 } ;;

# quelquun.age <- quelquun.age + 1;;
- : unit = ()

# quelquun;;
- : personne = { prenom = "Bob" ; nom = "Dylan" ;
                  age = 44 }

# quelquun.prenom <- "Robert";;
  ^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error: The record field prenom is not mutable
```

Avec un peu de polymorphisme

```
# type 'a ref = {mutable contents : 'a};;
# let n = {contents = 0};;
val n : int ref = {contents = 0}
```

Avec un peu de polymorphisme

```
# type 'a ref = {mutable contents : 'a};;
# let n = {contents = 0};;
val n : int ref = {contents = 0}
# n.contents <- 1;;
- : unit = ()
# n;;
val n : int ref = { contents = 1 }
# n.contents;;
- : int = 1
```

Avec un peu de polymorphisme

```
# type 'a ref = {mutable contents : 'a};;
# let n = {contents = 0};;
val n : int ref = {contents = 0}
# n.contents <- 1;;
- : unit = ()
# n;;
val n : int ref = { contents = 1 }
# n.contents;;
- : int = 1
```

Le type ref est prédéfini en OCaml.

- `ref x` est un raccourci pour `{contents = x}`
- `!r` est un raccourci pour `r.contents`
- `r := v` est un raccourci pour `r.contents <- v`

Avec un peu de polymorphisme

```
# type 'a ref = {mutable contents : 'a};;
# let n = {contents = 0};;          # let n = ref 0;;
val n : int ref = {contents = 0}
# n.contents <- 1;;                 # n := 1;;
- : unit = ()
# n;;
val n : int ref = { contents = 1 }
# n.contents;;                      # !n ;;
- : int = 1
```

Le type ref est prédéfini en OCaml.

- `ref x` est un raccourci pour `{contents = x}`
- `!r` est un raccourci pour `r.contents`
- `r := v` est un raccourci pour `r.contents <- v`

Exemples

- Incrémenter r ?

Exemples

- Incrémenter r ?

```
let r = ref 5;;
val r : int ref = {contents = 5}
r := !r + 1;;
- : unit = ()
!r;;
- : int = 6
```

- Calculer la longueur d'une liste avec une référence ?

Exemples

- Incrémenter r ?

```
let r = ref 5;;
val r : int ref = {contents = 5}
r := !r + 1;;
- : unit = ()
!r;;
- : int = 6
```

- Calculer la longueur d'une liste avec une référence ?

```
let lg = ref 0;;
let rec len l = match l with
  | [] -> !lg
  | _ :: q -> lg := !lg + 1 ; len q;;
```

Exemples

- Incrémenter r ?

```
let r = ref 5;;
val r : int ref = {contents = 5}
r := !r + 1;;
- : unit = ()
!r;;
- : int = 6
```

- Calculer la longueur d'une liste avec une référence ?

```
let lg = ref 0;;
let rec len l = match l with
  | [] -> !lg
  | _ :: q -> lg := !lg + 1 ; len q;;
```

Mais `lg` n'est pas remis à 0 à chaque appel !

Compteur

Écrire une fonction `compteur` de type : `unit -> int` qui renvoie 1 au premier appel, 2 au second, puis l'entier suivant à chaque appel.

Compteur

Écrire une fonction `compteur` de type : `unit -> int` qui renvoie 1 au premier appel, 2 au second, puis l'entier suivant à chaque appel.

```
# let c = ref 0;;
# let compteur () = c := !c + 1 ; !c;;
val compteur : unit -> int = <fun>
```

Écrire une fonction `raz` qui remet le compteur à zéro.
Quel est son type ?

Compteur

Écrire une fonction `compteur` de type : `unit -> int` qui renvoie 1 au premier appel, 2 au second, puis l'entier suivant à chaque appel.

```
# let c = ref 0;;
# let compteur () = c := !c + 1 ; !c;;
val compteur : unit -> int = <fun>
```

Écrire une fonction `raz` qui remet le compteur à zéro.
Quel est son type ?

```
# let raz () = c := 0;;
val raz : unit -> unit = <fun>
```

Que peut-on reprocher à ce compteur ?

Aliasing pour les références

Les égalités

```
# let r = ref 0;;
# let q = ref 0;;
```

Que vaut q = r ?

Aliasing pour les références

Les égalités

```
# let r = ref 0;;
# let q = ref 0;;
```

Que vaut q = r ?

true, car même contenu

Aliasing pour les références

Les égalités

```
# let r = ref 0;;
# let q = ref 0;;
```

Que vaut q = r ?

true, car même contenu

Que vaut q == r ?

Aliasing pour les références

Les égalités

```
# let r = ref 0;;
# let q = ref 0;;
```

Que vaut q = r ?

true, car même contenu

Que vaut q == r ?

false, car pas la même "adresse"

Aliasing pour les références

Les égalités

```
# let r = ref 0;;
# let q = ref 0;;
```

Que vaut q = r ?

true, car même contenu

Que vaut q == r ?

false, car pas la même "adresse"

Aliasing

```
# let r = ref 0;;
# let q = r;
```

Même objet, donc même contenu : true dans les deux cas