

~~Programmation fonctionnelle avec Objective Caml~~

4^{ème} séance, 26 mars 2015

Programmation impérative

samuel.hornus@inria.fr

<http://www.loria.fr/~shornus/ocaml/>

Les effets de bord

Une expression peut avoir des effets de bords :

Les effets de bord

Une expression peut avoir des **effets de bords** :

- modifier une zone mémoire accessible par d'autres fonctions :
 - variables globale
 - pointeurs partagés
 - 'a ref passée en argument
 - etc.

Les effets de bord

Une expression peut avoir des **effets de bords** :

- modifier une zone mémoire accessible par d'autres fonctions :
 - variables globale
 - pointeurs partagés
 - 'a ref passée en argument
 - etc.

- écrire ou lire des données sur
 - l'écran
 - l'imprimante
 - `std{in|out|err}`
 - la webcam
 - etc.

Les effets de bord

Une expression peut avoir des **effets de bords** :

effet de bord (ou effet ou side effect)

≈

tout ce qui est susceptible de modifier
le comportement futur de la fonction ou
d'autres fonctions.

Les effets de bord

```
Exemple  let r = ref 0;;
          let compteur =
            function () ->
              let () = r := !r + 1 in !r;;

# compteur ();;
- : int = 1

# compteur ();;
- : int = 2

# compteur ();;
- : int = 3

# r := 1983; compteur ();;
- : int = 1984
```

Les effets de bord

```
Exemple  let r = ref 0;;  
         let compteur =  
           function () ->  
             let () = r := !r + 1 in !r;;
```

```
# compteur ();;
```

```
- : int = 1
```

```
# compteur ();;
```

```
- : int = 2
```

```
# compteur ();;
```

```
- : int = 3
```

```
# r := 1983; compteur ();;
```

BIM BAM BOUM

On a tout cassé

```
- : int = 1984
```

Les effets de bord

```
Exemple  let compteur =  
          let r = ref 0 in  
          function () ->  
            let () = r := !r + 1 in !r;;
```

```
# compteur ();;
```

```
# compteur ();;
```

```
# compteur ();;
```

```
# compteur ();;
```

Ici, la solution est de
cacher la référence
dans la portée de
compteur.

```
- : int = 1
```

```
- : int = 2
```

```
- : int = 3
```

```
- : int = 4
```


Dans un monde sans effet de bord...

ou encore... dans un monde purement fonctionnel...

- on peut évaluer les fonctions (et leurs arguments) dans n'importe quel ordre : `let a = f x + g x + h (z x)` : le résultat sera toujours le même.
- si `f x` est très long à calculer, le compilateur peut stocker le résultat pour le réutiliser plus tard.

Dans un monde sans effet de bord...

ou encore... dans un monde purement fonctionnel...

- on peut évaluer les fonctions (et leurs arguments) dans n'importe quel ordre : `let a = f x + g x + h (z x)` : le résultats sera toujours le même.
- si `f x` est très long à calculer, le compilateur peut stocker le résultat pour le réutiliser plus tard.
- simplifie la compréhension (humaine et automatique) d'un programme.
- ouvre plein de possibilité pour l'optimisation du code.
- réduit fortement la présence de bug.

Dans un monde sans effet de bord...

ou encore... dans un monde purement fonctionnel...

- on peut évaluer les fonctions (et leurs arguments) dans n'importe quel ordre : `let a = f x + g x + h (z x)` : le résultat sera toujours le même.
- si `f x` est très long à calculer, le compilateur peut stocker le résultat pour le réutiliser plus tard.
- simplifie la compréhension (humaine et automatique) d'un programme.
- ouvre plein de possibilités pour l'optimisation du code.
- réduit fortement la présence de bug.

...mais on a quand besoin des effets de bords de temps en temps... (au moins pour afficher un résultat !)

Les effets de bords dans OCaml

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

`Printf.printf`, ...

`read_int`, `read_line`, `read_float`, ...

- Les opérations sur les fichiers

- Les références :

```
let i = ref 0
```

```
let f x = begin i := !i + 1; x+ !i end
```

```
# f 10;;
```

```
- : int = 11
```

```
# f 10;;
```

```
- : int = 12
```

- *etc.*

Les effets de bords dans OCaml

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

`Printf.printf`, ...

`read_int`, `read_line`, `read_float`, ...

- Les opérations sur les fichiers

- Les références :

```
let i = ref 0
```

```
let f x = begin i := !i + 1; x+ !i end
```

```
# f 10;;
```

```
- : int = 11
```

```
# f 10;;
```

f est un fonction à effet de bord

```
- : int = 12
```

- *etc.*

Le temps entre en piste

Lorsque des effets de bords sont présents, l'ordre d'évaluation d'une expression / des arguments d'une fonction, **peut influencer sur le résultat**. On **doit** prendre en compte l'aspect **temporel** de l'exécution du programme. C'est une **complication supplémentaire**.

Le temps entre en piste

Lorsque des effets de bords sont présents, l'ordre d'évaluation d'une expression / des arguments d'une fonction, **peut influencer sur le résultat**. On **doit** prendre en compte l'aspect **temporel** de l'exécution du programme. C'est une **complication supplémentaire**.

```
let f x = print_int x; x
```

```
# f 1 + f 2 + f 3 + f 4 + f 5 + f 6;;
```

```
654321- : int = 21
```

Les séquences

$expr ::= expr_1 ; expr_2$

$expr ::= (expr)$

$expr ::= \text{begin } expr \text{ end}$

Les séquences

$expr ::= expr_1 ; expr_2$

$expr ::= (expr)$

$expr ::= begin\ expr\ end$

Quand $expr$ est évaluée :

1. $expr_1$ est évaluée en premier.
2. $expr_2$ est évaluée ensuite.
3. la valeur de $expr$ est celle de $expr_2$ (la dernière valeur de la séquence)

Les séquences

```
expr ::= expr_1 ; expr_2
```

```
expr ::= ( expr )
```

```
expr ::= begin expr end
```

Une séquence **ordonne**
l'évaluation de
plusieurs expressions
dans le temps

```
# f 1 ; f 2 ; f 3;;
```

```
Warning 10: this expression should have type unit.
```

```
Warning 10: this expression should have type unit.
```

```
123- : int = 3
```

Les séquences

```
expr ::= expr_1 ; expr_2
```

```
expr ::= ( expr )
```

```
expr ::= begin expr end
```

Une séquence **ordonne**
l'évaluation de
plusieurs expressions
dans le temps

```
# f 1 ; f 2 ; f 3;;
```

```
Warning 10: this expression should have type unit.
```

```
Warning 10: this expression should have type unit.
```

```
123- : int = 3
```

```
# let g x = print_int x;;
```

```
# g 1 ; g 2 ; f 3;;
```

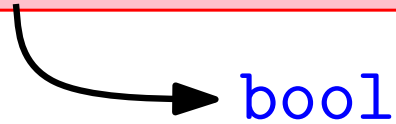
```
123- : int = 3
```

```
# begin g 100 ; g 200 ; f 300 end;;
```

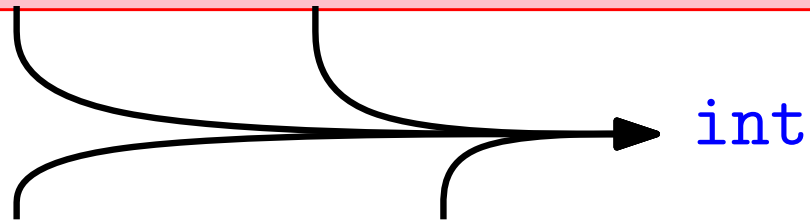
```
100200300- : int = 300
```

Séquences répétitives : les boucles

`expr ::= while expr1 do expr2 done`



`expr ::= for name = expr1 to expr2 do expr3 done`



`expr ::= for name = expr1 downto expr2 do expr3 done`

Quel est le type de `expr` ?

Séquences répétitives : les boucles

`expr ::= while expr1 do expr2 done`

`bool`

`expr ::= for name = expr1 to expr2 do expr3 done`

`int`

`expr ::= for name = expr1 downto expr2 do expr3 done`

`expr : unit = ()`

Les références

On les a déjà vu...

```
# let upu = ref (1+1) and d = 30;;  
      val upu : int ref = {contents = 2}
```

```
val d : int = 30
```

```
# !upu;;
```

```
- : int = 2
```

```
# upu := d + 10 + !upu; !upu;;
```

```
- : int = 42
```

Deux fonctions utiles du module `Pervasives` :

```
decr : int ref -> unit
```

```
incr : int ref -> unit
```

```
# decr r; !upu;;
```

```
- : int = 41
```

Les tableaux

```
# let tab = [|1; 2+3; !upu|];;
```

```
val tab : int array = [|1; 5; 41|]
```

```
# Array.make 10 0.0;;
```

```
- : float array = [|0.; 0.; 0.; 0.; 0.; 0.; 0.; 0.; 0.; 0.|]
```

```
# tab.(1) <- 2011;;
```

les cases d'un tableau sont mutables

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

```
# [tab.(2); tab.(1); tab.(0)];;
```

```
- : int list = [41; 2011; 1]
```

Les tableaux

```
# let tab = [|1; 2+3; !upu|];;
                                val tab : int array = [|1; 5; 41|]

# Array.make 10 0.0;;
- : float array = [|0.; 0.; 0.; 0.; 0.; 0.; 0.; 0.; 0.; 0.|]

# tab.(1) <- 2011;;
                                les cases d'un tableau sont mutables
                                - : unit = ()

# [tab.(2); tab.(1); tab.(0)];;
                                - : int list = [41; 2011; 1]
```

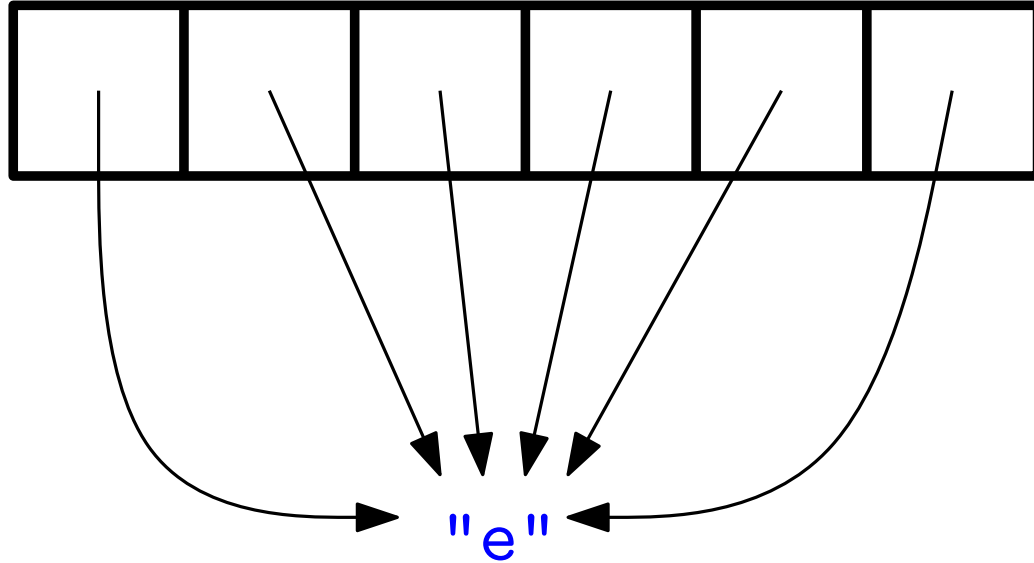
Voir la doc du module [Array](#)

<http://caml.inria.fr/pub/docs/manual-ocaml/libref/Array.html>

Partage dans les tableaux

```
let tab = Array.make 6 "e"
```

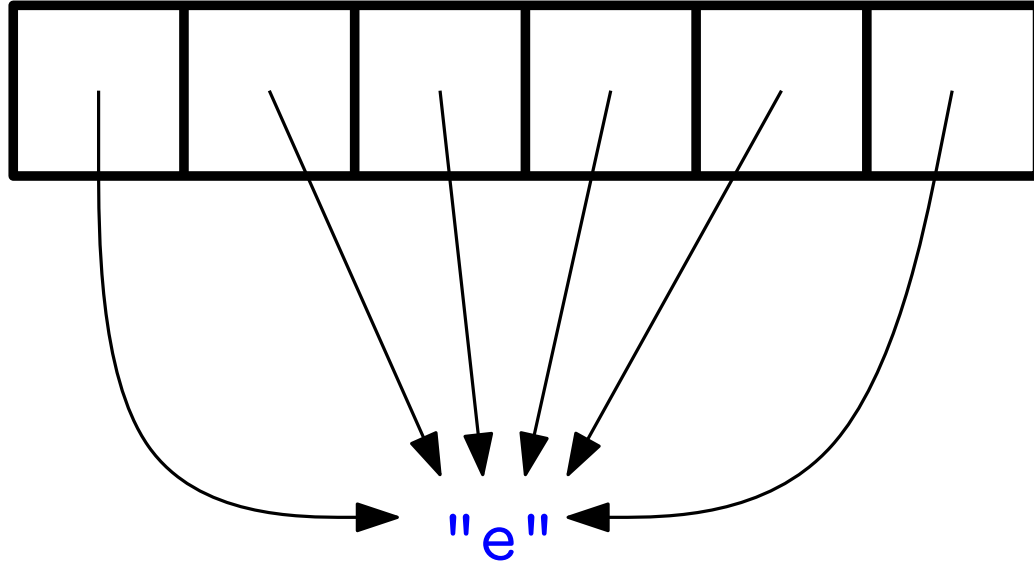
tab =



Partage dans les tableaux

```
let tab = Array.make 6 "e"
```

tab =

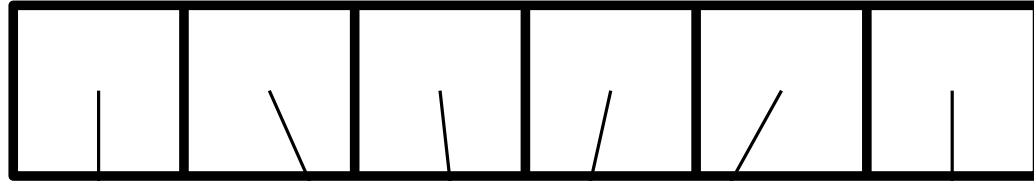


```
tab.(5) <- "z"
```

Partage dans les tableaux

```
let tab = Array.make 6 "e"
```

tab =



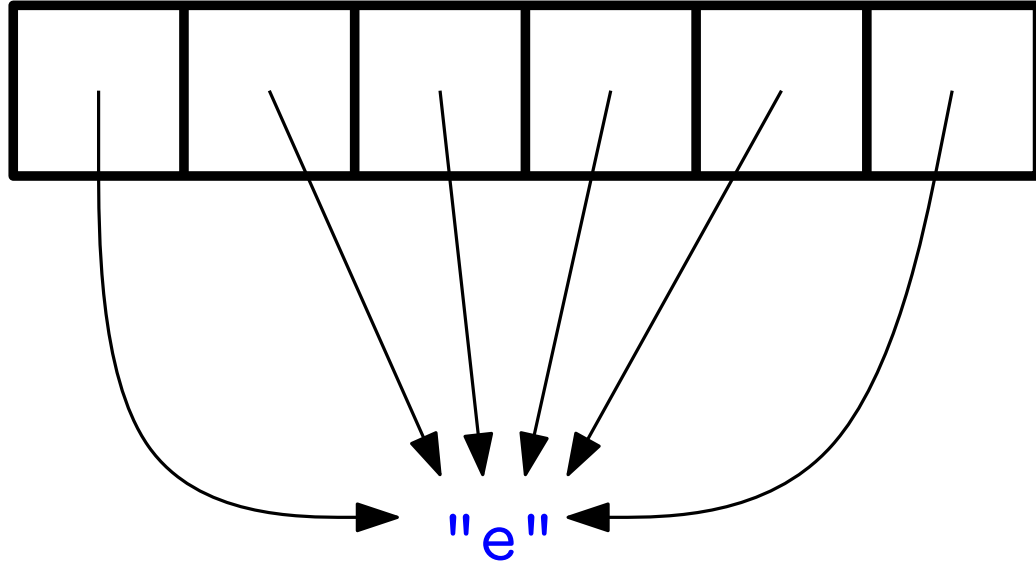
```
tab.(5) <- "z"
```

"z"

Partage dans les tableaux

```
let tab = Array.make 6 "e"
```

tab =

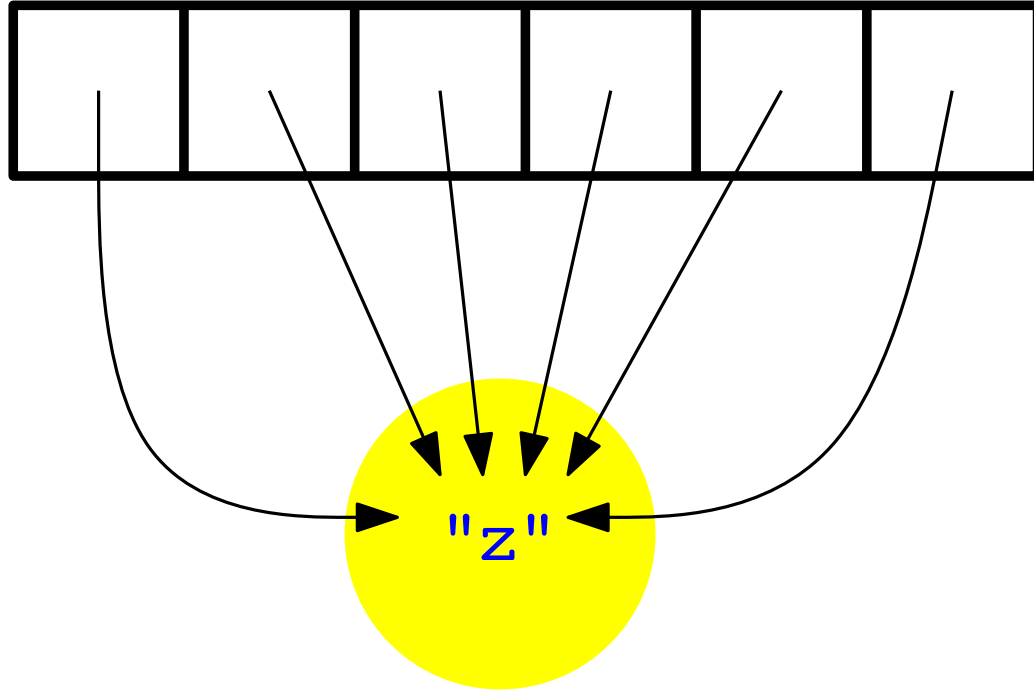


```
tab.(5).[0] <- 'z'
```

Partage dans les tableaux

```
let tab = Array.make 6 "e"
```

tab =



```
tab.(5).[0] <- 'z'
```

... et le tableau ne contient plus que des "z"

Partage dans les tableaux

Pour avoir des chaînes^a différentes avec le même contenu, voici une solution **possible** :

```
# for i = 0 to 5 do
  tab.(i) <- String.make 1 'z'
done; tab;;
      - : string array = [|"z";"z";"z";"z";"z";"z"|]

# tab.(5).[0] <- 'X'; tab;;
      - : string array = [|"z"; "z"; "z"; "z"; "z"; "X"|]
```

a. ou quelque-chose de *mutable*

Les chaînes de caractères (*string*)

Comme pour les tableaux : les chaînes de caractères sont mutables.

La syntaxe est un peu différente ; [] au lieu de () :

```
let nom = "Hornus"
```

```
# nom.[3];;
```

```
- : char = 'n'
```

```
# nom.[0] <- 'C'; nom;;
```

```
- : string = "Cornus"
```

Les champs mutables (=modifiables)

Retour sur les enregistrements

```
type cercle =
{
  centre : float * float;
  rayon : int
}
# let c1 = { centre = (1.0, 2.0); rayon = 3 };;
      val c1 : cercle = {centre = (1., 2.); rayon = 3}
# let c2 = {c1 with centre = (4.0, 3.0)};;
      val c2 : cercle = {centre = (4., 3.); rayon = 3}
# let aire c = let pi = 4.0 *. atan 1.0
               and r = float c.rayon
               in pi *. r *. r;;
      val aire : cercle -> float = <fun>
# aire c1;;
      - : float = 28.274333882308138
```


Les champs mutables (=modifiables)

Mutabilité : On peut déclarer qu'un champs est *mutable* (modifiable).

Les champs mutables (=modifiables)

Mutabilité : On peut déclarer qu'un champs est *mutable* (modifiable).

```
type cercle =  
{  
  centre : float * float;  
  mutable rayon : int  
}
```

Les champs mutables (=modifiables)

Mutabilité : On peut déclarer qu'un champs est *mutable* (modifiable).

```
type cercle =
{
  centre : float * float;
  mutable rayon : int
}
# c2;;
- : cercle = {centre = (4., 3.); rayon = 3}
# c2.rayon <- 25;;
- : unit = ()
# c2;;
- : cercle = {centre = (4., 3.); rayon = 25}
# c2.centre <- (0.0, 0.0);;
Error: The record field label centre is not mutable
```

Retour sur les références

Les références permettent d'introduire facilement une variable mutable. Voici leur implémentation :

```
type 'a ref = { mutable contenu : 'a }  
# let ref x = { contenu = x };;  
                                val ref : 'a -> 'a ref = <fun>  
# let (!) r = r.contenu;;  
                                val (!) : 'a ref -> 'a = <fun>  
# let (:=) r v = r.contenu <- v;;  
                                val (:=) : 'a ref -> 'a -> unit = <fun>
```

Fin du cours