

Chapitre 0 - Introduction

INF6120: Programmation fonctionnelle et logique

Quentin Stiévenart

Université du Québec à Montréal

v251



Meta

Informations pratique

- **Titre du cours** : Programmation fonctionnelle et logique
- **Sigle** : INF6120
- **Département** : Informatique
- **Enseignant** : Quentin Stiévenart
- **Courriel** : stievenart.quentin@uqam.ca
- **Bureau** : PK-4735
- **Site du cours** : <https://inf6120.uqam.ca>
- **Plan de cours** : Voir sur https://info.uqam.ca/plan_cours/

Objectifs du cours

Description du cours :

Faire l'acquisition de nouvelles techniques et stratégies de programmation par l'apprentissage des concepts fondamentaux des langages de programmation fonctionnels et logiques. Apprécier l'apport de ces langages au développement logiciel.

Pré-réquis

- INF3105 - Structure de données et algorithmes

Contenu du cours

2 langages :

- OCaml
- Prolog

Mais ce n'est pas un cours de langages, c'est un cours de paradigmes !

2 paradigmes :

- Programmation fonctionnelle
- Programmation logique

Contenu du cours (1)

Programmation fonctionnelle

- Historique
- Effet de bord, référence immuable, évaluation des expressions
- Répartition par appariements de motifs (*pattern matching*)
- Fonctions comme argument et valeur de retour
- Stratégie d'évaluation des arguments
- Polymorphisme et déduction des types
- Fermetures, curryage et application partielle
- Fonction d'ordre supérieur
- Concurrence et parallélisme en programmation fonctionnelle
- Monades

Contenu du cours (2)

Programmation logique

- Règles, inférence et clauses de Horn
- Recherche par retour arrière et déduction
- Unification et résolution
- Coupure et négation
- Programmation par contraintes, consistance d'arc, de chemins et problèmes de satisfaction de contraintes
- Comparaison des approches de programmation logique et par contraintes

Structure du cours

- Chapitre 0 : Introduction
- Chapitre 1 : Concepts de base
- Chapitre 2 : Fonctions et récursivité
- Chapitre 3 : Types fonctionnels
- Chapitre 4 : Fonctions d'ordre supérieur
- Chapitre 5 : Encapsulation et modules
- Chapitre 6 : Tests et preuves
- Chapitre 7 : Programmation logique
- Chapitre 8 : Interprétation d'un langage fonctionnel
- Chapitre 9 : Unification et typage
- ~~Chapitre 10 : Interprétation d'un langage logique~~
- Chapitre 11 : Programmation par contraintes

Modalités d'évaluation

Description	Pondération	Échéance
Quiz 1	2.5%	Semaine 4
TP0	5%	Semaine 6
Examen intra	32.5%	Semaine 8
TP1	15%	Semaine 10
Quiz 2	2.5%	Semaine 12
TP2	10%	Semaine 14
Examen final	32.5%	Semaine 15

Modalités d'évaluation

Quiz

Sur Moodle, à faire chez soi, seul

Examen

- À l'heure du cours la semaine correspondante.
- La salle sera annoncée à l'avance.
- Accès à une page de notes manuscrites recto-verso (2 faces) au format lettre ou A4.

Travaux pratiques

Individuels, l'utilisation d'IA générative proscrite

Seuil

Pas de double seuil ! Réussite = 50%

Référence

Programmation fonctionnelle

Clarkson, M. **OCaml Programming: Correct + Efficient + Beautiful**.

- couvre 80% de la matière
- chaque chapitre est également expliqué en vidéo

Programmation logique

Flach, P. **Simply Logical**.

- va plus en détail que le cours
- un bon complément

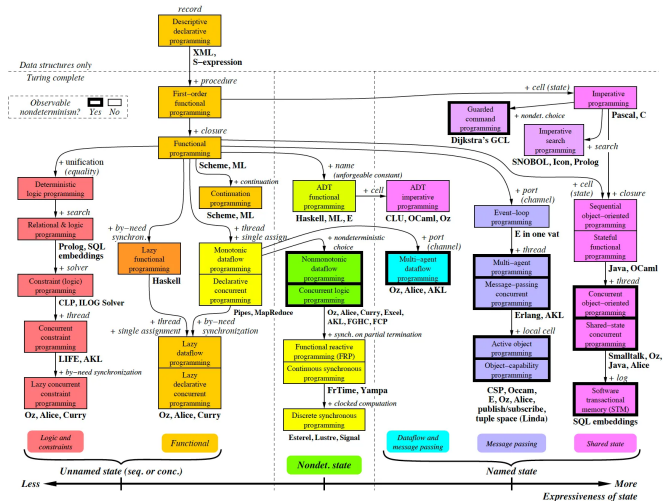
Paradigme fonctionnel

Paradigme

*Un paradigme est — en épistémologie et dans les sciences humaines et sociales — une **représentation du monde**, une manière de voir les choses, un modèle cohérent du monde qui repose sur un fondement défini (matrice disciplinaire, modèle théorique, courant de pensée).*

– Wikipédia

Exemples de paradigmes (Van Roy, 2009)



Source : "Programming Paradigms for Dummies", P. Van Roy

Paradigme

A language that doesn't affect the way you think about programming is not worth knowing.
– Alan Perlis

Paradigme fonctionnel

- Élement de base: **fonctions**
- Paradigme **déclaratif**: on évalue des expressions vers des valeurs, plutôt qu'utiliser des commandes impératives qui modifient un état – **quoi** vs. **comment**
- Absence d'**état**
- Les fonctions sont **de première classe** et **d'ordre supérieur** : ce sont des valeurs
 - On peut les **nommer**
 - On peut les **passer en argument**
 - On peut les **renvoyer**

Exemple: factorielle

Mathématiques

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n - 1)! & \text{sinon} \end{cases}$$

Exemple: factorielle

Mathématiques

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n - 1)! & \text{sinon} \end{cases}$$

OCaml

```
let rec fact = function  
| 0 -> 1  
| n -> n * (fac (n - 1))
```

Exemple: factorielle

Java (itératif)

```
public static int factorial(int n) {  
    int fact = 1;  
    for (int i = 0; i <= n; i++) {  
        fact = fact * i;  
    }  
    return fact  
}
```

Java (récuratif)

```
public static int factorial(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n - 1);  
}
```

Exemple: tri par insertion

Java

```
public static void isort(int[] array) {  
    int n = array.length;  
    for (int j = 1; j < n; j++) {  
        int key = array[j];  
        int i = j - 1;  
        while (i > -1 && array[i] > key) {  
            array[i + 1] = array[i];  
            i--;  
        }  
        array[i + 1] = key;  
    }  
}
```

Exemple: tri par insertion

OCaml

```
let rec insert x = function
| [] -> [x]
| y :: ys ->
  if x < y then
    x :: y :: ys
  else
    y :: (insert x ys)

let rec isort = function
| [] -> []
| x :: xs -> insert x (isort xs)
```

Historique des langages fonctionnels

Historique : langages fonctionnels théoriques

Lambda calcul (1930s, Alonzo Church)

$M, N \in \text{Term} ::= x$

variable

| $\lambda x.M$

abstraction

| $M N$

application

Historique : langages fonctionnels théoriques

Nombres en lambda calcul

$zero = \lambda f. \lambda x. x$

$one = \lambda f. \lambda x. f \ x$

$two = \lambda f. \lambda x. f \ (f \ x)$

$next = \lambda n. \lambda f. \lambda x. f \ ((n \ f) \ x)$

$add = \lambda m. \lambda n. \lambda f. \lambda x. (((m \ next) \ n) \ f) \ x$

$mult = \lambda m. \lambda n. \lambda x. m \ (n \ x)$

Historique : langages fonctionnels théoriques

Conditions en Lambda calcul

$$true = \lambda x. \lambda y. x$$

$$false = \lambda x. \lambda y. y$$

$$and = \lambda p. \lambda q. (p \ q) \ false$$

Boucles en Lambda calcul

$$Y = \lambda f. (\lambda x. f \ (x \ x)) (\lambda x. f \ (x \ x))$$

$$fac = Y (\lambda f. \lambda n. (isZero \ n) \ 1 \ (mult \ n \ (f \ (pred \ n))))$$

Historique : langages fonctionnels théoriques

Logique combinatoire (1920s, Schönfinkel & Curry)

$$\mathbf{I} \ x = x$$

$$\mathbf{K} \ x \ y = x$$

$$\mathbf{S} \ x \ y \ z = x \ z \ (y \ z)$$

(Jeu : **Combinatris**)

Historique : langages fonctionnels pratiques

LISP (1957, McCarthy)

```
(defun factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

Historique : langages fonctionnels pratiques

LISP (1957, McCarthy)

```
(DERIV (LAMBDA (E X)                                072
  (COND ((ATOM E) (COND ((EQ E X) 1) (T 0)))         073
        ((OR (EQ (CAR E) (QUOTE PLUS)) (EQ (CAR E) (QUOTE DIFFERENCE))) 074
          (LIST (CAR E) (DERIV (CADR E) X) (DERIV (CAADR E) X)))      075
        ((EQ (CAR E) (QUOTE TIMES))                076
          (LIST (QUOTE PLUS)                        077
                (LIST (CAR E) (CADR E) (DERIV (CADR E) X))           078
                (LIST (CAR E) (CAADR E) (DERIV (CAADR E) X))))        079
        ((EQ (CAR E) (QUOTE QUOTIENT))              080
          (LIST (CAR E)                                081
                (LIST (QUOTE DIFFERENCE)              082
                      (LIST (QUOTE TIMES) (CADR E) (DERIV (CADR E) X)) 083
                      (LIST (QUOTE TIMES) (CAADR E) (DERIV (CAADR E) X))) 084
                      (LIST (QUOTE TIMES) (CAADR E) (CAADR E))))      085
        ((EQ (CAR E) (QUOTE EXPT))                  086
          (LIST (QUOTE TIMES)                        087
                (LIST (QUOTE TIMES) (CAADR E)         088
                      (COND ((EQUAL (CAADR E) 2) (CAADR E))          089
                            (T (LIST (CAR E) (CADR E) (SUB1 (CAADR E)))) 090
                      (DERIV (CADR E) X)))                        091
                (T NIL))))                                       092
```

Extrait du manuel de **LISP 1.5 Primer** (1967)

Historique : langages fonctionnels pratiques

LISP Machine (années 1980)



Source: [Wikimedia](#)

Historique : langages fonctionnels pratiques

APL (1966, Iverson)

```
init ← (← ? #⊖) { ⊢  
alloc ← ↑⊖I.⊖(= ⌊/ )⊖(+ / ⊖⊖⊖-⊖1)  
step ← (+ / ÷ #)⊖>⊖(alloc⊖1 2 <⊖ ⊢)  
kmc ← ⊢ step⊖_ init
```

Source: wmjn.github.io

Historique : langages fonctionnels pratiques

ML (1973, Milner)

```
fun fac (0 : int) : int = 1  
  | fac (n : int) : int = n * fac (n - 1)
```

Historique : langages fonctionnels pratiques

Scheme (1975, Steele)

```
(define (double x)
  (* 2 x))
```

Utilisation de langages fonctionnels aujourd'hui

- Haskell utilisé par Facebook pour combattre le spam ([source](#))
- Erlang utilisé par WhatsApp ([source](#))
- Elixir utilisé par Discord ([source](#))
- Scala utilisé par LinkedIn, Twitter, Netflix, ... ([source](#))
- OCaml utilisé par Docker, Facebook, ... ([source](#))
- ...

Aspects fonctionnels des langages *modernes*

Fonctions anonymes (*lambda*)

Possibilité de créer une fonction sans la nommer

Java (Java 8, 2014)

```
(int x, int y) -> { return x + y; }  
(x, y) -> x + y
```

Python (1.0, 1994):

```
lambda x, y: x + y
```

Fonctions anonymes (*lambda*)

C++ (C++11, 2011)

```
[] (int x, int y) { return x + y; }
```

JavaScript

```
function(x, y) { return x * x }  
(x, y) => x * y;
```

Fonctions de première classe (*first-class functions*)

Possibilité d'utiliser une fonction comme une valeur

JavaScript

```
function hello() {  
    console.log("Hello");  
}  
hi = hello;  
hi();
```

Fonctions de première classe (*first-class functions*)

Possibilité d'utiliser une fonction comme une valeur

C

```
int add1(int x) {  
    return x + 1;  
}  
  
int main() {  
    int (*f)(int) = &add1;  
    return f(5);  
}
```

Fonctions d'ordre supérieur (*higher-order functions*)

Fonction qui prend d'autres fonctions en argument, ou renvoie une fonction

Java

```
Function<IntUnaryOperator, IntUnaryOperator>  
    twice = f -> f.andThen(f);
```

Fonctions d'ordre supérieur (*higher-order functions*)

Fonction qui prend d'autres fonctions en argument, ou renvoie une fonction

JavaScript

```
function sayHello() {  
    return "Hello, ";  
}  
  
function greeting(helloMessage, name) {  
    console.log(helloMessage() + name);  
}  
  
greeting(sayHello, "JavaScript!");
```

Fonctions d'ordre supérieur (*higher-order functions*)

Fonction qui prend d'autres fonctions en argument, ou renvoie une fonction

C

```
void sort(int (*cmp)(int, int), int array[]) {  
    // ...  
}  
  
typedef int (*ptr)(int*);  
ptr do_twice(int (*f)(int)) {  
    // Requiert les extensions GNU de C99, c'est hors standard  
    return ({ int g(int x) { f(f(x)); } &g; });  
}
```

Immuabilité (*immutability*)

Valeurs constantes qui ne peuvent pas être modifiées.

JavaScript

```
const immutableValue = 1;
```

```
let obj = { a: { b: 1 } };
```

```
Object.freeze(obj); // shallow freeze
```

Immuabilité

TypeScript

```
interface Todo {  
  title: string;  
}
```

```
const todo: Readonly<Todo> = {  
  title: "Delete inactive users",  
};
```

```
// todo.title = "Hello"; // does not compile
```

Évaluation paresseuse

On n'évalue que ce qui est demandé, quand c'est nécessaire

Python

```
range(65536)
```

Scala

```
def foo(x: => Int, y: Int) = {  
    // ...  
}  
foo(fac(5000), 10);
```

Évaluation paresseuse

Souvent simulée avec des *thunks*

JavaScript

```
function foo(x, y) {  
  if (...) {  
    return x();  
  } else {  
    ...  
  }  
}  
foo(() => something(), 5);
```

Évaluation paresseuse

Certains langages introduisent des *générateurs*, une autre forme d'évaluation paresseuse

Python

```
>>> def numbers(n):  
...     yield n  
...     yield from numbers(n+1)  
...  
>>> g = numbers(0)  
>>> [next(g) for i in range(10)]  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Récurtivité

La plupart des langages supportent la récursivité, mais pas de façon optimale !

Python

```
def f(n):  
    if n == 0:  
        return  
    f(n - 1)
```

Récurtivité

La plupart des langages supportent la récursivité, mais pas de façon optimale !

Python

```
>>> f(10)
```

```
>>> f(1000)
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
File "<stdin>", line 4, in f
```

```
File "<stdin>", line 4, in f
```

```
File "<stdin>", line 4, in f
```

```
[Previous line repeated 995 more times]
```

```
File "<stdin>", line 2, in f
```

```
RecursionError: maximum recursion depth exceeded
```

```
in comparison
```

Polymorphisme paramétrique

Un type qui dépend d'un autre type arbitraire.

Parfois appelé « générique » (Java, Go)

Java (5, 2004)

```
public interface List<E> {  
    void add(E x);  
    Iterator<E> iterator();  
}  
  
public interface Iterator<E> {  
    E next();  
    boolean hasNext();  
}
```

Polymorphisme paramétrique

Un type qui dépend d'un autre type arbitraire.

Parfois appelé « générique » (Java, Go)

Go (1.18, 2022)

```
func (lst *List[T]) GetAll() []T {  
    var elems []T  
    for e := lst.head; e != nil; e = e.next {  
        elems = append(elems, e.val)  
    }  
    return elems  
}
```

Polymorphisme paramétrique borné

Un type qui dépend d'un autre type qui respecte certaines opérations

Java

```
public static <S extends Comparable> S min(S a, S b) {  
    if (a.compareTo(b) <= 0)  
        return a;  
    else  
        return b;  
}
```

Polymorphisme paramétrique borné

Un type qui dépend d'un autre type qui respecte certaines opérations

Rust

```
pub fn notify<T: Summary>(item: &T) {  
    println!("Breaking news! {}", item.summarize());  
}
```

Higher-Kinded Types (HKT)

Un type qui dépend d'un autre type générique.

Scala

```
trait Collection[T[_]] {  
  def wrap[A](a: A): T[A]  
  def first[B](b: T[B]): B  
}
```

Higher-Kinded Types (HKT)

Un type qui dépend d'un autre type générique.

C++ (*template parameters*)

```
template <template <typename> class m>
struct Monad {
    template <typename a>
    static m<a> mreturn(const a&);

    template <typename a, typename b>
    static m<b> mbind(const m<a>&, m<b>(*)(const a&));
};
```

Monades

Patron de conception fonctionnel pour composer des calculs

Scala

```
for {  
  order    <- os.loadOrder("username")  
  item     <- is.loadItem(order)  
  purchase <- ps.purchaseItem(item)  
  log      <- ps.logPurchase(purchase)  
} yield log
```

Ramasse-miettes (*garbage collector*)

Introduit dans Lisp (1959) pour simplifier la gestion de la mémoire.

Présent dans la majorité des langages modernes.

Boucle d'évaluation (*read-eval-print loop*, REPL)

Boucle d'interaction introduite par Lisp :

- ① lecture
- ② évaluation
- ③ écriture
- ④ retour à l'étape 1

Scheme

```
> (define x 1)
> (+ x 1)
2
```

Boucle d'évaluation (*read-eval-print loop*, REPL)

Aujourd'hui présent dans de nombreux langages (Python, JavaScript, ...)

Python

```
Python 3.11.7 (main, Jan 29 2024, 16:03:57) [GCC 13.2.1 20230801] on linux  
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> x = 1
```

```
>>> x + 1
```

```
2
```

Boucle d'évaluation (*read-eval-print loop*, REPL)

Aujourd'hui présent dans de nombreux langages (Python, JavaScript, ...)

JavaScript

```
Welcome to Node.js v16.19.1.  
Type ".help" for more information.  
> x = 1  
1  
> x + 1  
2  
... ou dans le navigateur (Ctrl-Maj-I)
```

Le shell comme langage fonctionnel

Shell

```
ps aux | awk '{print $2}' | sort -n | xargs echo
```

Appels de fonction

```
xargs(echo, sort(n, awk('print $2', ps(aux))))
```

Shell: calculs paresseux

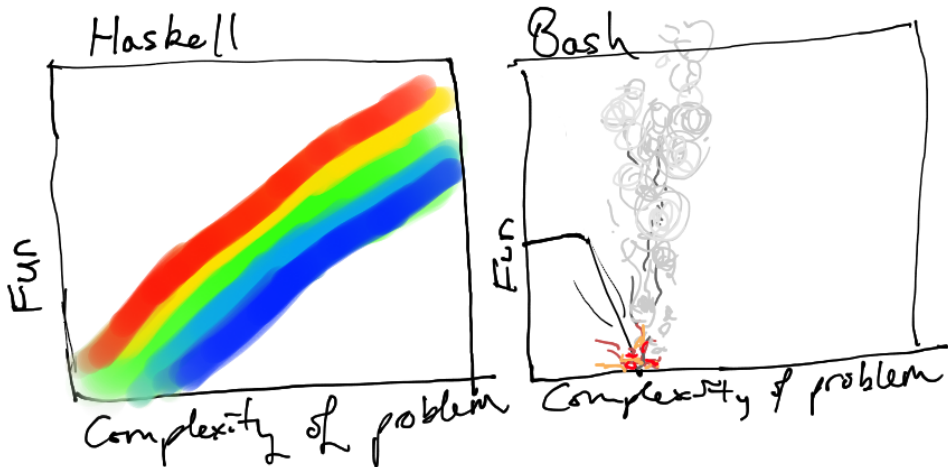
Shell

```
yes | head -n 5
```

Haskell

```
take 5 (repeat "y")
```

Unix pipes



Source: xahlee.info

Concepts clés

- **Paradigme** : une représentation du monde; en programmation, il existe de nombreux paradigmes, dont les paradigmes impératif, orienté-objet, et fonctionnel
- **Fonction** : élément essentiel du paradigme fonctionnel, elles sont
 - de **première classe** : manipulables comme toute autre valeur du langage
 - d'**ordre supérieur** : elles peuvent être prises en argument et renvoyées par d'autres fonctions
- **Immuabilité** : autre élément essentiel du paradigme fonctionnel, les valeurs ne peuvent pas être modifiées