

## Programmation Fonctionnelle en OCaml

Chargée de cours: Irène Durand,  
Chargés de TD/TM Frédérique Carrère, Irène Durand,  
Stefka Gueorguieva,  
Jonas Sénizergues.

Cours, 7 séances de 1h20

TM en présentiel, 13 séances de 1h20

Devoir surveillé: Sem 43 (ou 45 si pas d'amphi disponible semaine 43)

TP noté: Sem 49

Travail individuel

<https://moodle.u-bordeaux.fr/course/view.php?id=1208>

MCC: Session1:0.5EX1+ 0.5CC, Session2:0.5EX2 + 0.5CC

CC: DS 1h30 40%, TP noté 1h30 40%, exos Moodle 20%

Examens 1h30

## Art de la programmation

La programmation est un **art**.

Pour progresser:

- ▶ lire du code écrit par des experts
- ▶ lire la littérature sur la programmation
- ▶ programmer
- ▶ maintenir du code écrit par d'autres
- ▶ être bien organisé

## Objectifs de ce cours

- ▶ **Acquérir** les bases de la programmation fonctionnelle
- ▶ **Appliquer** les principes généraux de programmation
  - ▶ Lisibilité du code
  - ▶ Réutilisabilité
  - ▶ Maintenabilité
  - ▶ Efficacité (quand elle ne nuit pas à la lisibilité) ⇒ Complexité
  - ▶ Test

## Paradigmes de programmation

Un paradigme correspond à un style de programmation.

- ▶ impératif
- ▶ fonctionnel
- ▶ objet
- ▶ macro

Un langage peut offrir au programmeur plusieurs paradigmes. Un programme peut utiliser de plusieurs paradigmes.

- ▶ Python: ?
- ▶ Common Lisp: ?
- ▶ OCaml ?
- ▶ C: ?
- ▶ C++: ?
- ▶ Java: ?

## Paradigmes de programmation: exemples

Paradigmes:

- ▶ impératif
- ▶ fonctionnel
- ▶ objet
- ▶ macro

Exemples de langages et leurs paradigmes:

- ▶ Python: impératif, fonctionnel, objet
- ▶ Common Lisp: les 4 paradigmes
- ▶ OCaml: impératif, fonctionnel, objet
- ▶ C: impératif, macros (basiques)
- ▶ C++: impératif, objet, macros (basiques)
- ▶ Java: impératif, objet

## Propriétés d'un langage

Un langage de programmation peut-être

- ▶ interactif/non interactif
- ▶ compilé et/ou interprété
- ▶ dynamiquement typé/statiquement typé

OCaML peut-être

- ▶ utilisé en mode interactif ou en mode batch.



## Paradigme fonctionnel vs paradigme impératif

Le paradigme **fonctionnel** s'oppose au paradigme **impératif**

Paradigme **impératif**:

- ▶ un programme a un **état**: la valeur de l'ensemble de ses variables
- ▶ instruction de base est l'**affectation**
- ▶ structure de contrôle de base est la **boucle tant-que**
- ▶ source de nombreuses difficultés et bugs
- ▶ produit des programmes plus difficiles à comprendre
- ▶ voir bugs célèbres aux conséquences désastreuses

## Quand utiliser le paradigme fonctionnel?

- ▶ quand il y a des enjeux de sécurité (programmes plus sûrs et même dans certains cas prouvés)
- ▶ quand on veut développer rapidement un prototype

L'inconvénient de la programmation fonctionnelle sera principalement la **performance**; mais ce défaut diminue avec des compilateurs de plus en plus **optimisés**.



## Prise de contact avec OCaml

développé à l'INRIA [caml.inria.fr](http://caml.inria.fr)  
disponible sur de nombreuses architectures (Linux, Windows,  
MacOS X, ...)

- ▶ Mode **interactif**:

- ▶ Dans un terminal

```
$ ocaml
OCaml version 4.08.1
# 1 + 2;;
- : int = 3
#
```

- ▶ Avec `utop` dans un terminal

- ▶ sous Emacs, modes Tuareg et utop

- ▶ sous VSCode nouveau plugging permettant l'interactivité

- ▶ Mode non interactif

- ▶ dans un terminal

- ▶ sous vs-code

## Boucle REPL

OCaML est un langage **interactif**. Quand on le lance, on se trouve dans une REPL<sup>1</sup>, dans laquelle on peut taper des **phrases** qui sont soit **expressions** soit des **requêtes**.

Le système boucle sur les trois opérations suivantes:

- ▶ lit (READ) une expression ou une requête (et la met sous une forme interne)
- ▶ évalue (EVAL) la forme interne
- ▶ affiche (PRINT) le résultat sous forme lisible par l'utilisateur

---

<sup>1</sup>Read Eval Print Loop

## Expressions et requêtes

Une **expression** a toujours une **valeur** et toute valeur a un **type**.

Il existe des types de base comme

`int`, `float`, `bool`, `char`, ...

Nous verrons dans un prochain chapitre des types **composés** à l'aide d'opérateurs et comment **nommer** les types ainsi construits.

Le type d'une expression est le type de sa valeur.

Une **requête** fait un **effet de bord** et peut avoir une valeur.

## Expressions

Une **expression** est

- ▶ soit un objet de base (nombre, caractère, booléen, fonction, ...),
- ▶ soit une expression prédéfinie (`if then else`, `let .. in`, `match with`),
- ▶ soit l'application d'une fonction à des arguments qui sont eux-mêmes des expressions<sup>2</sup>.

langage interactif  $\Rightarrow$  pas de **programme principal**  
n'importe quelle fonction peut être appelée

---

<sup>2</sup>Noter la définition récursive d'une expression

## Application d'une fonction

Notation par défaut: **préfixe**

la fonction  $f$  est placée **avant** ses arguments:  $f\ e1\ e2\ \dots$

Ne **pas** séparer les arguments par une virgule, comme en C.

L'application de fonction est **plus prioritaire** que les opérateurs usuels.

```
# max 20 12;;  
- : int = 20  
# max 30 12 * 2;;  
- : int = 60  
# max 30 (12 * 2);;  
- : int = 30
```

## Parenthésage

**par défaut:** `f e1 e2 ... en` équivaut à  
`((f e1) e2) ... en)`.

Pour un autre parenthésage, il faut le préciser:  
`sqrt (max (cos 3.1415) (sin 3.1415))`.

## Opérateurs binaires

opérateurs binaires courants prédéfinis, en particulier opérateurs arithmétiques binaires  $\Rightarrow$  notation **infixe**: opérateur **entre** les deux opérandes.

```
# 2 * (1 + 3);;  
- : int = 8
```

Infixe  $\Rightarrow$  préfixe

Version préfixe d'un opérateur infixe: le mettre en parenthèse:

```
# 5 + 2;;  
- : int = 7  
# 5 - 2;;  
- : int = 3  
# (+) 5 2;;  
- : int = 7  
# (-) 5 2;;  
- : int = 3
```

## Nombres et Expressions numériques

types de base: `int` (entiers), `float` (flottants)

Entiers: 3, 4, 1000

Flottants 2.1, 3.14e-3

## Opérateurs arithmétiques

syntaxe proche du langage mathématique (notation *infixe*)  
À cause du typage, *pas de surcharge* des opérateurs  $\Rightarrow$   
un jeu d'opérateurs pour chaque type:

**int**: +, -, \*, /, **mod**, abs, succ, pred, ...

**float**:

+. , -. , \*. , /. , \*\*, abs\_float, truncate, sqrt, ...

```
# 2.1 +. 4.5;;
```

```
-: float 6.6
```

```
# 2.1 +. 4.5;;
```

```
-: float 6.6
```



## Commentaires

délimités par les caractères `(*` et `*)`  
pas de commentaire de ligne

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

## Expressions conditionnelles

```
(* nombre de solutions d'une équation du 1er degré *)  
(*  $ax + b = 0$  *)  
if a = 0 then  
  if b = 0 then -1  
  else 0  
else 1
```

## Expression `let in`

Pour éviter la duplication d'expressions dans le code, il est conseillé d'utiliser l'expression `let in` qui permet de mémoriser temporairement la valeur d'une ou plusieurs expressions dans des variables temporaires. Ceci permet d'éviter

- ▶ la duplication du code
- ▶ l'évaluation multiple d'une expression

On peut donner des valeurs à plusieurs variables en parallèle à l'aide du mot-clé `and`.

```
# let x = 1 and y = 2 in x + y;;  
- : int = 3
```

Pour des affectations séquentielles, on utilise le `let in` en cascade.

```
# let x = 2 in  
  let y = x * x in y + 1;;  
- : int = 5
```

## Fonction anonyme (fonction sans nom)

La fonction qui à  $x$  associe  $3 * x$  est clairement définie et se note en mathématique:  $x \mapsto 3 * x$

peut être définie avec une expression utilisant le mot `fun` et la syntaxe: `fun x1 x2 ... -> expression`

paramètres de la fonction: `x1`, `x2`, ...

`expression`: corps de la fonction; évaluation donne, après passage des paramètres, la **valeur de retour** de la fonction.

la fonction ci-dessus s'écrit

```
fun x -> 3 * x;;
```

Si on entre cette ligne dans l'interpréteur `OCaML`, l'interpréteur répond

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

Il indique que cette expression est une fonction par `<fun>`. En effet, d'habitude, pour une expression, il affiche la valeur de l'expression, mais pas dans le cas d'une fonction.