

# Programmation Fonctionnelle

**Manuel Clergue**

*Programmer doit être amusant !*

*Programmer doit être beau !*



## **Chapitre 2 :**

**0 - Formes spéciales**

**1 - Collections ordonnées**

# 0 - Formes spéciales

## a - def, fn et defn

Ce sont des formes spéciales :

leur évaluation ne suit pas celle des expressions clojure

La forme `def` permet d'associer une valeur (au sens large) avec un symbole :

```
user=> x
Syntax error compiling at (REPL:0:0).
Unable to resolve symbol: x in this context
```

```
user=> (def x 7)
#'user/x
```

```
user=> x
7
```

```
user=> (+ x x)
14
```

# a - def, fn et defn

Ce sont des formes spéciales :

leur évaluation ne suit pas celle des expressions clojure

La forme **def** permet d'associer une valeur (au sens large) avec un symbole :

```
user=> x
Syntax error compiling at (REPL:0:0).
Unable to resolve symbol: x in this context
```

```
user=> (def x 7)
#user/x
```



namespace

```
user=> x
7
```

```
user=> (+ x x)
14
```

a - def, fn **et** defn

La forme `fn` permet de créer une fonction (anonyme) :

```
user=> (fn [x] (* x x))  
#object[user$eval140$fn__141 0xcf4dd3df "user$eval140$fn__141@cf4dd3df"]
```

## a - def, fn et defn

La forme **fn** permet de créer une fonction (anonyme) :

```
user=> (fn [x] (* x x))  
#object[user$eval140$fn__141 0xcf4dd3df "user$eval140$fn__141@cf4dd3df"]
```



**a** - def, fn **et** defn

Usage :

```
user=> ((fn [x] (* x x)) 5)  
25
```



## a - def, fn et defn

### Usage :

```
user=> ((fn [x] (* x x)) 5)  
25
```

### Association à un symbole :

```
user=> (def carre (fn [x] (* x x)))  
#'user/carre
```

```
user=> (carre 5)  
25
```

```
user=> (carre 4)  
16
```

# a - def, fn et defn

Forme **defn**, définition compacte de fonction :

```
user=> (defn carre [x] (* x x))  
#'user/carre
```

Définition de fonctions avec plusieurs arités :

```
user=> (defn plus  
  ([] 0)  
  ([x] x)  
  ([x y] (+ x y)))  
#'user/plus
```

```
user=> (plus)  
0
```

```
user=> (plus 5)  
5
```

```
user=> (plus 5 10)  
15
```

```
user=> (plus 5 10 15)
```

```
Execution error (ArityException) at user/eval181 (REPL:1).  
Wrong number of args (3) passed to: user/plus
```

## a - def, fn et defn

Définition de fonctions d'arité variable (fonctions variadiques):

```
user=> (defn plus_v
  ([] 0)
  ([x] x)
  ([x y & l] (+ x y)))
#'user/plus_v
```

```
user=> (plus_v 5 10 15)
15
```

```
user=> (plus_v 1 2 3 4 5 6)
3
```

Les arguments supplémentaires sont récupérés dans une liste :

```
user=> (defn av [x & y] (println "le premier arg : " x) (println "les autres : "y))
#'user/av
```

```
user=> (av 1 2 3 4 5 6)
le premier arg : 1
les autres : (2 3 4 5 6)
nil
```

## b - let

La forme spéciale **let** permet de faire des liaisons (symbole valeur) locales :

```
user=> (let [y 1] (+ y 1))
```

```
2
```

```
user=> y
```

```
Syntax error compiling at (REPL:0:0).
```

```
Unable to resolve symbol: y in this context
```

## b - let

La forme spéciale **let** permet de faire des liaisons (symbole valeur) locales :

```
user=> (let [y 1] (+ y 1))
```

```
2
```

```
user=> y
```

```
Syntax error compiling at (REPL:0:0).
```

```
Unable to resolve symbol: y in this context
```

## b - let

La forme spéciale **let** permet de faire des liaisons (symbole valeur) locales :

```
user=> (let [y 1] (+ y 1))  
2
```

```
user=> y
```

```
Syntax error compiling at (REPL:0:0)  
Unable to resolve symbol: y in this context
```



## b - let

La forme spéciale **let** permet de faire des liaisons (symbole valeur) locales :

```
user=> (let [y 1] (+ y 1))  
2
```

```
user=> y  
Syntax error compiling at (REPL:0:0).  
Unable to resolve symbol: y in this context
```

On peut définir plusieurs liaisons et les utiliser dans plusieurs expressions :

```
user=> (let [x 1  
            y x] (println x "+" y) (+ x y))  
  
1 + 1  
2
```

```
user=> (let [x 1,y x] (println x "+" y) (+ x y))  
1 + 1  
2
```

# 1 - Collections ordonnées



# a - La séquence vector

La structure vector est une collection de données séquentielle et indexée

```
user=> (def tab [\a \b \c])  
#'user/tab
```

```
user=> tab  
[\a \b \c]
```

Accès par l'indice :

```
user=> (get tab 1)  
\b
```

```
user=> (get tab 6)  
nil
```

Nombre d'éléments :

```
user=> (count tab)  
3
```

# a - La séquence vector

## Constructeur :

```
user=> (vector 1 2 3)
[1 2 3]
```

```
user=> (apply vector '(1 2 3))
[1 2 3]
```

```
user=> (vec '(1 2 3))
[1 2 3]
```

## Ajout d'éléments :

```
user=> (conj tab 1 2 3)
[\a \b \c 1 2 3]
```

## Les objets sont immutables :

```
user=> tab
[\a \b \c]
```

# a - La séquence vector

## Concaténation :

```
user=> (into [1 2 3] [4 5 6])  
[1 2 3 4 5 6]
```

## Slicing :

```
user=> (subvec [1 2 3 4 5 6] 1 4)  
[2 3 4]
```

```
user=> (subvec [1 2 3 4 5 6] 1)  
[2 3 4 5 6]
```

## b - La séquence `list`

La structure de donnée `list` est une collection ordonnée d'éléments :

```
user=> (def l1 '(1 2 3))  
#'user/l1
```

```
user=> l1  
(1 2 3)
```

Un objet `list` se parcourt avec les fonctions **`first`** et **`rest`** :

```
user=> (first l1)  
1  
user=> (rest l1)  
(2 3)
```

C'est une liste chaînée : une liste est composée d'un élément et d'une liste

Il existe une liste spéciale, la liste vide : `'()`

## b - La séquence `list`

La fonction `empty?` permet de savoir si la liste est vide :

```
user=> (empty? '())  
true
```

```
user=> (empty? '(1))  
false
```

On compte les éléments avec `count` :

```
user=> (count l1)  
3
```

La fonction `conj` ajoute les éléments (en tête de liste) :

```
user=> (conj l1 0)  
(0 1 2 3)
```

```
user=> (conj l1 0.5 0)  
(0 0.5 1 2 3)
```

## b - La séquence `list`

Concaténation :

```
user=> (concat l1 '(4 5 6))  
(1 2 3 4 5 6)
```

## 2 - Récursivité

# a - Principe

La récursivité est le moyen de faire des boucles en programmation fonctionnelle.  
Exemple du calcul de factoriel :

**Dans un langage de programmation impératif les variables sont modifiées**

```
int n, f;           // a
n= 3;               // b
f=1;                // c
while(n != 0){
    f = n * f;       // d
    n = n - 1;       // e
}
return f;
```

**Comment faire avec des langages dont les objets sont immutables ?**



## a - Principe

Avec la programmation fonctionnelle, ce sont les valeurs des liaisons qui vont être modifiées lors des appels **récur­sifs** :

```
user=> (defn fac [n]
  (if (zero? n)
      1
      (* n (fac (- n 1)))))
#'user/fac
```

```
user=> (fac 123N)
1214630436702532967576624324188129585545421708848338231532891816182923589236216766883115696
0612640202170735835221294047782591091570411651472186029519906261646730733907419814952960000
000000000000000000000000N
```

C'est un concept beaucoup plus général que celui de boucle !

## a - Principe

Cela fonctionne de la même façon avec les listes (en paramètre) :

```
user=> (defn longueur [l]
  (if (empty? l) 0
      (+ 1 (longueur (rest l)))))
```

```
#'user/longueur
user=> (longueur l1)
3
```

Ou en valeur de retour :

```
user=> (defn interval [a b]
  (if (= a b) '()
      (conj (interval (+ a 1) b) a)))
#'user/interval
```

```
user=> (interval 3 11)
(3 4 5 6 7 8 9 10)
```