

# CAP - Clojure: Introducció



Jordi Delgado, Gerard Escudero,

Tema 1



# Clojure: Introducció

## Expressions, avaluació, valors

Un programa funcional (no trivial) està format d'*expressions*, que s'*avaluen*, i retornen un *valor*.

Si aquest programa està escrit en un llenguatge funcional *pur* (p.ex. Haskell), això és tot. En canvi si el llenguatge de programació no és funcional pur (p.ex. Clojure), aquestes expressions poden tenir *side effects*.

Hi ha expressions que es poden considerar *constants*, és a dir, valors l'avaluació dels quals és precisament ells mateixos:

```
$ clj
Clojure 1.11.4
user=> "hola món"
"hola món"
user=> 235678587432035784084032475857N
235678587432035784084032475857N
user=> 3.141592653589793
3.141592653589793
user=> 1e-23
1.0E-23
```

```
user=> 1567843
1567843
user=> \t
\t
user=> :clau
:clau
user=> true
true
user=> 5/2
5/2
```

# Clojure: Introducció

## Expressions, avaluació, valors

En general, però, una *expressió* en Clojure o bé és un *símbol*, o bé té la forma d'una *llista*\* (`f e1 e2 ... eN`), on:

- Si és un *símbol*, s'avalua i es retorna el valor al que està lligat (*bound*). Error si no està lligat a cap valor.
- Si és una *llista*, l'expressió `f` és una *funció*, un *símbol* que s'avalui en una funció, una *special form* o el nom d'una *macro*.
- Si l'expressió `f` és una *funció* o un *símbol* que s'avalua en una funció, el resultat d'aquesta avaluació és un *valor-funció*, que anomenarem `fv`. Les expressions `e1 e2 ... eN` s'avaluen i resulten en valors `v1 v2 ... vN`. En aquest cas, el resultat de l'expressió és l'aplicació del valor-funció sobre els valors que es passen com a arguments: `fv(v1 v2 ... vN)`.
- Si l'expressió `f` és una *special form* o el nom d'una *macro*, els arguments no tenen per què avaluar-se i cal veure com es defineix l'*special form* o la macro en qüestió.

\* Veurem les llistes de seguida, de moment penseu-hi com un contenidor d'elements entre parèntesis

# Clojure: Introducció

## Expressions, avaluació, valors

- Si l'expressió  $f$  és una *funció* o un *símbol* que s'avalua en una funció, el resultat d'aquesta avaluació és un *valor-funció*, que anomenarem  $fv$ . Les expressions  $e_1 \ e_2 \ \dots \ e_N$  s'avaluen i resulten en valors  $v_1 \ v_2 \ \dots \ v_N$ . En aquest cas, el resultat de l'expressió és l'aplicació del valor-funció sobre els valors que es passen com a arguments:  $fv(v_1 \ v_2 \ \dots \ v_N)$ .

Per exemple,

```
$ clj
Clojure 1.11.4
user=> (+ 10 20 30 40) ;; el símbol '+' s'avalua en el valor-funció de la suma
100
user=> (+ 1)
1
user=> (+)
0
user=> (max (+ 3 4) (* (- 4 2) (+ 8 10)) 10) ;; altres expressions com a arguments
36
user=>
```

# Clojure: Introducció

## Expressions, avaluació, valors

- Si l'expressió `f` és una **special form** o el nom d'una **macro**, els arguments no tenen per què avaluar-se i cal veure com es defineix l'**special form** o la macro en qüestió.

Exemple: La **special form** `def`.

Per lligar (*bind*) valors a noms (símbols) fem servir la **special form** `def`.

`(def s (max (+ 23 10) (- 100 85)))` lliga el símbol `s` al valor `33`. Podem veure que `def` evalua el seu segon argument, però NO evalua el primer argument, que ha de ser un símbol.

Està pensat per modificar símbols *globals*, i usualment no modificarem un lligam creat amb `def`.

Al símbol creat amb `def` se'l coneix com a **var**, i *no és el mateix que una variable* (tal i com les entenem en altres llenguatges de programació).

# Clojure: Introducció

## Expressions, avaluació, valors

A Clojure tenim **Nombres** (enters, reals, també tenim fraccions), **Caràcters** (notació \a), **Booleans, keywords** (notació :s), **símbols** i **strings** (que també són col·leccions).

Operacions *habituals* on la seva funcionalitat és (més o menys) òbvia: +, -, \*, /, quot, rem, mod, inc, dec, =, not=, <, <=, >, >=, zero?, pos?, neg?, number?, min, max, not, and, or, print, println, char, keyword, keyword?, str, subs, string?

**Exercici:** Obriu un *repl* de Clojure i feu-les servir. Exploreu.

```
$ clj
Clojure 1.11.4
user=> (- 10 9 8)
-7
user=> (/ 100 10 2)
5
user=> (number? "no")
false
user=>
```

```
$ clj
Clojure 1.11.4
user=> (def v 7)
#'user/v
user=> (inc v)
8
user=> v
7
user=>
```

# Clojure: Introducció

## Llistes (I)

Hem vist que les expressions són llistes, però les llistes són un dels contenidors de dades que podem fer servir a Clojure. La *Llista* no és més que una col·lecció d'elements en un ordre determinat, delimitada entre parèntesis:

(1 2 3 4) -- Llista amb 1, 2, 3, 4

(\a "pep" 2 3.141592) -- Elements de diferents tipus

() -- Llista buida

(def tt 2.781) -- Una expressió és una llista

(max 34 -23 1) -- Una expressió és una llista

**Homoiconicitat:** Un programa Clojure és una expressió a avaluar i una llista, per tant és també un valor, o una dada, de Clojure. *Code is Data*

# Clojure: Introducció

## Llistes (II)

Aleshores, com distingeix Clojure entre una llista que només conté dades i una expressió?

```
(def x (\a "pep" 2 3.141592))  Execution error  
(max 34 -23 1)  34
```

Quan Clojure troba una llista (o, ja que hi som, un símbol) **sempre l'avalua com si fos una expressió**. Si volem que una llista (o un símbol) no s'avalui com una expressió cal dir-ho explícitament amb `quote`:

```
(def x (quote (\a "pep" 2 3.141592)))  
x  (\a "pep" 2 3.141592)
```

Com que `quote` es fa servir molt sovint, tenim una manera d'abreujar-ho:

```
(def x '(\a "pep" 2 3.141592))  
x  (\a "pep" 2 3.141592)
```

El caràcter `'` fa el mateix paper que la *special form* `quote`

# Clojure: Introducció

## Llistes (III): Operacions sobre llistes:

Suposem que hem fet `(def lst '(\a "c" 20 :k))`

`first - (first lst)` ➡ \a

`cons - (cons 'a lst)` ➡  
(a \a "c" 20 :k)

`rest - (rest lst)` ➡ ("c" 20 :k)

`next - (next lst)` ➡ ("c" 20 :k)

però

`rest - (rest '())` ➡ ()

`next - (next '())` ➡ nil

`conj - (conj lst 'a)` ➡

(a \a "c" 20 :k)

`peek - (peek lst)` ➡ \a

`pop - (pop lst)` ➡ ("c" 20 :k)

`list - (list \a "c" 20 :k)` ➡  
(\a "c" 20 :k)

`list? - (list? lst)` ➡ true

`count - (count lst)` ➡ 4

`empty? - (empty? lst)` ➡ false

# Clojure: Introducció

## Estructures de Control Condicionals

Important!: Les estructures de control també són expressions

En el cas de les estructures de control condicionals, tenim:

- `(if bool-expr expr-true expr-false)` -- `if` és una *special form* (també existeix la macro `if-not`).
- `(when bool-expr expr)` -- `when` i `when-not` són macros definides en termes d'`if`.
- Les macros `when-let` i `if-let` (en parlarem més endavant).
- L'expressió condicional més general és la macro `cond`:

```
(cond
  bool-expr expr
  bool-expr expr
  ...
  bool-expr expr)
```

Important!: Només `false` i `nil` són *falsy*. La resta de valors són *truthy*.

# Clojure: Introducció

## Funcions: Funcions anònimes, l'*special form fn*

Fem servir l'*special form fn* per definir funcions anònimes:

```
;;    paràmetres    cos de la funció  
;;    -----  
(fn [p1 p2 ... pN] expr1 ... exprM)
```

Poden haver \$M\$ expressions, però **el valor de retorn de la funció és la darrera expressió avaluada** (no hi ha `return`).

Per exemple:

```
(def valor_absolut (fn [x] ((if (> x 0) + -) x)))  
(valor_absolut 234)  ↗ 234  
(valor_absolut -234) ↗ 234  
  
(def que-fer (fn [temp] (cond  
                           (> temp 30) "Em quedo a casa amb l'aire acondicionat"  
                           (> temp 15) "Me'n vaig a fer un café"  
                           :else "Em quedo al llit")))  
(que-fer 31) ↗ "Em quedo a casa amb l'aire acondicionat"  
(que-fer 25) ↗ "Me'n vaig a fer un café"  
(que-fer 10) ↗ "Em quedo al llit"
```

# Clojure: Introducció

## Funcions: Funcions anònimes, notació abreujada #( . . . )

Podem escriure les funcions anònimes amb una notació molt més còmode d'utilitzar. De fet, ho farem servir sovint:

```
#(cos de la funció)
```

I els paràmetres? Dins el cos de la funció podem fer referència als arguments que passem a la crida a la funció amb la notació: %1, %2, etc. Si només hi ha un paràmetre podem fer servir %

Per exemple:

```
(def valor_absolut #((if (> % 0) + -) %) )
(valor_absolut 234)  ↗ 234
(valor_absolut -234) ↗ 234

(def que-fer #(cond (> % 30) "Em quedo a casa amb l'aire acondicionat"
                     (> % 15) "Me'n vaig a fer un café"
                     :else "Em quedo al llit") )
(que-fer 31)  ↗ "Em quedo a casa amb l'aire acondicionat"
(que-fer 25)  ↗ "Me'n vaig a fer un café"
(que-fer 10)  ↗ "Em quedo al llit"
```

# Clojure: Introducció

## Funcions: La macro defn (I)

Existeix una macro, `defn` amb la que podem definir funcions de manera més compacta:

;;	paràmetres	cos de la funció
;;	-----	-----
( <b>defn</b> nom-de-funció "Comentari textual" [p1 p2 ... pN]		expr1 ... exprM )

Aquesta és la manera en que habitualment definirem funcions. Per exemple:

```
(defn valor_absolut "Calcula |x|" [x] ((if (> x 0) + -) x))

(valor_absolut 234) ➡ 234
(valor_absolut -234) ➡ 234

(defn que-fer [temp] (cond
                      (> temp 30) "Em quedo a casa amb l'aire acondicionat"
                      (> temp 15) "Me'n vaig a fer un café"
                      :else "Em quedo al llit"))

(que-fer 31) ➡ "Em quedo a casa amb l'aire acondicionat"
(que-fer 25) ➡ "Me'n vaig a fer un café"
(que-fer 10) ➡ "Em quedo al llit"
```

# Clojure: Introducció

## Funcions: La macro `defn` (II)

Amb `defn` també podem definir funcions d'*aritats múltiples*, és a dir, funcions amb diferent nombre de paràmetres.

Per exemple:

```
(defn producte
  "Retorna 1, el paràmetre o el producte dependent del nombre de paràmetres"
  ([] 1)
  ([x] x)
  ([x y] (* x y)))

(producte)           ↗ 1                                ;; cap argument
(producte 129)       ↗ 129                             ;; un argument
(producte 23 34)      ↗ 782                             ;; dos arguments
(producte 231 134 23) ↗ Execution error (ArityException) ;; no més
```

# Clojure: Introducció

## Funcions: Els paràmetres (I)

Els paràmetres formals de les funcions (excepte en les funcions anònimes en notació abreujada) s'especifiquen amb un *vector\** de símbols. Clojure permet *definir funcions amb un nombre variable de paràmetres*.

La manera de fer-ho és fent servir el símbol especial &.

Suposem que definim una funció amb paràmetres formals [ $p_1 \dots p_N$  &  $p$ ]. Aquesta funció requereix que la crida es faci amb un mínim de  $$N$$  arguments, però tots els arguments a partir de  $'$N+1$'$  apareixeran dins una seqüència\* lligada al símbol  $p$ .

El símbol que segueix a & "recull" en una seqüència tots els arguments que es passin a la funció (més enllà dels obligatoris) i la lliga a aquest símbol (dins de la funció).

\* Veurem els vectors i les seqüències ben aviat. Ara penseu les seqüències com una mena de llista.

# Clojure: Introducció

## Funcions: Els paràmetres (II)

Per exemple:

```
(defn producte
  "Retorna 1, el paràmetre o el producte dependent del nombre de paràmetres"
  ([] 1)
  ([x] x)
  ([x y & z] (apply * x y z))) ;; z - seqüència amb el 3r, 4t, etc. arguments

(producte)           ↗ 1                                     ;; cap argument
(producte 129)        ↗ 129                                ;; un argument
(producte 23 34)       ↗ 782                                ;; dos arguments
(producte 231 134 23) ↗ 711942                            ;; tres arguments
```

on `(apply f e1 e2 '(v3 ... vN)) = (f e1 e2 v3 ... vN)`

(e són expressions, v són valors)

```
(apply + 1 2 3 '(4 5 6 7))      ↗ 28
(apply * (- 4 3) 2 3 '())        ↗ 6
(apply producte 1 2 3 '(4 5 6 7)) ↗ 5040
(apply max 10 20 30)              ↗ Execution error (IllegalArgumentException)
(apply max 10 20 30 '())          ↗ 30
```

# Clojure: Introducció

## La recursivitat (*naïve*)

Òbviament les funcions en Clojure poden ser recursives. Una funció pot fer servir el seu nom per invocar-se ella mateixa. De moment farem servir la recursivitat d'aquesta manera. Més endavant hi tornarem...

## La *special form* do

La *special form* do serveix per avaluar expressions una darrera l'altra, seqüencialment: (do expr1 expr2 ... exprN). Aquesta expressió retorna el valor resultant de l'avaluació de la darrera expressió.

I què passa amb els valors retornats per l'avaluació de les altres expressions? Es perden. Usualment, es fan servir pels seus efectes colaterals (*side effects*).

El cos d'una funció té una estructura similar. Direm que *el cos d'una funció és dins d'un do implícit*.

# Clojure: Introducció

## La *special form* do

```
(do
  (println "Efecte colateral: escrivim un missatge") ;; S'escriu el missatge
  (* 5 4 3 2 1)                                     ;; Aquest valor, 120, es perd
  (quot 343 5))                                    ;; Aquest valor és el retorn del do
```



- Efecte colateral: escrivim un missatge

Les funcions tenen un **do** implícit:

```
(defn foo [x y & z]
  (println "Els dos primers arguments són:",x,y)
  (println "La resta d'arguments:",z)
  :ok)      ;; la funció retorna un keyword
```

```
(foo 1 2)  :ok
```

- Els dos primers arguments són: 1 2
- La resta d'arguments: nil

```
(foo 1 2 3 4 5 6)  :ok
```

- Els dos primers arguments són: 1 2
- La resta d'arguments: (3 4 5 6)

# Clojure: Introducció

## Lligams locals: La *special form* let (I)

Sovint voldrem lligar localment valors a símbols dins una expressió. Això ho farem amb el `let`:

;;	binding-forms	cos del let
;;	-----	-----
( <b>let</b> [s1 ex1 s2 ex2 ... sN exN]		expr1 ... exprM )

Les ***binding forms*** que farem servir de moment són símbols `s1, s2, ..., sN`. Més endavant veurem la seva forma general.

S'avaluen, en l'ordre que apareixen, les expressions i es lliguen als símbols corresponents: S'avalua `ex1` i es lliga el resultat al símbol `s1`, després s'avalua `ex2` i es lliga el resultat al símbol `s2`, etc. A una expressió `exj` podem fer servir qualsevol símbol `sk` (amb  $k < j$ ) que s'hagi lligat abans.

El cos del `let` té un `do` implícit, per tant es retorna la darrera expressió avaluada.

Els lligams locals només són visibles dins el cos del `let` (*lexical scope*) i **no es poden modificar** dins el cos del `let`. Un cop més, no estem parlant de variables.

# Clojure: Introducció

## Lligams locals: La *special form* let (II)

```
(defn segons-a-setmanes
  "Converteix un cert nombre de segons a setmanes"
  [segons]
  (let [minuts (/ segons 60)
        hores   (/ minuts 60)
        dies    (/ hores 24)
        setmanes (/ dies 7) ]
    setmanes))

(segons-a-setmanes 0)           ↗ 0
(segons-a-setmanes 604800)      ↗ 1
(segons-a-setmanes (* 4 604800)) ↗ 4
(segons-a-setmanes 60483)       ↗ 20161/201600
```

```
(defn foo
  "Forma molt tonta de multiplicar per 3 un nombre"
  [n]
  (let [x n
        y x
        z y]
    (+ x y z)))

(foo 9) ↗ 27
```

# Clojure: Introducció

## Exemple: Aproximació de l'arrel quadrada

```
(def valor-absolut #((if (> % 0) + -) %))

(def mitjana #(/ (+ %1 %2) 2))

(defn prou-bo?
  "Retorna si l'aproximació és prou bona"
  [x aprox]
  (let [diff (- (* aprox aprox) x)]
    (< (valor-absolut diff) 0.001)))

(defn arrel
  "Retorna l'arrel quadrada aproximada d'un nombre positiu"
  ([x] (arrel x 1.0))
  ([x aprox]
   (if (prou-bo? x aprox)
       aprox
       (arrel x (mitjana aprox (/ x aprox))))))

(arrel 25)  ↗  5.000023178253949
(arrel 36)  ↗  6.000000005333189
(arrel 100) ↗  10.000000000139897
```

# Clojure: Introducció

## Lligams locals: La *special form letfn*

És similar al `let`, però permet definir *funcions locals*. Dins les funcions definides amb el `letfn`, qualsevol funció pot referenciar qualsevol altre.

```
(defn arrel
  "Retorna l'arrel quadrada aproximada d'un nombre positiu"
  ([x] (arrel x 1.0))
  ([x aprox]
   (letfn [(prou-bo? [x aprox]
             (let [diff (- (* aprox aprox) x)]
               (< (valor-absolut diff) 0.001)))
           (valor-absolut [x] ((if (> x 0) + -) x))
           (mitjana [x y] (/ (+ x y) 2))]
     (if (prou-bo? x aprox)
         aprox
         (arrel x (mitjana aprox (/ x aprox)))))))
  (arrel 25)    ↗  5.000023178253949
  (arrel 36)    ↗  6.000000005333189
  (arrel 100)   ↗  10.000000000139897
```

**Exercici:** Per quina raó cal el `letfn`? No en tenim prou amb el `let` per definir funcions locals? O, dit d'una altra manera, què puc fer amb el `letfn` que no puc fer amb el `let`? Investigueu.

# Clojure: Introducció

## Buckles: Las *special forms* `loop/recur` (I)

`loop` és similar a un `let`, establint al principi el lligam entre símbols i els seus *valors inicials*. `loop` estableix un *punt de retorn*. Tot seguit trobem un `do` implícit dins el que podem fer servir `recur`. La *special form* `recur` fa dues coses: Una és donar nous valors als símbols definits amb `loop`, i una altra és transferir el control al punt de retorn definit per `loop`.

```
(loop [result '(), x 5]
  (if (zero? x)
    result
    (recur (conj result x) (dec x))))
```

👉 (1 2 3 4 5)

```
(defn arrel-loop
  "Retorna l'arrel quadrada aproximada d'un nombre positiu"
  [x]
  (loop [aprox 1.0]
    (if (prou-bo? x aprox) ;; prou-bo? i mitjana ja definides
        aprox
        (recur (mitjana aprox (/ x aprox))))))
```

(arrel-loop 36) 👈 6.00000005333189

# Clojure: Introducció

## Buckles: Las *special forms* `loop/recur` (II)

Un factorial iteratiu:

```
(defn factorial
  "calcula el factorial d'un nombre enter positiu o zero"
  [n]
  (loop [i n, r 1]  ;; el valor inicial d'i és n i el d'r és 1
    (if (<= i 1)
        r
        (recur (dec i) (* r i)))))

(factorial 0) ➔ 1
(factorial 1) ➔ 1
(factorial 4) ➔ 24
(factorial 5) ➔ 120
(factorial 6) ➔ 720
(factorial 1000) ➔ Execution error (ArithmetricException)
(factorial 1000N) ➔ 4023872600...00000N (2568 díigits!)
```

Tornarem a trobar `recur` més endavant en un context més general i entendrem el per què d'aquesta manera tan *estranya* de definir els bucles.

# Clojure: Introducció

Definim la funció *subfactorial* d'un nombre enter no negatiu:

$\text{subfactorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } n = 1 \\ (n-1) * (\text{subfactorial}(n-1) + \text{subfactorial}(n-2)) & \text{otherwise} \end{cases}$

Solució recursiva (múltiple, dues crides), ineficient<sup>1</sup>:

```
(defn subfact_recursiva [n]
  (cond
    (or (= n 0) (= n 2)) 1
    (= n 1) 0
    :else (*' (dec n) (+' (subfact_recursiva (dec n)) (subfact_recursiva (- n 2)
(subfact_recursiva 0) 1
(subfact_recursiva 10) 1334961
(subfact_recursiva 23) 9510425471055777937262N
(subfact_recursiva 10000) Execution error (StackOverflowError)
```

<sup>1</sup> Les operacions `+` i `*` són per treballar amb nombres enters molt grans. Les operacions `+*` generen *overflow*.

# Clojure: Introducció

Provem de fer-ne una versió recursiva final:

```
(defn subfact_final
  ([n] (cond
          ((or (= n 0) (= n 2))) 1
          (= n 1) 0
          :else (subfact_final n 2 0 1)))
  ([n k nm1 nm2] (if (> k n)
                        nm1
                        (subfact_final n (inc k) (*' (dec k) (+' nm1 nm2)) nm1)))))

(subfact_final 2) ➡ 1
(subfact_final 10) ➡ 1334961
(subfact_final 23) ➡ 9510425471055777937262N
(subfact_final 100000) ➡ Execution error (StackOverflowError)
```

No sembla que les coses hagin millorat gaire, però podrien haver-ho fet!.

Quan hi ha una crida a funció en *tail position* no cal crear cap estructura addicional (*stack frame*) per a aquella crida, i així no es consumeix espai. D'això se'n diu *tail call optimization* (TCO). Una funció recursiva final té la crida recursiva en *tail position* (és essencialment una *iteració*), i hi ha llenguatges de programació que optimitzen aquest fet amb TCO.

Clojure no té TCO en general, per raons que tenen a veure amb l'arquitectura de la JVM.

# Clojure: Introducció

## La *special form* `recur`

Podem dir-li a Clojure explícitament que una crida recursiva està en *tail position* (una funció és recursiva final si totes les crides recursives estan en *tail position*) fent servir `recur`:

```
(defn subfact_recur
  ([n] (cond
          ((or (= n 0) (= n 2))) 1
          (= n 1) 0
          :else (subfact_recur n 2 0 1)))
  ([n k nm1 nm2] (if (> k n)
                       nm1
                       (recur n (inc k) (*' (dec k) (+' nm1 nm2)) nm1)))))

(subfact_recur 2) ➡ 1
(subfact_recur 10) ➡ 1334961
(subfact_recur 23) ➡ 9510425471055777937262N
(subfact_recur 10000) ➡ 10470804208445737513419...39696860001N (35660 díigits!)
```

Ara evitem l'`StackOverflowError`, ja que l'avaluació de `subfact_recur` no consumeix espai addicional. `recur` serveix per dir-li a Clojure que faci TCO.

# Clojure: Introducció

## Tail Position

Ara bé, `recur` només es pot fer servir en tail position i fent referència a una crida recursiva. `recur` no serveix per fer TCO en general:

Table 7.1 Tail positions and recur targets

Form(s)	Tail position	Recur target?
fn, defn	(fn [args] expressions tail)	Yes
loop	(loop [bindings] expressions tail)	Yes
let, letfn, binding	(let [bindings] expressions tail)	No
do	(do expressions tail)	No
if, if-not	(if test then-tailelse-tail)	No
when, when-not	(when test expressions tail)	No
cond	(cond test test tail ...:else else tail)	No
or, and	(or test test... tail)	No
case	(case const const tail ... default tail)	No

Font: *The Joy of Clojure*, p. 160

# Clojure: Introducció

## La *special form* `recur`

Finalment, podem fer una versió *iterativa*:

```
(defn subfact_iter
  [n]
  (cond
    (or (= n 0) (= n 2)) 1
    (= n 1) 0
    :else (loop [k 2, nm1 0, nm2 1]
             (if (> k n)
                 nm1
                 (recur (inc k) (*' (dec k) (+' nm1 nm2)) nm1)))))

(subfact_iter 2) ➡ 1
(subfact_iter 10) ➡ 1334961
(subfact_iter 23) ➡ 9510425471055777937262N
(subfact_iter 10000) ➡ 10470804208445737513419...39696860001N (35660 dígits!)
```

Ara podem entendre millor el sentit de `recur`. En realitat el que fa és considerar un bucle com una mena de funció recursiva final *implícita*. O bé, també podríem entendre que els paràmetres d'una funció defineixen un *loop implícit*, fent d'una funció recursiva final un bucle. Sigui com sigui, l'ús de `recur` és *equivalent* en tots dos casos.

**Exercici:** Compareu la versió iterativa amb la recursiva final.

# Clojure: Introducció

## Les col·leccions: Vectors

Són col·leccions d'elements en un ordre determinat, similars a les llistes, però són **molt més eficients**. Els preferirem a les llistes, ja que l'accés a un element via un **índex** té un cost quasi-constant, i no són expressions, per tant no cal fer servir **quote**.

Un vector literal té la notació [1 2 \t "hi" :k], és a dir, elements entre claudàtors. El primer element té índex 0.

```
[1 2 (+ 1 2)] ➡ [1 2 3]

(def nums (vec '(1 2 3 4 5))) ➡ #'user/nums ;; convertim llista en vector
nums ➡ [1 2 3 4 5]
(get nums 3) ➡ 4
(nums 3) ➡ 4
(nth nums 3) ➡ 4

(nth nums 10 :no-hi-soc) ➡ :no-hi-soc ;; nth - argument per si l'índex no hi és
(get nums 10 :no-hi-soc) ➡ :no-hi-soc ;; get - argument per si l'índex no hi és

(conj nums 6 7) ➡ [1 2 3 4 5 6 7]
(conj '(1 2 3 4 5) 6 7) ➡ (7 6 1 2 3 4 5) ;; ep!!
```

# Clojure: Introducció

## Les col·leccions: Accés als elements dels vectors

**Table 5.2 Vector lookup options: the three ways to look up an item in a vector, and how each responds to different exceptional circumstances**

	<b>nth</b>	<b>get</b>	<b>Vector as a function</b>
If the vector is nil	Returns nil	Returns nil	Throws an exception
If the index is out of range	Throws exception by default or returns a “not found” if supplied	Returns nil	Throws an exception
Supports a “not found” arg	Yes (nth [] 9 :whoops)	Yes (get [] 9 :whoops)	No

Font: *The Joy of Clojure*, p. 93

# Clojure: Introducció

## Les col·leccions: Operacions sobre Vectors

Suposem que hem fet (def v [\a "c" 20 :k])

first - (first v) ➡ \a

cons - (cons 'a v) ➡  
(a \a "c" 20 :k) (!)

rest - (rest v) ➡ ("c" 20 :k) (!)

next - (next v) ➡ ("c" 20 :k) (!)

però

rest - (rest []) ➡ () (!)

next - (next []) ➡ nil

conj - (conj v 'a) ➡  
[\a "c" 20 :k a]

peek - (peek v) ➡ :k

pop - (pop v) ➡ [\a "c" 20]

vector - (vector \a "c" 20 :k) ➡  
[\a "c" 20 :k]

vector? - (vector? v) ➡ true

count - (count v) ➡ 4

empty? - (empty? v) ➡ false

subvec - (subvec v 2 4) ➡ [20 :k]

assoc - (assoc v 1 -100) ➡  
[\a -100 20 :k]

# Clojure: Introducció

## Les col·leccions: Misteri (I)

Hem vist que hi ha operacions que *aparentment* retornen una llista, ja operin sobre llistes o vectors:

```
(def lst '(:a :b :c))      ↗ #'user/lst
(def vct  [:a :b :c])      ↗ #'user/vct

(list? lst)                ↗ true
(vector? vct)              ↗ true

(rest lst)                 ↗ (:b :c)
(rest vct)                 ↗ (:b :c)

(= (rest lst) (rest vct))  ↗ true
```

Fins aquí tot s'entén. Però...

```
(list? (rest lst))        ↗ true
(list? (rest vct))        ↗ false
(vector? (rest vct))      ↗ false
```

Si `(rest lst)` és una llista i `(rest vct)` no ho és, com poden ser `=?`

# Clojure: Introducció

## Les col·leccions: Maps (diccionaris) (I)

Un diccionari, *map* és com els anomena Clojure, és una col·lecció de parelles clau-valor entre *tirants (curly braces)*:

```
(def my-array-map (array-map :dos 2 :tres 3 :un 1)) ;; array-map, ordre d'inscripció
(def my-hash-map {:un 1 :tres 3 :dos 2})   ;; també (hash-map :un 1 :tres 3 :dos 2)
(def my-sorted-map (sorted-map :un 1 :dos 2 :tres 3)) ;; sorted-map segons claus

(my-array-map :dos)      ↗ 2
(get my-array-map :dos) ↗ 2
(:dos my-array-map)    ↗ 2 ;; només si les claus són keywords
(my-hash-map :tres)     ↗ 3
(get my-hash-map :tres) ↗ 3
(:tres my-hash-map)    ↗ 3 ;; només si les claus són keywords
(my-sorted-map :un)     ↗ 1
(get my-sorted-map :un) ↗ 1
(:un my-sorted-map)    ↗ 1 ;; només si les claus són keywords
(my-sorted-map :zero)   ↗ nil
(get my-hash-map :zero) ↗ nil
```

Les claus i els valors poden ser qualsevol objecte de Clojure, però és habitual fer servir *keywords* per a les claus, tal i com hem fet a l'exemple.

# Clojure: Introducció

## Les col·leccions: Maps (diccionaris) (II)

Veiem algunes de les operacions més bàsiques:

```
(assoc my-hash-map :quatre 4)    ↗ {:un 1, :tres 3, :dos 2, :quatre 4}
(assoc my-sorted-map :quatre 4)  ↗ {:dos 2, :quatre 4, :tres 3, :un 1}
;; també
(assoc my-hash-map "a" \a "b" \b)  ↗ {:un 1, :tres 3, :dos 2, "a" \a, "b" \b}
(assoc my-sorted-map "a" \a "b" \b)  ↗ Execution error (ClassCastException)

(dissoc my-array-map :un :tres)  ↗ {:dos 2}
(dissoc my-array-map :zero)      ↗ {:dos 2, :tres 3, :un 1}

(conj my-hash-map {:quatre 4})  ↗ {:un 1, :tres 3, :dos 2, :quatre 4}
(conj my-hash-map [:quatre 4])   ↗ {:un 1, :tres 3, :dos 2, :quatre 4}

(keys my-hash-map)  ↗ (:un :tres :dos) ;; què retorna keys?
(vals my-array-map) ↗ (2 3 1)          ;; què retorna vals?

contains? my-sorted-map :un)    ↗ true
contains? my-sorted-map :zero)   ↗ false

(empty? my-hash-map)  ↗ false
(empty? {})           ↗ true
```

# Clojure: Introducció

## Les col·leccions: Maps (diccionaris) (III)

Veiem exemples més enrevessats:

```
(let [m {:a 1, 1 :b, [1 2 3] "4 5 6"}]  
  [(m :a) (m [1 2 3])])  
👉 [1 "4 5 6"]  
  
(into (sorted-map) [ [:a 1] [:c 3] [:b 2] ] ) 👉 {:a 1, :b 2, :c 3}  
(into (hash-map) [ [:a 1] [:c 3] [:b 2] ] ) 👉 {:a 1, :c 3, :b 2}  
  
(zipmap [:a :b :c :d :e] [1 2 3 4 5]) 👉 {:a 1, :b 2, :c 3, :d 4, :e 5}  
  
;; Les claus són nombres i els valors keywords! 🤔, cap problema  
(assoc {1 :int} 1.0 :float) 👉 {1 :int, 1.0 :float}  
(assoc (sorted-map 1 :int) 1.0 :float) 👉 {1 :float} ;; (= 1 1.0) 👉 true  
  
(assoc (array-map :a 1 :c 3 :b 2) :e 5 :d 4) 👉 {:a 1, :c 3, :b 2, :e 5, :d 4}  
(assoc (sorted-map :a 1 :c 3 :b 2) :e 5 :d 4) 👉 {:a 1, :b 2, :c 3, :d 4, :e 5}
```

**Exercici:** Mirar **into** i **zipmap** amb una mica de detall.

# Clojure: Introducció

## Les col·leccions: Sets (conjunts) (I)

Els conjunts són col·leccions d'elements únics (no poden haver elements repetits, és a dir, que siguin `=`). Es corresponen força bé a la intuició que ja teniu del concepte de conjunt en matemàtiques. Els conjunts es defineixen literalment amb `#{ ... }` o amb `(sorted-set ...)` (i en aquest cas els seus elements han de ser comparables).

<code>(def s #{\a 3 "foo"})</code>	👉 #'user/s
<code>s</code>	👉 #{"foo" \a 3}
<code>(def ss (sorted-set \a 3 "foo"))</code>	👉 Execution error (ClassCastException)
<code>(def ss (sorted-set \k \l \a \d \v))</code>	👉 #'user/ss
<code>ss</code>	👉 #{\a \d \k \l \v}
<code>#{:a :b :c :b}</code>	👉 (...) Duplicate key: :b
<code>(sorted-set :a :b :c :b)</code>	👉 #{:a :b :c}
<code>(conj s :nou)</code>	👉 #{"foo" \a :nou 3}
<code>(conj ss \y)</code>	👉 #{\a \d \k \l \v \y}
<code>(disj s "foo")</code>	👉 #{\a 3}
<code>(disj ss \k)</code>	👉 #{\a \d \l \v}
<code>(disj ss \m)</code>	👉 #{\a \d \k \l \v}

# Clojure: Introducció

## Les col·leccions: Sets (conjunts) (II)

Les operacions matemàtiques típiques dels conjunts requereixen un afegit:

```
(require 'clojure.set)

(clojure.set/intersection #{:a :b :c} #{:c :d :e})  ↗ #{:c}
(clojure.set/union #{:a :b :c} #{:c :d :e})        ↗ #{:e :c :b :d :a}
(clojure.set/difference #{:a :b :c} #{:c :d :e})   ↗ #{:b :a}

(def s1 (sorted-set :a :b :c))  ↗ #'user/s1
(def s2 (sorted-set :c :e :d))  ↗ #'user/s2

(clojure.set/intersection s1 s2) ↗ #{:c}
(clojure.set/union s1 s2)      ↗ #{:a :b :c :d :e}
(clojure.set/difference s1 s2) ↗ #{:a :b}
```

# Clojure: Introducció

## Les col·leccions: Misteri (II)

```
(def lst '(:a 1 :b 2 :c 3 :d 4))
(def vct [:a 1 :b 2 :c 3 :d 4])
(def dic {:a 1 :b 2 :c 3 :d 4})
(def con #{:a 1 :b 2 :c 3 :d 4})
```

lst ➡ (:a 1 :b 2 :c 3 :d 4)  
vct ➡ [:a 1 :b 2 :c 3 :d 4]  
dic ➡ {:a 1, :b 2, :c 3, :d 4}  
con ➡ #{1 4 :c 3 2 :b :d :a}

```
(first lst) ➡ :a
(first vct) ➡ :a
(first dic) ➡ [:a 1]
(first con) ➡ 1
```

```
(rest lst) ➡ (1 :b 2 :c 3 :d 4)
(rest vct) ➡ (1 :b 2 :c 3 :d 4)
(rest dic) ➡ ([:b 2] [:c 3] [:d 4])
(rest con) ➡ (4 :c 3 2 :b :d :a)
```

```
(list? (rest lst)) ➡ true
(vector? (rest vct)) ➡ false ;; però (list? (rest vct)) ➡ false
(map? (rest dic)) ➡ false ;; però (list? (rest dic)) ➡ false
(set? (rest con)) ➡ false ;; però (list? (rest con)) ➡ false
```

Solució: La propera plana 😊

# Clojure: Introducció

## Les Seqüències (I):

Clojure posa a disposició del programador una *abstracció* que permet tractar de manera similar diferents col·leccions: **La Seqüència**.

Totes les col·leccions de Clojure que hem vist (llistes, vectors, diccionaris, conjunts, *strings*) són el que s'anomena **seqable**, és a dir, podem utilitzar amb aquestes estructures les funcions que defineixen l'API de les seqüències.

Resolguem els misteris:

```
(list? (rest lst))  ↗ true
(vector? (rest vct)) ↗ false  ;; però (list? (rest vct))  ↗ false
(map? (rest dic))    ↗ false  ;; però (list? (rest dic))  ↗ false
(set? (rest con))   ↗ false  ;; però (list? (rest con))  ↗ false
```

;; En realitat, rest, next, cons retornen seqüències

```
(seq? (rest lst))  ↗ true
(seq? (rest vct))  ↗ true
(seq? (rest dic))  ↗ true
(seq? (rest con))  ↗ true
```

# Clojure: Introducció

## Les Seqüències (II):

Les operacions principals sobre seqüències són:

`first`: Obtenir el primer element d'una seqüència

`rest`: Obtenir una seqüència nova amb els mateixos elements que la seqüència original, menys el primer element. Retorna () (seqüència buida) si no hi ha més elements

`cons`: Obtenir una seqüència nova amb un element afegit al davant dels elements de la seqüència original.

```
;;; continuant amb l'exemple dels misteris...
(cons :u lst) ➡ (:u :a 1 :b 2 :c 3 :d 4)
(cons :u vct) ➡ (:u :a 1 :b 2 :c 3 :d 4)
(cons :u dic) ➡ (:u [:a 1] [:b 2] [:c 3] [:d 4])
(cons :u con) ➡ (:u 1 4 :c 3 2 :b :d :a)
```

`seq`: retorna la seqüència corresponent a la col·lecció que rep com a argument.

`next`: el mateix que `rest`, excepte que si no hi ha més elements retorna `nil`. Si `s` és una seqüència, (`next s`) = (`seq (rest s)`)

# Clojure: Introducció

## Les Seqüències (III):

```
;;; continuant amb l'exemple dels misteris...
```

```
(seq lst)  ↗ (:a 1 :b 2 :c 3 :d 4)
(seq vct)  ↗ (:a 1 :b 2 :c 3 :d 4)
(seq dic)  ↗ ([:a 1] [:b 2] [:c 3] [:d 4])
(seq con)  ↗ (1 4 :c 3 2 :b :d :a)
```

La *Clojure sequence library* és una col·lecció de funcions que operen sobre qualsevol seqüència. Vegem-ne alguns exemples:

```
;; (range start? end? step?)
(range 10)      ↗ (0 1 2 3 4 5 6 7 8 9)
(range 5 10)    ↗ (5 6 7 8 9)
(range 10 0 -2) ↗ (10 8 6 4 2)

;; (repeat n x)
(repeat 5 :abc) ↗ (:abc :abc :abc :abc :abc)

;; (take n sequence)
(take 3 (range 100)) ↗ (0 1 2)

;; etc. Anirem veient aquestes funcions durant el curs...
```

# Clojure: Introducció

## Les Seqüències (IV):

El que és interessant és que qualsevol estructura de dades que sigui susceptible de ser argument de `first/rest/cons` (i totes les que hem vist ho són) pot fer servir la *Clojure sequence library*.

De fet, això ens permet crear funcions molt generals, que treballin sobre qualsevol seqüència.

```
(defn nombre-elements
  "compta quants elements té la seqüència"
  [sequencia]
  (loop [c 0, s sequencia]
    (if (not (seq s))
        c
        (recur (inc c) (rest s)))))

(nombre-elements '(1 2 3 4))           ↗ 4
(nombre-elements [1 2 3 4])            ↗ 4
(nombre-elements {:a 1 :b 2 :c 3 :d 4}) ↗ 4
(nombre-elements #{:a 1 :b 2 :c 3 :d 4}) ↗ 8
```

# Clojure: Introducció

## Les Seqüències (V): Aclarint terminologia...

**Table 5.1 Sequence terms in brief**

Term	Brief description	Example(s)
Collection	A composite data type	[1 2], {:a 1}, #{1 2}, and lists and arrays
Sequential	Ordered series of values	[1 2 3 4], (1 2 3 4)
Sequence	A sequential collection that may or may not exist yet	The result of (map a-fun a-collection)
Seq	Simple API for navigating collections	first, rest, nil, and ()
clojure.core/seq	A function that returns an object implementing the seq API	(seq []) ;=> nil and (seq [1 2]) ;=> (1 2)

Font: *The Joy of Clojure*, p. 87

# Clojure: Introducció

## Les Seqüències (VI): Exemples (amb el que sabem fins ara<sup>\*</sup>!)

Fusionar (*merge*) dues seqüències ordenades:

```
(defn fusiona
  "Fusiona (merge) dues seqüències que suposem ordenades"
  [s1 s2]
  (loop [seq1 s1, seq2 s2, resultat ()]
    (let [p1 (first seq1) p2 (first seq2)]
      (cond
        (nil? p1)   (concat resultat seq2)
        (nil? p2)   (concat resultat seq1)
        (= p1 p2)   (recur (rest seq1) (rest seq2) (concat resultat [p1] [p2]))
        (< p1 p2)   (recur (rest seq1) seq2 (concat resultat [p1])))
        :else        (recur seq1 (rest seq2) (concat resultat [p2])))))))

(fusiona (range 1 20 4) (range 1 20 3)) ➡ (1 1 4 5 7 9 10 13 13 16 17 19)
(fusiona [1 3 5 7 9] [2 4 6]) ➡ (1 2 3 4 5 6 7 9)
(fusiona {:a 1 :b 2} [10 20 30]) ➡ Execution error (ClassCastException)
(fusiona (sorted-set 100 2 200 3 300) [10 20 30]) ➡ (2 3 10 20 30 100 200 300)
```

Si volguéssim fusionar *vectors* en lloc de seqüències generals, podríem fer més eficient aquesta funció fent servir `conj` en lloc de `concat`.

\* No faríem aquests exemples d'aquesta manera si sabéssim més Clojure.

# Clojure: Introducció

## Les Seqüències (VII): Exemples (amb el que sabem fins ara!)

Comprovar si una seqüència de caràcters (,),[],[] està ben balancejada.

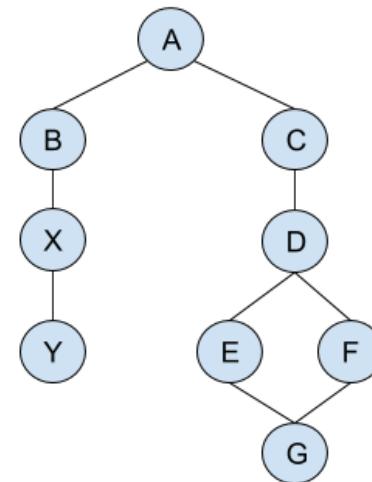
```
(defn balancejat
  [sequencia]
  (let [obrir #{ \[ \( } ]
    (loop [s sequencia, pila []]   ;; fem servir un vector com a pila
      (if (empty? s)
          (empty? pila)
          (let [c (first s)]
            (cond
              (obrir c)      (recur (rest s) (conj pila c))
              (empty? pila)  false
              :else          (let [t (peek pila)]
                            (if (or (and (= t \[) (= c \]))
                                    (and (= t \() (= c \))))
                                (recur (rest s) (pop pila))
                                false)))))))
  (balancejat "()")  ↗ true
  (balancejat "[]()" ↗ false
  (balancejat "([())[()()[]])" ↗ true
  (balancejat "[]()[((()))][[]]" ↗ false
  (balancejat '(\[ \( \) \(\ \) \])) ↗ true
```

# Clojure: Introducció

## Més Exemples (amb el que sabem fins ara!):

Suposem que representem grafs amb llistes d'adjacència de la següent manera:

```
(def G { :A [ :B :C]
         :B [ :A :X]
         :X [ :B :Y]
         :Y [ :X]
         :C [ :A :D]
         :D [ :C :E :F]
         :E [ :D :G]
         :F [ :D :G]
         :G [ :E :F]})
```



Podem fer servir com a exemples alguns algorismes que haurien de ser vells coneguts...

Font

# Clojure: Introducció

## Més Exemples (amb el que sabem fins ara!):

Recorregut en profunditat des d'un node inicial. Retorna els nodes visitats, accessibles des del node inicial, en l'ordre en que han estat visitats.

```
(defn recorregut-profunditat
  "graf és un graf representat com hem vist a la transparència anterior"
  "primer és el node del que es parteix per fer el recorregut"
  [graf primer]
  (loop [pendents [primer] ;; vector amb conj/pop/peek = Pila
         visitats #{}
         recorregut []]
    (cond
      (empty? pendents)           recorregut
      (visitats (peek pendents)) (recur (pop pendents) visitats recorregut)
      :else (let [actual (peek pendents)
                 pendents (into (pop pendents) (actual graf)) ;; Apila veïns
                 visitats (conj visitats actual)
                 recorregut (conj recorregut actual)]
              (recur pendents visitats recorregut)))))

(recorregut-profunditat G :A) ➡ [:A :C :D :F :G :E :B :X :Y]
```

Estem acostumats a veure versions recursives del recorregut en profunditat. Aquesta versió, en ser iterativa, necessita una pila auxiliar. Aquest és el paper del vector `pendents`. En ser un vector sobre el que només farem servir les operacions `conj/peek/pop/into`, es comporta com una pila on el cim és el *final* del vector (el cim és l'element amb l'índex més gran).