

INF201  
Algorithmique et Programmation Fonctionnelle  
Cours 1: Informations Générales et Introduction

Année 2018 - 2019

$f(x)$



## Quelques informations pratiques (groupes IMA)

### Cours :

- ▶ Lundi, 13h30 - 14h45
- ▶ Nicolas Basset  
([bassetni@univ-grenoble-alpes.fr](mailto:bassetni@univ-grenoble-alpes.fr))
- ▶ Supports de cours et autre sur ma page web:  
[www-verimag.imag.fr/~bassetni/INF201](http://www-verimag.imag.fr/~bassetni/INF201)

### TD et TP :

- ▶ IMA-1 : Catherine Parent
- ▶ IMA-2 : Francois Puitg
- ▶ IMA-3 : Cristian Ene
- ▶ IMA-4 : Jean-François Monin
- ▶ IMA-5 : Valentin Gledel
- ▶ IMA-6 : Paul Raynaud

### Début des enseignements

- ▶ Cours : semaine du 13 janvier
- ▶ TD et TP : semaine du 22 janvier

### sauf groupes

- ▶ **IMA-2** → **jeudi 17h00, TD.**
- ▶ **IMA-4** → **vendredi après midi, TD.**

## Semaine type INF 201

- ▶ 1 séance de cours d'1h30
- ▶ 1 séance de TD d'1h30
- ▶ 2 séances de TP
  - ▶ une séance encadrée d'1h30
  - ▶ une séance non encadrée d'1h30
- ▶ Du **travail personnel régulier** (~ 3 heures/semaine minimum)
  - ▶ revoir le cours, **faire des exercices** ...
  - ▶ 1 TP en libre-service (finir les TPs, faire le projet ...)

## Ressources Pédagogiques

- ▶ Des notes de cours (copies des transparents) en ligne
- ▶ Un poly de TD et un poly de TP
- ▶ Une page web
- ▶ un interpréteur OCaml en ligne (<https://try.ocamlpro.com/>)
- ▶ Des références bibliographiques

## Evaluation

- ▶ Des “quicks” en TD (2 à 4)
- ▶ un Devoir Surveillé (DS), pendant la semaine de partiels
- ▶ un Examen Final

## Evaluation

- ▶ Des “quicks” en TD (2 à 4)
- ▶ un Devoir Surveillé (DS), pendant la semaine de partiels
- ▶ un Examen Final

Note Finale = 60%.Examen Final + 20%.CC1 + 20%.CC2

CC1= Quicks + Projet

CC2= Devoir Surveillé

## Evaluation

- ▶ Des “quicks” en TD (2 à 4)
- ▶ un Devoir Surveillé (DS), pendant la semaine de partiels
- ▶ un Examen Final

Note Finale = 60%.Examen Final + 20%.CC1 + 20%.CC2

CC1= Quicks + Projet

CC2= Devoir Surveillé

Une feuille (A4) autorisée pour les examens (à confirmer)

## Objectifs et contenu de INF201

## Découvrir :

⇒ Un **nouveau mode d'expression** : la programmation fonctionnelle

- Fonctionnel** :
- ▶ les fonctions sont les éléments de base du langage
  - ▶ opérations sur les fonctions : composition, paramétrage

**Typage fort** : toute valeur/fonction est typée avant exécution

**polymorphisme**: fonctions génériques

Proche de concepts mathématiques, moins d'erreurs, programmes concis  
Intégré dans de nombreux langages "généraux" (ex: Java)

## Découvrir :

⇒ Un **nouveau mode d'expression** : la programmation fonctionnelle

- Fonctionnel** :
- ▶ les fonctions sont les éléments de base du langage
  - ▶ opérations sur les fonctions : composition, paramétrage

**Typage fort** : toute valeur/fonction est typée avant exécution

**polymorphisme**: fonctions génériques

Proche de concepts mathématiques, moins d'erreurs, programmes concis  
Intégré dans de nombreux langages "généraux" (ex: Java)

⇒ Un **nouveau langage** : OCAML

- ▶ un représentant "moderne" des langages fonctionnels (expressif, produit du code efficace)
- ▶ largement utilisé pour certains domaines d'application
- ▶ conçu et développé en France ...

## Quelques éléments de OCAML (1)

let x = ... in ...

Langage impératif : **affectation** de *variable*

```
x = 42 ;      /* initialisation de la valeur de x */  
x = 43;      /* modification de la valeur de x */
```

## Quelques éléments de OCAML (1)

let x = ... in ...

Langage impératif : **affectation** de *variable*

```
x = 42 ;      /* initialisation de la valeur de x */  
x = 43;      /* modification de la valeur de x */
```

OCaml : **associer** un nom (= *identificateur*) à une valeur dans un contexte

```
let x=42 in ... ;;  
      (* x représente la valeur 42 dans la suite *)
```

## Quelques éléments de OCAML (1)

let x = ... in ...

Langage impératif : **affectation** de *variable*

```
x = 42 ;      /* initialisation de la valeur de x */  
x = 43;      /* modification de la valeur de x */
```

OCaml : **associer** un nom (= *identificateur*) à une valeur dans un contexte

```
let x=42 in ... ;;  
      (* x représente la valeur 42 dans la suite *)
```

**Exemples :**

- ▶ expression avec plusieurs variables

```
y = 5 ;          let y = 5 in  
x = 12 * y + 2 ;      let x = 12 * y + 2 in ... ;;
```

## Quelques éléments de OCAML (1)

let x = ... in ...

Langage impératif : **affectation** de *variable*

```
x = 42 ;      /* initialisation de la valeur de x */  
x = 43;      /* modification de la valeur de x */
```

OCaml : **associer** un nom (= *identificateur*) à une valeur dans un contexte

```
let x=42 in ... ;;  
      (* x représente la valeur 42 dans la suite *)
```

### Exemples :

- ▶ expression avec plusieurs variables

```
y = 5 ;                let y = 5 in  
x = 12 * y + 2 ;      let x = 12 * y + 2 in ... ;;
```

- ▶ incrémenter une variable

```
x = 5 ;                let x = 5 in  
x = x+1 ;              let x = x + 1 in ... ;;
```

## Quelques éléments de OCAML (2)

typage

Un programme “correct” en langage Python :

```
x = 5 ;      /* x est de type entier */  
y = x > 0 ; /* y est de type booleen */  
z = x + y ; /* type de z ? valeur de z ? */
```

## Quelques éléments de OCAML (2)

typage

Un programme “correct” en langage Python :

```
x = 5 ;      /* x est de type entier */  
y = x > 0 ; /* y est de type booleen */  
z = x + y ; /* type de z ? valeur de z ? */
```

En langage OCaml ?

```
let x = 5 in  
  let y = x > 0 in  
    let z = x + y in ... (* erreur de type ! *)
```

## Quelques éléments de OCAML (2)

typage

Un programme “correct” en langage Python :

```
x = 5 ;      /* x est de type entier */  
y = x > 0 ; /* y est de type booleen */  
z = x + y ; /* type de z ? valeur de z ? */
```

En langage OCaml ?

```
let x = 5 in  
  let y = x > 0 in  
    let z = x + y in ... (* erreur de type ! *)
```

Une meilleure solution :

```
let x = 5 in  
  let y = x > 0 in  
    let z = x + (if y then 1 else 0) in ...
```

⇒ typage fort, inférence de type

## Quelques éléments de OCAML (2)

fonctions

Valeur absolue :  $|x|$

## Quelques éléments de OCAML (2)

fonctions

Valeur absolue :  $|x|$

`val_abs :  $\mathbb{R} \rightarrow \mathbb{R}^+$`  (le résultat est un réel positif)

## Quelques éléments de OCAML (2)

### fonctions

Valeur absolue :  $|x|$

`val_abs :  $\mathbb{R} \rightarrow \mathbb{R}^+$`  (le résultat est un réel positif)

```
let val_abs (x : float) : float =  
    if x >= 0 then x else -. x ;;
```

## Quelques éléments de OCAML (2)

### fonctions

Valeur absolue :  $|x|$

`val_abs :  $\mathbb{R} \rightarrow \mathbb{R}^+$`  (le résultat est un réel positif)

```
let val_abs (x : float) : float =  
    if x >= 0 then x else -. x ;;
```

Racines d'un polynôme du 2nd degré :  $ax^2 + bx + c$

## Quelques éléments de OCAML (2)

### fonctions

Valeur absolue :  $|x|$

$\text{val\_abs} : \mathbb{R} \rightarrow \mathbb{R}^+$  (le résultat est un réel positif)

```
let val_abs (x : float) : float =  
    if x >= 0 then x else -. x ;;
```

Racines d'un polynôme du 2nd degré :  $ax^2 + bx + c$

$\text{racine} : (\mathbb{R} \times \mathbb{R} \times \mathbb{R}) \rightarrow (\mathbb{R} \times \mathbb{R} \cup \{\perp\})$

(le résultat est un couple de réels, ou n'est pas défini)

## Quelques éléments de OCAML (2)

### fonctions

Valeur absolue :  $|x|$

`val_abs :  $\mathbb{R} \rightarrow \mathbb{R}^+$`  (le résultat est un réel positif)

```
let val_abs (x : float) : float =  
    if x >= 0 then x else -. x ;;
```

Racines d'un polynôme du 2nd degré :  $ax^2 + bx + c$

`racine :  $(\mathbb{R} \times \mathbb{R} \times \mathbb{R}) \rightarrow (\mathbb{R} \times \mathbb{R} \cup \{\perp\})$`

(le résultat est un couple de réels, ou n'est pas défini)

```
let racine (a : float) (b : float) (c : float) :  
    float * float =  
    let delta = b*.b -. 4.*.a*.c in  
    if delta >= 0. then  
        ((-.b -. sqrt(delta))./(2.*.a),  
         (-.b +. sqrt(delts))./(2.*.a))  
    else  
        failwith ("pas de racines") ;;
```

## Quelques éléments de OCAML (2 suite)

fonctions (suite)

### Fonction affine :

Etant donnés deux réels  $a$  et  $b$ , produire la fonction  $x \mapsto a * x + b$

## Quelques éléments de OCAML (2 suite)

fonctions (suite)

### Fonction affine :

Etant donnés deux réels  $a$  et  $b$ , produire la fonction  $x \mapsto a * x + b$

`affine :  $\mathbb{R} \times \mathbb{R} \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$`  (le résultat est une fonction)

## Quelques éléments de OCAML (2 suite)

fonctions (suite)

### Fonction affine :

Etant donnés deux réels  $a$  et  $b$ , produire la fonction  $x \mapsto a * x + b$

`affine :  $\mathbb{R} \times \mathbb{R} \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$`  (le résultat est une fonction)

```
let affine (a : float) (b : float) : float-> float =  
  fun x -> a *. x +. b
```

## Quelques éléments de OCAML (2 suite)

fonctions (suite)

### Fonction affine :

Etant donnés deux réels  $a$  et  $b$ , produire la fonction  $x \mapsto a * x + b$

`affine :  $\mathbb{R} \times \mathbb{R} \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$`  (le résultat est une fonction)

```
let affine (a : float) (b : float) : float-> float =  
    fun x -> a *. x +. b
```

### Composition de fonctions :

Etant données deux fonctions  $f$  et  $g$  de  $\mathbb{N} \rightarrow \mathbb{N}$ , écrire la fonction  $f \circ g$

## Quelques éléments de OCAML (2 suite)

### fonctions (suite)

#### Fonction affine :

Etant donnés deux réels  $a$  et  $b$ , produire la fonction  $x \mapsto a * x + b$

`affine :  $\mathbb{R} \times \mathbb{R} \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$`  (le résultat est une fonction)

```
let affine (a : float) (b : float) : float-> float =  
  fun x -> a *. x +. b
```

#### Composition de fonctions :

Etant données deux fonctions  $f$  et  $g$  de  $\mathbb{N} \rightarrow \mathbb{N}$ , écrire la fonction  $f \circ g$

`compo :  $(\mathbb{N} \rightarrow \mathbb{N}) \times (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$`

(les arguments sont des fonctions, le résultat est une fonction)

## Quelques éléments de OCAML (2 suite)

### fonctions (suite)

#### Fonction affine :

Etant donnés deux réels  $a$  et  $b$ , produire la fonction  $x \mapsto a * x + b$

`affine :  $\mathbb{R} \times \mathbb{R} \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$`  (le résultat est une fonction)

```
let affine (a : float) (b : float) : float-> float =  
    fun x -> a *. x +. b
```

#### Composition de fonctions :

Etant données deux fonctions  $f$  et  $g$  de  $\mathbb{N} \rightarrow \mathbb{N}$ , écrire la fonction  $f \circ g$

`compo :  $(\mathbb{N} \rightarrow \mathbb{N}) \times (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$`

(les arguments sont des fonctions, le résultat est une fonction)

```
let compo (f : int -> int) (g : int -> int) : int-> int =  
    fun x -> g (f x) ;;
```

⇒ En OCaml, les fonctions sont des valeurs comme les autres !

## Quelques éléments de OCAML (3)

récurtivité

Factorielle d'un entier :  $n! = n \times (n - 1) \times (n - 2) \times \dots \times 2 \times 1$

## Quelques éléments de OCAML (3)

récurtivité

Factorielle d'un entier :  $n! = n \times (n - 1) \times (n - 2) \times \dots \times 2 \times 1$

### Mode impératif (Python)

```
def fact(n):  
    r=1  
    while n>0:  
        r=r*n  
        n=n-1  
    return r
```

## Quelques éléments de OCAML (3)

### récurtivité

Factorielle d'un entier :  $n! = n \times (n - 1) \times (n - 2) \times \dots \times 2 \times 1$

### Mode impératif (Python)

```
def fact(n):  
    r=1  
    while n>0:  
        r=r*n  
        n=n-1  
    return r
```

### Mode fonctionnel (OCaml)

**fact** :  $\mathbb{N} \rightarrow \mathbb{N}$

```
let rec fact (n : int) : int =  
    if (n=0 || n=1) then 1 else n * fact (n-1);;
```

## Quelques éléments de OCAML (3)

### récurtivité

Factorielle d'un entier :  $n! = n \times (n - 1) \times (n - 2) \times \dots \times 2 \times 1$

### Mode impératif (Python)

```
def fact(n):
    r=1
    while n>0:
        r=r*n
        n=n-1
    return r
```

### Mode fonctionnel (OCaml)

**fact** :  $\mathbb{N} \rightarrow \mathbb{N}$

```
let rec fact (n : int) : int =
    if (n=0 || n=1) then 1 else n * fact (n-1);;
```

### Mode fonctionnel (Python)

```
def Fact(n):
    if n==0:
        return(1)
    else
        return(n*Fact(n-1))
```

# Plan du cours

4 grandes parties

1. Types, expressions, fonctions
2. Récursivité
3. Ordre supérieur
4. Structures arborescentes

## Références

- ▶ Guy Cousineau et Michel Mauny. *Approche fonctionnelle de la programmation*. Ediscience (Collection Informatique), Paris, 1995, ISBN 2-84074-114-8.
- ▶ Emmanuel Chailloux, Pascal Manoury et Bruno Pagano. *Développement d'applications avec Objective Caml*. Editions O'Reilly, Paris, 2000, ISBN 2-84177-121-0.
- ▶ Xavier Leroy et Pierre Weis. *Manuel de référence du langage Caml*. InterEditions, Paris, 1993, ISBN 2-7296-0492-8. Version électronique
- ▶ Le site web Ocaml de l'Inria
- ▶ Le manuel de référence Ocaml
- ▶ L'interpréteur OCaml (en ligne ou non)
- ▶ Conventions de programmation en Ocaml:
  - ▶ <http://caml.inria.fr/resources/doc/guides/guidelines.fr.html>
  - ▶ [http://www.seas.upenn.edu/~cis500/cis500-f06/resources/programming\\_style.html](http://www.seas.upenn.edu/~cis500/cis500-f06/resources/programming_style.html)

# Remerciements

Ce cours a été mis en place par Michaël Périn et François Puitg

Ces transparents de cours sont basés sur :

- ▶ les transparents de Laurent Mounier eux-mêmes basés sur
  - ▶ les transparents du cours INF121 en anglais par Yliès Falcone
  - ▶ des notes de cours de Pascal Lafourcade