

# Chapitre 1 - Concepts de base d'OCaml

INF6120: Programmation fonctionnelle et logique

Quentin Stiévenart

Université du Québec à Montréal

v251



# Historique d'OCaml

- ML (*Meta Language*) créé en 1973 par Robert Milner (Édimbourg)
- Caml (*Categorical Abstract Machine Language*) créé en 1985 (INRIA/ENS)
- OCaml (*Objective Caml*) créé en 1996 (INRIA)
  - Le langage évolue constamment, de façon rétro-compatible

# Caractéristiques

*An industrial-strength functional programming language with an emphasis on expressiveness and safety* ocaml.org

*OCaml (/o kæmə/ oh-KAM-əl, formerly Objective Caml) is a general-purpose, high-level, multi-paradigm programming language* Wikipedia

- **Fondation:** lambda calcul
- **Modèle d'exécution:** interprété et compilé
- **Paradigme:** fonctionnel / multi-paradigme
- **Typage:** statique fort avec inférence
- **Modèle d'évaluation:** strict
  - Comme la majorité des langages fonctionnels (mais pas Haskell)

# Outils de base

- **Implémentation:** [ocaml.org](https://ocaml.org)
  - le langage est défini par son implémentation
  - on utilisera la dernière version LTS (4.14.2)
- **Documentation:**
  - [Bibliothèque standard](#)
- **Gestionnaire de paquets/dépendances:**
  - [opam](#)
- **Outil de compilation**
  - [dune](#)

# Utilisateurs commerciaux de OCaml

Quelques utilisateurs listés sur [ocaml.org](https://ocaml.org), notamment :

- Docker: utilise OCaml (MirageOS) pour la portabilité sur Mac/Windows
- Facebook: utilisé dans Hack (compilateur PHP), Flow (vérification de types pour JavaScript)
- JaneStreet: gros contributeur au langage, service financiers
- ...

# Installer OCaml

Voir labo

# Interaction avec l'interpréteur

```
$ utop  
...  
Type #utop_help for help about using utop.  
  
#
```

# UTop : développement interactif

```
# 3 + 39 ;;  
- : int = 42
```

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

```
# #use "foo.ml";;  
val x : int = 2
```

**Attention** : utop n'exécute rien tant qu'on ne termine pas par un ;;

# Compiler

```
hello.ml
```

```
let _ = print_endline "Hello, world!"
```

Interprétation d'un fichier :

```
$ ocaml hello.ml
```

Compilation d'un fichier unique :

```
$ ocamlc hello.ml -o hello
```

```
$ ./hello
```

# Compiler avec dune

```
$ dune init proj mon-projet  
$ cd projet/  
$ dune build  
$ dune exec projet  
Hello, World!
```

# Convention

Dans les diapos, on utilise cette notation pour des interactions avec UTop :

```
# 1 + 2;;
```

```
- : int 3
```

Hors de UTop (dans un fichier .ml), on évitera les ; ;

# Identifiants du langage

Un identifiant a pour première lettre:

- Minuscule : c'est une liaison (*binding*) ou un type : `pi`, `x`, `string`, `list`
- Apostrophe : c'est une variable de type *polymorphe* : `'a`
- Majuscule: c'est un constructeur de type ou un module : `Stdlib`, `Some`

**Note:** possibilité d'utiliser d'autres caractères comme première lettre pour définir des opérateurs

# Littéraux

```
# 3;;  
- : int = 3  
# 3.1415;;  
- : float = 3.1415  
# 1e10;;  
- : float = 10000000000.  
# 0xdeadbeef;;  
- : int = 3735928559  
# 'a';;  
- : char = 'a'  
# "hello";;  
- : string = "hello"  
# true;;  
- : bool = true
```

# Expressions

```
# 5 * (-3);;  
- : int = - 15  
# abs 42;;  
- : int = 42  
# abs (-3);;  
- : int = 3  
# true || false;;  
- : bool = true
```

# Types de données de base

## Types de données de base

- booléens : `bool`
- nombres entiers : `int`
- nombres flottants : `float`
- caractères : `char`
- chaînes de caractère : `string`
- type unité : `unit`
- uplets : `'a * 'b`
- listes : `'a list`

## Booléens : bool

```
# true;;  
- : bool = true  
# false;;  
- : bool = false  
# true && false;;  
- : bool = false  
# true || false;;  
- : bool = true  
# not true;;  
- : bool = false
```

## Nombres : int et float

```
# 3;;  
- : int = 3  
  
# 0x1a;;  
- : int = 26  
  
# 0b1100;;  
- : int = 12  
  
# 3.14;;  
- : float = 3.14
```

Représentation interne : int est représenté dans un mot mémoire avec un bit utilisé en interne, on a donc des entiers de 63 bits.

## Nombres : pas de conversion implicite

OCaml ne permet pas les conversions implicites des nombres :

```
# 3 + 3.14;;  
      ^^^^
```

Error: This expression has **type float** but  
an expression was expected **of type int**

## Nombres : pas de conversion implicite

OCaml ne permet pas les conversions implicites des nombres :

```
# 3 + 3.14;;  
      ^^^^
```

Error: This expression has **type float** but  
an expression was expected **of type int**

```
# 3.0 + 3.14;;  
      ^^^
```

Error: This expression has **type float** but  
an expression was expected **of type int**

## Nombres : pas de conversion implicite

OCaml ne permet pas les conversions implicites des nombres :

```
# 3 + 3.14;;  
      ^^^^
```

Error: This expression has **type float** but  
an expression was expected **of type int**

```
# 3.0 + 3.14;;  
      ^^^
```

Error: This expression has **type float** but  
an expression was expected **of type int**

```
# 3.0 +. 3.14;;  
- : float = 6.14000000000000057
```

# Nombres : conversions explicites

```
# int_of_float 3.14;;  
- : int = 3
```

```
# float_of_int 3;;  
- : float = 3.
```

# Nombres : fonctions sur les entiers

```
# 1 + (2 * (3 - 4));  
- : int = -1  
# 5 / 2;;  
- : int = 2  
# 5 mod 2;;  
- : int = 1  
# succ 1;;  
- : int = 2  
# pred 1;;  
- : int = 0
```

Voir [Stdlib: Integer arithmetic](#)

## Nombres : fonctions sur les flottants

```
# 1. +. (2. *. (3. -. 4.));;  
- : float = -1.  
# 5. /. 2.;;  
- : float = 2.5  
# 5. ** 2.;;  
- : float = 25.  
# sqrt 25.;;  
- : float = 5.  
# sin 3.14;;  
- : float = 0.00159265291648682823
```

Voir [Stdlib: Floating-point arithmetic](#)

## Caractères : char

```
# 'a';;  
- : char = 'a'  
  
# '\n';;  
- : char = '\n'  
  
# int_of_char '\n';;  
- : int = 10  
  
# char_of_int 65;;  
- : char = 'A'
```

## Chaînes de caractères : string

```
# "hello";;  
- : string = "hello"  
  
# string_of_int 37;;  
- : string = "37"  
  
# "hello" ^ "world";;  
- : string = "helloworld"
```

## Type unité : `unit`

```
# ();;  
- : unit = ()
```

Utilisé notamment pour les fonctions à *effet de bord* (*side effect*) :

```
# print_endline "Hello!";;  
Hello!  
- : unit = ()
```

Souvent appelé `void` dans d'autres langages

## Uplets : 'a \* 'b, etc.

Un uplet (*tuple*) est un type avec :

- une **taille fixe** : `int * int` contient *exactement* deux éléments
- un **contenu hétérogène** : `int * bool` contient des types différents

Le type s'écrit avec une étoile, la valeur s'écrit avec une virgule

```
# (3, "foo");;  
- : int * string = (3, "foo")
```

```
# (1, 2, 3, 4);;  
- : int * int * int * int = (1, 2, 3, 4)
```

Paires : 'a \* 'b

```
# fst ("foo", 42);;  
- : string = "foo"
```

```
# snd ("foo", 42);;  
- : int = 42
```

'a \* 'b en Java serait Pair<A, B>

## Listes : 'a list

'a list est une liste contenant des valeurs de type 'a.

Une liste a :

- une **taille variable** : int list contient 0, 1, ou plusieurs entiers
- un **contenu homogène** : int list ne contient *que* des entiers

**Attention** : les éléments d'une liste sont séparé par ;

## Listes : 'a list

```
# [1; 2];;  
- : int list = [1; 2]
```

```
# [1; 2; 3; 4];;  
- : int list = [1; 2; 3; 4]
```

```
# [1; "hello"];;
```

Error: This expression has **type string** but  
an expression was expected **of type int**

## Listes : 'a list

Question :

- quel est le type de [1; 2] ?

## Listes : 'a list

Question :

- quel est le type de `[1; 2]` ?
- quel est le type de `[1, 2]` ?

## Listes : 'a list

Question :

- quel est le type de `[1; 2]` ?
- quel est le type de `[1, 2]` ?
- quel est le type de `[1, 2, 3]` ?

## Listes : 'a list

Question :

- quel est le type de `[1; 2]` ?
- quel est le type de `[1, 2]` ?
- quel est le type de `[1, 2, 3]` ?
- quel est le type de `[1; 2, 3]` ?

## Fonctions : 'a -> 'b

Les fonctions forment également un type de base.

On peut définir des fonctions anonymes :

```
# fun x -> x * x;;
```

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

```
# fun x y z -> (x + y) * z;;
```

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

# Appels de fonctions

Une fonction s'appelle en donnant les arguments **séparés par des espaces**

```
# (fun x -> x * x) 1;;  
- : int = 1
```

```
# (fun x y z -> (x + y) * z) 1 2 3;;  
- : int = 9
```

```
# (fun x y z -> (x + y) * z)(1, 2, 3);;
```

Error: This expression has **type** 'a \* 'b \* 'c  
but an expression was expected **of type** int

# Combinaisons de types

`'a list list`

- se lit : `('a list) list`
- c'est une liste contenant des `'a list`

En Java, cela serait un `List<List<A>>`

# Combinaisons de types

Questions :

- que représente le type `int list` ?

# Combinaisons de types

Questions :

- que représente le type `int list` ?
- que représente le type `int list -> int` ?

# Combinaisons de types

Questions :

- que représente le type `int list` ?
- que représente le type `int list -> int` ?
- que représente le type `(int -> int) list` ?

# Combinaisons de types

Questions :

- que représente le type `int list` ?
- que représente le type `int list -> int` ?
- que représente le type `(int -> int) list` ?
- que représente le type `int -> int list` ?

# Constructions du langage

## Afficher des valeurs

```
# print_int 1;;  
1- : unit = ()  
# print_string "foo";;  
foo- : unit = ()  
# print_endline "foo";;  
foo  
# print_char 'a';;  
a- : unit = ()  
# print_newline ();;  
  
- : unit = ()
```

# Afficher des valeurs

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

Ces fonctions n'ont “pas” de valeur de retour, mais elles effectuent des **effets de bord**

Un *effet de bord* est une action qui est *observable* hors de l'environnement local.

C'est un élément **non fonctionnellement pur** du langage.

→ Pour les TPs, les éléments non fonctionnellement purs seront proscrits sauf mention contraire.

## Afficher des valeurs

```
# Printf.printf "%s %d %x\n" "hello" 42 42;;  
hello 42 2a
```

# Séquençage

Si on souhaite séquencer plusieurs opérations ayant `unit` comme type de retour, on peut utiliser ;

```
# print_int 1; print_newline (); print_string "hello";;  
1  
hello- : unit = ()
```

## Liaisons : let

Par abus de langage, on parle de *variables*, mais on ne va manipuler que des **constantes** (immuables), associées via des **liaisons**

Une liaison est introduite avec le mot clé `let` :

```
# let x = 5;;  
val x : int = 5  
# let y = 5 + x;;  
val y : int = 10  
# let z = y + 2;;  
val z : int = 12
```

## Liaison locale : `let ... in ...`

On peut définir une liaison qui s'applique uniquement dans une sous-expression :

`let <var> = <expr1> in <expr2>`

- évalue `<expr1>` a une valeur `v`
- crée la liaison `<var> = v` *uniquement* dans `<expr2>`

## Liaison locale : let ... in ...

```
# let x = 5 in
```

```
  x;;
```

```
- : int = 5
```

```
# x;;
```

Error: Unbound value x

```
# let x = 5 in
```

```
  let y = 2 in
```

```
    x + y;;
```

```
- : int = 7
```

## Redéfinition de liaison (*shadowing*)

Question : quel est le résultat de l'évaluation suivante ?

```
# let x = 42 in  
  let x = 5 in  
    x;;
```

## Redéfinition de liaison (*shadowing*)

Question : quel est le résultat de l'évaluation suivante ?

```
# let x = 42 in  
  let x = 5 in  
    x;;  
  
- : int = 5
```

## Redéfinition de liaison (*shadowing*)

C'est équivalent à :

```
let x = 42 in  
(let x = 5 in x)
```

On peut remplacer la parenthèse par :

```
let x = 42 in  
5
```

# Redéfinition de liaison

Question : quel est le résultat de l'évaluation suivante ?

```
# let x = 42 in  
  let x = x+1 in  
  x;;
```

# Redéfinition de liaison

Question : quel est le résultat de l'évaluation suivante ?

```
# let x = 42 in  
  let x = x+1 in  
    x;;  
  
- : int = 43
```

Rappel :

let <var> = <expr1> in <expr2>

- évalue <expr1> a une valeur  $v$
- crée la liaison <var> =  $v$  *uniquement* dans <expr2>

# Redéfinition de liaison

Question : quel est le résultat de l'évaluation suivante ?

```
# let x = 42 in  
  print_int x;  
  let x = 5 in  
    print_int x;;
```

# Redéfinition de liaison

Question : quel est le résultat de l'évaluation suivante ?

```
# let x = 42 in  
  print_int x;  
  let x = 5 in  
    print_int x;;  
425- : unit = ()
```

# Redéfinition de liaison

Question : quel est le résultat de l'évaluation suivante ?

```
# let x = 42 in
  begin
    print_int x;
    let x = 5 in
      ()
    end;
  print_int x
;;
```

## Redéfinition de liaison

Question : quel est le résultat de l'évaluation suivante ?

```
# let x = 42 in
  begin
    print_int x;
    let x = 5 in
      ()
    end;
  print_int x
;;

4242 : unit = ()
```

# Redéfinition de liaison

L'introduction d'une nouvelle variable “cache” les définitions précédentes, uniquement dans le *champ* (*scope*) de définition de la nouvelle variable

Lorsqu'on évalue une variable, on prend la définition englobante la plus proche

# Conditions

```
# let x = 5 ;;  
val x : int = 5  
# if x < 0 then (- x) else x;;  
- : int = 5
```

# Groupeage d'expressions séquencées

Utilisation de `begin ... end` :

```
# if x < 0 then begin
    print_endline "négatif";
    (- x)
end else begin
    print_endline "positif";
    x
end;;
positif
- : int = 5
```

# Définir une fonction

```
# let f = fun x y -> x + y;;  
val f : int -> int -> int = <fun>
```

```
# f 2 3;;  
- : int = 5
```

## Définir une fonction : version alternative

```
# let f x y = x + y;;  
val f : int -> int -> int = <fun>
```

```
# f 2 3;;  
- : int = 5
```

## Définir une fonction : annotation de types

On peut ajouter les types des arguments et des valeurs de retour :

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

```
# f 2 3;;  
- : int = 5
```

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

```
# f 2 3;;  
- : int = 5
```

→ Pour les TP, il sera demandé d'annoter les types des fonctions.

# Nombre d'arguments

Type d'une fonction a un argument qui renvoie un entier :

```
int -> int
```

Type d'une fonction à trois arguments qui renvoie un entier :

```
int -> int -> int -> int
```

# Exercice

Donner une valeur pour les types suivants :

- `int list`

# Exercice

Donner une valeur pour les types suivants :

- `int list : [1; 2]`
- `(int * char) list`

# Exercice

Donner une valeur pour les types suivants :

- `int list : [1; 2]`
- `(int * char) list : [(1, 'a'); (2, 'b')]`
- `(int list * char list)`

# Exercice

Donner une valeur pour les types suivants :

- `int list : [1; 2]`
- `(int * char) list : [(1, 'a'); (2, 'b')]`
- `(int list * char list) : ([1; 2], ['a'; 'b'])`
- `(unit -> int) list`

## Exercice

Donner une valeur pour les types suivants :

- `int list : [1; 2]`
- `(int * char) list : [(1, 'a'); (2, 'b')]`
- `(int list * char list) : ([1; 2], ['a'; 'b'])`
- `(unit -> int) list : [(fun (x : unit) -> 1); (fun (y: unit) -> 2)]`

## Exemple : abs

```
let abs (x : int) : int =  
  if x > 0 then  
    x  
  else  
    -x
```

## Exercice : rms

Définir une fonction `rms` qui calcule la *moyenne quadratique* de deux nombres `x` et `y`, c'est-à-dire  $\sqrt{(x^2 + y^2)/2}$

- Étape 1 : quel est le type de la fonction ?

## Exercice : rms

Définir une fonction `rms` qui calcule la *moyenne quadratique* de deux nombres `x` et `y`, c'est-à-dire  $\sqrt{(x^2 + y^2)/2}$

- Étape 1 : quel est le type de la fonction ?

```
val rms : float -> float -> float
```

- Étape 2 : définir la fonction

## Exercice : rms

Définir une fonction `rms` qui calcule la *moyenne quadratique* de deux nombres `x` et `y`, c'est-à-dire  $\sqrt{(x^2 + y^2)/2}$

- Étape 1 : quel est le type de la fonction ?

```
val rms : float -> float -> float
```

- Étape 2 : définir la fonction

```
let square (x : float) : float = x *. x
```

```
let rms (x : float) (y : float) =  
  ((square x) +. (square y)) /. 2
```

## Exercice

Définir une fonction de type `'a -> 'b -> 'a`

## Exercice

Définir une fonction de type `'a -> 'b -> 'a`

```
let f (x : 'a) (y : 'b) : 'a = x
```

(C'est la seule solution possible !)

Équivalent en Java :

```
B f<A, B>(A x, B y) {  
    return x;  
}
```

# Égalité : = et ==

Deux égalités présentes :

- `(=)` : `'a -> 'a -> bool` : égalité structurelle
  - le `.equals` de Java
  - `x <> y` est not `(x = y)`
- `(==)` : `'a -> 'a -> bool` : égalité physique
  - le `==` de Java
  - `x != y` est not `(x == y)`
  - on n'utilisera pas ceci

## Égalité structurelle : =

```
# 1 = 1;;  
- : bool = true  
# [1; 2; 3] = [1; 2; 3];;  
- : bool = true  
# "foo" = "foo";;  
- : bool = true
```

L'égalité structurelle sera automatiquement définie pour tout nouveau type introduit.

## Égalité structurelle : =

Les seules valeurs qu'on ne peut pas comparer sont les fonctions

```
# (fun x -> 1) = (fun x -> 2);;
```

```
Exception: Invalid_argument "compare: functional value".
```

## Égalité physique : ==

```
# 1 == 1;;  
- : bool = true  
# [1; 2; 3] == [1; 2; 3];;  
- : bool = false  
# "foo" == "foo";;  
- : bool = false  
# print_int == print_int;;  
- : bool = true  
# (+) == (+) ;;  
- : bool = false
```

C'est rarement ce que l'on souhaite, on ne l'utilisera donc pas.

→ Pour les TP, l'utilisation de == est interdite.

## Filtrage par motif (*pattern matching*) : `match`

La construction `match <expr> with <patterns>` évalue `<expr>` et regarde chaque motif dans l'ordre définit :

- avec `<pattern> = <condition> -> <résultat>`, si `<condition>` est vrai, le retour de l'expression en entier est le retour de `<résultat>`
- `<condition>` peut être une valeur : on a correspondance par égalité structurelle
- `<condition>` peut être un nouvel identifiant : on a correspondance et on introduit une nouvelle liaison
- `<condition>` peut être sous la forme `... when <expression-booleenne>` : il faut en plus que l'expression booléenne soit vraie
- `<condition>` peut être le mot clé `_` : on a correspondance, toujours

## Filtrage par motif (*pattern matching*) : `match`

```
let is_zero (x : int) : bool =  
  match x with  
  | 0 -> true  
  | _ -> false
```

## Filtrage par motif (*pattern matching*) : match

Un motif peut lister plusieurs valeurs :

```
# let zero_or_one (x : int) : bool =  
  match x with  
    | 0 | 1 -> true  
    | _ -> false ;;  
  
val zero_or_one : int -> bool = <fun>  
  
# zero_or_one 1;;  
- : bool = true  
  
# zero_or_one 2;;  
- : bool = false
```

## Filtrage par motif (*pattern matching*) : `match`

```
let abs (x : int) : int =  
  match x with  
  | y when y > 0 -> y  
  | _ -> -x;;
```

## Délimitation d'expressions : begin et end

Parfois il faut désambiguïser une expression :

```
# let two_zeros (x : int) (y : int) : bool =  
  match x with  
  | 0 -> match y with  
    | 0 -> true  
    | _ -> false  
  | _ -> false ;;
```

```
(* quelques warnings *)
```

```
val two_zeros : int -> int -> bool = <fun>
```

```
# two_zeros 1 0;;
```

```
Exception: Match_failure ("//toplevel//", 2, 2).
```

## Délimitation d'expressions : begin et end

On a en fait défini ceci :

```
let two_zeros (x : int) (y : int) : bool =  
  match x with  
  | 0 -> match y with  
    | 0 -> true  
    | _ -> false  
  | _ -> false
```

C'est incorrect! Les constructions begin et end permettent de clarifier cela.

## Délimitation d'expressions : begin et end

On voulait définir ceci :

```
let two_zeros (x : int) (y : int) : bool =  
  match x with  
  | 0 -> begin match y with  
    | 0 -> true  
    | _ -> false  
  end  
  | _ -> false
```

# Warnings

L'implémentation d'OCaml fourni des *warning* utiles.

Line 6, characters 4-5:

Warning 11 [redundant-case]: this match case is unused.

Lines 2-6, characters 2-14:

Warning 8 [partial-match]: this pattern-matching is not exhaustive.

Here is an example of a case that is not matched:

De manière générale, un *warning* indique un problème réel avec le code.

→ pour les TPs, les warnings sont traités comme des erreurs de compilation

# Commentaires

```
(* ceci est  
   un commentaire  
   (* ceci est un commentaire imbriqué *)  
   *)
```

## Exercice : `valid_date`

Définir une fonction `valid_date` qui prend en argument :

- un nombre `day`
- une chaîne `month`

et renvoie `true` si `day` et `month` représentent une date valide pour une année non-bissextile, où `month` est une chaîne parmi Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Nov, Dec.

## Exercise : valid\_date

```
let valid_date (day : int) (month : string) : bool =  
  let in_month = match month with  
    | "Jan" | "Mar" | "May" | "Jul" | "Sep" | "Dec" -> day <= 31  
    | "Apr" | "Jun" | "Aug" | "Nov" -> day <= 30  
    | "Feb" -> day <= 28  
    | _ -> false in  
  day >= 1 && in_month
```

# Concepts clés

- **OCaml** est un langage fonctionnel strict, non pur, avec typage statique fort
- Types de données de base en OCaml :
  - `bool`, `int`, `float`, `char`, `string` : les classiques
  - `unit` : type ne contenant que `()` comme valeur
  - `'a * 'b` : paires, et leur généralisation, les uplets, collections hétérogènes à taille fixe
  - `'a list` : listes, collections homogènes à taille variable
  - `'a -> 'b` : fonctions
- **Liaisons** : `let`, associe un nom à une valeur (immuable)
  - Peuvent être redéfinies, ce qui *cache* l'ancienne liaison dans un certain *scope*
- Les **fonctions à plusieurs arguments** ne sont que des fonctions appliquées plusieurs fois
- **Filtrage par motif** avec `match`
- **Égalité structurelle** avec `=` et `<>`