Ocaml: Un langage fonctionnel
impératif / fonctionnel

Programmation "impérative" ou "procédurale"

- Approche par le bas: abstraction de l'architecture de l'ordinateur vers un langage de haut niveau
  - depuis le langage machine, l'assembleur, le macro-assembleur, FORTRAN, Pascal, C, Java, etc …
- Etat: représente l'ensemble des variables
- Exécution:
  - état initial
  - suite finie d'instructions d'affectation
  - chacune modifie l'état courant
  - exécutées séquentiellement
  - elles peuvent être itérées par des boucles "while"
  - elles peuvent être exécutées conditionnellement à l'aide de "if ... then ... else".

Ocaml: Un langage fonctionnel
impératif / fonctionnel

Programmation “fonctionnelle”

- Approche par le haut: le langage est conçu comme un outil pour spécifier les algorithmes
- Pas d'état
- Pas d'affectation
- Pas d'instruction
- Pas de boucle
- Un programme : une expression
- Exécution d'un programme: évaluation de l'expression

Ocaml: Un langage fonctionnel
impératif / fonctionnel

Exemple

- Impératif:

```
int  square (int n) {
        int s;
        s = n*n;
        return(s); }
```

- Fonctionnel

```
let square = fun x -> x*x;;
```

```
let square x = x*x;;
```

Ocaml: Un langage fonctionnel

Avant toutes choses

Lancer Objective Caml: commande ocaml

```
$
$ ocaml
#
```

Sortir de Objective Caml: taper #quit;; ou ctrl D

```
#
##quit;;
$
```

Ocaml: Un langage fonctionnel

Pas de différence entre fonction / objet simple

Une fonction est une expression et à ce titre peut être elle-même l'argument ou le résultat d'une autre fonction (ordre supérieur).

Récursivité: remplace les boucles

Les expressions

Programme = Expression

Définies par une syntaxe et une sémantique

- Syntaxe concrète: les caractères ASCII composant l'expression
- Syntaxe abstraite: la structure profonde, l'arbre syntaxique

Possèdent un type (ensemble de valeurs) et une valeur

- Sémantique statique: le type de l'expression
- Sémantique dynamique: la valeur de l'expression

Adéquation entre les 2 sémantiques: le type de l'expression est identique à celui de sa valeur

Les expressions

Exécution = Evaluation

Résultat d'une évaluation :

val *nom* : *type* = *valeur*

\- : *type* = *valeur*

Exemples :

```
# 4/(2*2);;
   - : int = 1
# fun x->x*x;;
   - : int -> int = <fun>
```

Les expressions
Evaluation

Evaluation en 3 temps

analyse syntaxique

production de l'arbre syntaxique si syntaxe correcte

2+3 et 2 + “oiseau” sont syntaxiquement correctes

2+ ne l'est pas

analyse statique

production du type si l'expression est typable

2+3 est typable, de type int 2 + ”oiseau” ne l'est pas

évaluation proprement dite

production de la valeur si le programme ne boucle pas

Les expressions
Définitions

Nommer une expression

let *nom* = *expr*

```
#let x = 2*5+1;;
   val x : int = 11

#let pi = 3.1416;;
   val pi : float = 3.1416

#let square = fun x ->x*x;;
   val square : int -> int = <fun>

#let square x = x*x;;
   val square : int -> int = <fun>
```

Les expressions
Définitions

Nommer une expression

let *nom* = *expr*

```
#let x = 2*5+1;;
   val x : int = 11

#x;;
   - : int = 11

#let square = fun x ->x*x;;
   val square : int -> int = <fun>

#square;;
   - : int -> int = <fun>
```

Les expressions
Définitions locales

Nommer localement

let *nom* = $expr_1$ in $expr_2$

```
#let x = 2*5+1;;
   val x : int = 11

#let x = 2 in x*x;;
   - : int = 4

 #x;;
   - : int = 11
```

Les expressions
Définitions locales

Nommer localement

let *nom* = $expr_1$ in $expr_2$

```
#let a = 1 and b = 2 in 2*a+b;;
   - : int = 4

#let a = 1 in let b = 2*a in b+a;;
   - : int = 3

#let a = 1 and b = 2*a in b+2+a;;
   Unbound value
```

Les types de base
int et float

int : + - * / mod

float : +. -. *. /. ** sqrt cos sin tan asin atan log exp

```
#3.45;;
   - : float = 3.45

#3.45E10;;
   - : float = 34500000000.

#3.45e10;;
   -: float = 34500000000.

#3.45 E 10;;
This expression is not a function, it cannot be applied.
```

Les types de base
int et float

Conversion int/float

```
#int_of_float 1.;;
-  : int = 1
```

```
#int_of_float 1.1;;
- : int = 1
```

```
#float_of_int 76;;
- : float = 76.0
```

Typage fort des opérateurs

```
# 2.4 +. 1.1;;
   - : float = 3.5
# 2.4 + 1.1;;
This expression has type float but is here used with type int
```

Les types de base
bool

bool : true, false, & (&&), or (||), not

Construction d'expressions booléennes:

<, <=, >=, =, <>, if..then...else

#1<2 & 3<9;; - : bool = true	#1<=0 & (1/0)>1;; - : bool = false
0<1 || (1/0)>1;; - : bool = true	# 0<1 & (1/0)>1;; Exception: Division_by_zero.

Les types de base
bool

If then else

Syntaxe

```
if  expression booléenne  then
        expr1
else
        expr2
```

Typable si : $expr_1$ et $expr_2$ de même type T

Type : T

Une seule des expressions $expr_1$ et $expr_2$ est évaluée.

Les types de base
bool

If then else (Exemples)

```
# if 1<2 then 0 else 1;;
   - : int = 0
```

Les types de base
bool

If then else (Exemples)

```
# if 1<2 then 0 else 1;;
  - : int = 0
```

```
# if 1<2 then 0 else 1/0;;
  - : int = 0
```

Les types de base
bool

If then else (Exemples)

```
# if 1<2 then 0 else 1;;
  - : int = 0
```

```
# if 1<2 then 0 else 1/0;;
  - : int = 0
```

```
# if 2<1 then 0 else 1/0;;
  Exception: Division_by_zero.
```

Les types de base
bool

If then else (Exemples)

```
# if 1<2 then 0 else 1;;
   - : int = 0
```

```
# if 1<2 then 0 else 1/0;;
   - : int = 0
```

```
# if 2<1 then 0 else 1/0;;
   Exception: Division_by_zero.
```

```
# if 1<2 then "arbre" else "oiseau";;
   - : string = "arbre”
```

Les types de base
bool

If then else (Exemples)

```
# if 1<2 then 0 else 1;;
   - : int = 0
```

```
# if 1<2 then 0 else 1/0;;
   - : int = 0
```

```
# if 2<1 then 0 else 1/0;;
   Exception: Division_by_zero.
```

```
# if 1<2 then "arbre" else "oiseau";;
   - : string = "arbre"
```

```
# if 1<2 then 0 else "arbre";;
This expression has type string but is here used with type int
```

Les types de base
string

Tout texte entre “ “

Opérateur de concaténation : ^

```
#"ceci est une"^" chaine de caracteres";;
- : string = "ceci est une chaine de caracteres"

#"ceci est une"  ^  "chaine de caracteres";;
- : string = "ceci est unechaine de caracteres"
```

Les types de base
char

Entre ' '

#'a';;
- : char = 'a'

#"a";;
- : string = "a"

Les types de base
char

Entre ' '

#'a';;
- : char = 'a'

#"a";;
- : string = "a"

Conversion en ASCII : int_of_char

Les types de base
char

Entre ' '

#'a';;
- : char = 'a'

#"a";;
- : string = "a"

Conversion en ASCII : int_of_char

Réciproque: char_of_int

Les types de base
char

Entre ' '

```
#'a';;
   - : char = 'a'
```

```
#"a";;
   - : string = "a"
```

Conversion en ASCII : int_of_char

Réciproque: char_of_int

```
#int_of_char 'a';;
   - : int = 97
```

```
#char_of_int 97;;
   - : char = 'a'
```

Les types
Produit cartésien

```
# 1,2;;
- : int * int = (1, 2)
```

Les types
Produit cartésien

```
# 1,2;;
-: int * int = (1, 2)
```

```
# 1,2,3;;
- : int * int * int = (1, 2, 3)
```

Les types
Produit cartésien

```
# 1,2;;
- : int * int = (1, 2)
```

```
# 1,2,3;;
- : int * int * int = (1, 2, 3)
```

```
# 1, (2,3);;
- : int * (int * int) = (1, (2, 3))
```

Les types
Produit cartésien

```
# 1,2;;
- : int * int = (1, 2)
```

```
# 1,2,3;;
- : int * int * int = (1, 2, 3)
```

```
# 1, (2,3);;
- : int * (int * int) = (1, (2, 3))
```

```
# ("coucou" , 3.1, ('A',2));;
-   : string * float * (char * int) = ("coucou", 3.1, ('A', 2))
```

Les types
Produit cartésien

```
# 1,2;;
- : int * int = (1, 2)
```

```
# 1,2,3;;
- : int * int * int = (1, 2, 3)
```

```
# 1, (2,3);;
- : int * (int * int) = (1, (2, 3))
```

```
# ("coucou" , 3.1, ('A',2));;
- : string * float * (char * int) = ("coucou", 3.1, ('A', 2))
```

```
# "coucou" , 3.1, ('A',2);;
-   : string * float * (char * int) = ("coucou", 3.1, ('A', 2))
```

Les types
Produit cartésien

```
# 1,2;;
- : int * int = (1, 2)
```

```
# 1,2,3;;
- : int * int * int = (1, 2, 3)
```

```
# 1, (2,3);;
- : int * (int * int) = (1, (2, 3))
```

```
# ("coucou" , 3.1, ('A',2));;
- : string * float * (char * int) = ("coucou", 3.1, ('A', 2))
```

```
# "coucou" , 3.1, ('A',2);;
- : string * float * (char * int) = ("coucou", 3.1, ('A', 2))
```

```
# true & false, 3+3;;
- : bool * int = (false, 6)
```

Les types
Produit cartésien

Les projections

```
# fst;;
   - : 'a * 'b -> 'a = <fun>
```

(polymorphisme)

```
# snd;;
   - : 'a * 'b -> 'b = <fun>

# let x = (1,"coucou") and y= ("hello",2.1) in (snd x, fst y);;
   - : string * string = ("coucou", "hello")
```

Expressions Fonctionnelles
Application à un argument

```
# let square = fun x-> x*x;;          # let square x = x*x;;
val square : int -> int = <fun>
```

Opérateur d'application : Implicite (juxtaposition)

```
# square 4;;                          # (fun x -> x*x)4;;
                - : int = 16
```

Opérateur de plus forte priorité.

```
# square 3+1;;                        # square (3+1);;
- : int = 10                         - : int = 16
# fun x -> x*x 4;;
```

This expression is not a function, it cannot be applied

Expressions Fonctionnelles
Ordre supérieur

Currification

Une fonction Caml a un UNIQUE argument

```
# let m = fun (x,y) -> x*y;;
   val m : int * int -> int = <fun>
```

```
# let m (x,y) = x*y;;
```

Un seul argument qui est un couple (x,y)

```
# m (2,3);;
   - : int = 6
# m 2,3;;
   This expression has type int but is here used with type int * int
```

Expressions Fonctionnelles
Ordre supérieur

Une fonction Caml a un UNIQUE argument

```
# let m = fun (x,y) -> x*y;;
    val m : int * int -> int = <fun>
```

Expressions Fonctionnelles
Ordre supérieur

Une fonction Caml a un UNIQUE argument

```
# let m = fun (x,y) -> x*y;;
    val m : int * int -> int = <fun>
```

Currification

Expressions Fonctionnelles
Ordre supérieur

Une fonction Caml a un UNIQUE argument

```
# let m = fun (x,y) -> x*y;;
    val m : int * int -> int = <fun>
```

Currification

```
# let mult = fun x -> (fun y -> x*y);;
```

Expressions Fonctionnelles
Ordre supérieur

Une fonction Caml a un UNIQUE argument

```
# let m = fun (x,y) -> x*y;;
    val m : int * int -> int = <fun>
```

Currification

```
# let mult = fun x -> (fun y -> x*y);;      #let mult x = fun y -> x*y;;
```

Expressions Fonctionnelles
Ordre supérieur

Une fonction Caml a un UNIQUE argument

```
# let m = fun (x,y) -> x*y;;
    val m : int * int -> int = <fun>
```

Currification

```
# let mult = fun x -> (fun y -> x*y);;      #let mult x = fun y -> x*y;;
    val mult : int -> int -> int = <fun>
```

Expressions Fonctionnelles
Ordre supérieur

Une fonction Caml a un UNIQUE argument

```
# let m = fun (x,y) -> x*y;;
    val m : int * int -> int = <fun>
```

Currification

```
# let mult = fun x -> (fun y -> x*y);;      #let mult x = fun y -> x*y;;
    val mult : int -> int -> int = <fun>
```

int -> int -> int abréviation pour int -> (int -> int)

Expressions Fonctionnelles
Ordre supérieur

Une fonction Caml a un UNIQUE argument

```
# let m = fun (x,y) -> x*y;;
    val m : int * int -> int = <fun>
```

Currification

```
# let mult = fun x -> (fun y -> x*y);;      #let mult x = fun y -> x*y;;
```

int -> int -> int abréviation pour int -> (int -> int)

Argument: de type **int** Résultat : de type **int -> int**
(ordre supérieur)

Expressions Fonctionnelles
Ordre supérieur

Une fonction Caml a un UNIQUE argument

```
# let m = fun (x,y) -> x*y;;
    val m : int * int -> int = <fun>
```

Currification

```
# let mult = fun x -> (fun y -> m(x,y));; #let mult x = fun y -> m(x,y);;
```

int -> int -> int abréviation pour int -> (int -> int)

Argument: de type **int** Résultat : de type **int -> int**
(ordre supérieur)

mult est la currifiée de m
Que vaut (mult 2) ?

Expressions Fonctionnelles
Ordre supérieur

Une fonction Caml a un UNIQUE argument

```
# let m = fun (x,y) -> x*y;;
    val m : int * int -> int = <fun>
```

Currification

```
# let mult = fun x -> (fun y -> x*y);;      #let mult x = fun y -> x*y;;
```

int -> int -> int abréviation pour int -> (int -> int)

Argument: de type **int** Résultat : de type **int -> int**
(ordre supérieur)

```
# mult 2;;
- : int -> int = <fun>
```

Expressions Fonctionnelles
Ordre supérieur

Currification (suite)

```
# let double = mult 2;;
   val double : int -> int = <fun>

# double 3;;
   - : int = 6

 mult 2 3;;
   - : int = 6
```

Avantage de mult / m?

Expressions Fonctionnelles
Ordre supérieur

Currification (écritures équivalentes) de m

```
# let m = fun (x,y) -> x*y;;
val m : int * int -> int = <fun>
```

```
# let mult = fun x -> (fun y -> m(x,y));;
# let mult = fun x -> fun y -> m(x,y);;
# let mult = fun x  y -> m(x,y);;
# let mult  x = fun y -> m(x,y);;
# let mult x  y = m(x,y);;
```

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

Expressions Fonctionnelles
Ordre supérieur

Currification - Cas général

Soit f : 'a * 'b -> 'c f possède 2 currifiées

```
# let f1 = fun x -> fun y -> f(x, y);;
# let f2 = fun y -> fun x -> f(x, y);;
```

Si f(x, y) = f(y, x) alors $f_1 = f_2$
Faux dans le cas contraire.

f(x, y) = 2*x + y

(f_1 x) y = 2*x + y

(f_2 x) y = 2*y + x

Expressions Fonctionnelles
Ordre supérieur

Currification - Cas général

Soit f : 'a * 'b -> 'c f possède 2 currifiées

```
# let f1 = fun x -> fun y -> f(x, y);;
# let f2 = fun y -> fun x -> f(x, y);;
```

Si f(x, y) = f(y, x) alors $f_1 = f_2$
Faux dans le cas contraire.

f(x, y) = 2*x + y
(f_1 x) y = 2*x + y
(f_2 x) y = 2*y + x

Dissymétrie dans les arguments

Expressions Fonctionnelles
Ordre supérieur

Opérateurs de currification

```
# let c1 f =  fun x -> fun y -> f(x, y);;
   val c1 : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

Argument et résultat sont d'ordre supérieur

```
# let c2 f = fun y -> fun x -> f(x, y);;
   val c2 : ('a * 'b -> 'c) -> 'b -> 'a -> 'c = <fun>
```

Expressions Fonctionnelles
Ordre supérieur

Autre exemple de fonction d'ordre supérieur
La composition de 2 fonctions

```
#let comp = fun f -> fun g -> fun x -> f(g x);;
#let comp = fun f  g -> (fun x -> f(g x));;
#let comp = fun f g x -> f(g x);;
#let comp f g = fun x -> f(g x);;
#let comp f g x = f (g x);;
```

Expressions Fonctionnelles
Ordre supérieur

Autre exemple de fonction d'ordre supérieur
La composition de 2 fonctions

```
#let comp = fun f -> fun g -> fun x -> f(g x);;
#let comp = fun f  g -> (fun x -> f(g x));;
#let comp = fun f g x -> f(g x);;
#let comp f g = fun x -> f(g x);;
#let comp f g x = f (g x);;

val comp : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Expressions Fonctionnelles
Ordre supérieur

Composition de 2 fonctions

```
#let comp f g x = f (g x);;

# comp square double 3;;
   - : int = 36

# comp square;;
   - : ('_a -> int) -> '_a -> int = <fun>

# let f = comp square int_of_float;;
   val f : float -> int = <fun>

# f 3.4;;
   - : int = 9
```

Digression
Déclaration d'un opérateur infixe

(op) *(symboles dans { +, /, *, $, %, etc})*

```
#let (%)  = comp;;
#let (%)  f g  = fun x -> f(g x);;

# let f = square % int_of_float;;
   val f : float -> int = <fun>

# f 3.4;;
   - : int = 9

# + ;;
   Syntax error
```

```
# (+) ;;
- : int -> int -> int = <fun>
```

Fonctions récursives

let *x* = *expr*

Si *x* apparaît dans *expr*, il doit avoir été défini au préalable

```
# let x =1;;
   val x : int = 1

# let x = x+2;;
   val x : int = 3

# let x = let x=1 in x+2;;
   val x : int = 3

# let fact n  = if n = 0 then 1 else n * fact(n-1);;
   Unbound value fact
```

Fonctions récursives

let rec *nom* = *expr*

```
# let rec fact n = if n=0 then 1 else n * fact(n-1);;
   val fact : int -> int = <fun>

# fact 4;;
   - : int = 24
```

Attention!

```
#let rec fact n = if n=0 then 1 else n * (fact n-1);;
   val fact : int -> int = <fun>

# fact 4;;;
   Stack overflow during evaluation (looping recursion?).
```

Fonctions récursives
Exemple

Puissance $n^{ième}$ (calcul classique)

```
# let rec puiss n x= if n=0 then 1. else x *. puiss (n-1) x;;
   val puiss : int -> float -> float = <fun>

# puiss 2 3.;;
   - : float = 9.0

# puiss 200000 1.;;
   Stack overflow during evaluation (looping recursion?).
```

Fonctions récursives
Exemple

Puissance égyptienne

```
#let rec puiss n x =      if n = 0 then 1.
                          else
                                let v = puiss (n/2) x  in
                                  if  n mod 2 =0 then
                                      v*.v
                                  else
                                      v*.v*.x;;

val puiss : int -> float -> float = <fun>
```

Fonctions récursives
Exemple

Puissance égyptienne

```
# puiss 1000000000 1.;;     (1 milliard) résultat immédiat
   - : float = 1.

# puiss 2000000000 1.;;
   Integer literal exceeds the range of representable
   integers of type int
```

Fonctions récursives
Exemple

Puissance égyptienne (mauvaise version)

```
#let rec puiss_bad n x =
  if n=0 then 1.
  else
    if n mod 2 = 0 then
        (puiss_bad (n/2) x)*.(puiss_bad (n/2) x)
    else
        x*.(puiss_bad ((n-1)/2) x)*.(puiss_bad ((n-1)/2) x);;

# puiss_bad 1000000000 1.;;  (1 milliard, 10 minutes)
```

Fonctions mutuellement récursives

Exemple 1: les fonctions s'appellent mutuellement de façon globale

```
#let rec   even n =        if n=0 then true else odd(n-1)
  and
           odd n   =       if n=0 then false else even(n-1);;

  val even : int -> bool = <fun>
  val odd : int -> bool = <fun>
```

Les deux fonctions sont définies globalement

Fonctions mutuellement récursives

Exemple 2: Une des 2 fonctions est locale

```
# let rec even n =  if n=0 then true
                    else
                         let  rec odd n = if n=0 then false
                                          else   even(n-1)
                         in odd(n-1)
   val even : int -> bool = <fun>
```

even est connue globalement

odd est locale à la définition de *even*

Fonctions mutuellement récursives

Exemple 2: Une des 2 fonctions est locale

```
# let rec even n =  if n=0 then true
                    else
                         let  rec odd n = if n=0 then false
                                          else   even(n-1)
                         in odd(n-1)
   val even : int -> bool = <fun>

#even 1000000000;;  (1 milliard, 2 minutes)
```

Filtrage (pattern matching)
Introduction

```
# let neg b = match b with
     true -> false
   | false -> true;;
   val neg : bool -> bool = <fun>

# let xor b1 b2 = match (b1,b2) with
     (true, true) -> false
    |(false, true) -> true
    |(true, false) -> true
    |(false, false) ->false;;
val xor : bool -> bool -> bool = <fun>
```

Expression

↓

Filtrage

↓

Motif (pattern)

Ici, exclusif

et

exhaustif

Filtrage (pattern matching)
Introduction

Le motif peut contenir des variables

```
# let xor b1 b2 = match (b1,b2) with
    (false, b)  -> b
  | (true,  b) -> not b ;;

# let xor b1 b2 = match (b1,b2) with
    (false, false)  -> false
  | (true,  true) -> false
  |     _         -> true;;
```

“_” Filtre tout

Filtrage (pattern matching)
Introduction

Syntaxe

match *expr* with
| p_1 -> $expr_1$
| p_2 -> $expr_2$
...
| p_n -> $expr_n$

expr est filtrée séquentiellement sur les motifs p_1, ..., p_n

Motif: fait de variables, constantes des types de base, constructeurs ,
_ *(précisions par la suite)*

Filtrage (pattern matching)
Introduction

Exemple de filtrage non exclusif

```
# let rec fact n = match n with
        0 -> 1|
        x -> fact(x-1)*x;;
```

le choix se fait séquentiellement, de haut en bas

Filtrage (pattern matching)
Introduction

Exemple de filtrage non exhaustif

```
# let f n = match n with 0 -> 0
                       | 1 -> 2;;

Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched: 2
val f : int -> int = <fun>

# f 3;;
Exception: Match_failure ("", 74, -48)
```

Filtrage (pattern matching)
Introduction

Exemple de certains cas jamais atteints

```
# let f n = match n with  p -> 2*p
                        | 0 -> 1;;
   Warning U: this match case is unused.
   val f : int -> int = <fun>
```

Filtrage (pattern matching)
Introduction

Linéarité du motif

Pour une variable donnée, 1 seule occurrence par motif

```
# let div x y = match (x,y) with    ( _, 0 ) -> 1
                                    |( x, y )  -> x/y;;

   val div : int -> int -> int = <fun>

# let xor  b1 b2 = match (b1,b2) with       (b, b) -> false
                                          |    _    -> true ;;

   This variable is bound several times in this matching
```

Filtrage (pattern matching)
Introduction

Filtrage avec garde

Ne se fait que si la *condition de garde* est satisfaite

```
# let xor  b1 b2 = match (b1,b2) with
       (b1, b2) when b1 = b2 -> false
      |   _                  -> true ;;

  val xor : 'a -> 'a -> bool = <fun>
```

Remarquer le type de cette fonction

Expressions Fonctionnelles
Ordre supérieur

η – réduction

```
# let mult = fun x y -> x*y;;
# let double  = fun x -> mult 2 x;;
# let double = mult 2;;
```

fun x -> f x désigne la même fonction que f

cf. en math x-> sin(x) et sin
où f une expression (c'est mult 2 ci- dessus)

f est obtenue à partir de fun x -> f x par η-réduction
L'expression f est la forme η-réduite de fun x -> f x