

# CAP - Macros



Jordi Delgado, Gerard Escudero,

Tema 7



# Macros: Motivació

Les **Macros de Lisp** (i les macros de Clojure ho són) neixen l'any 1963 amb l'article de Timothy Hart **MACRO definitions for LISP**.

Encara avui dia no gaires llenguatges fora dels de la família de Lisp (Clojure, Scheme, Common Lisp, Dylan, Racket, etc.) tenen una capacitat similar. Julia o Elixir en són dos exemples.

El fet que els llenguatges de la família de Lisp siguin **homoiconics** és el que permet tractar codi com si fossin dades, i transformar programes en programes (metaprogramació) amb facilitat:

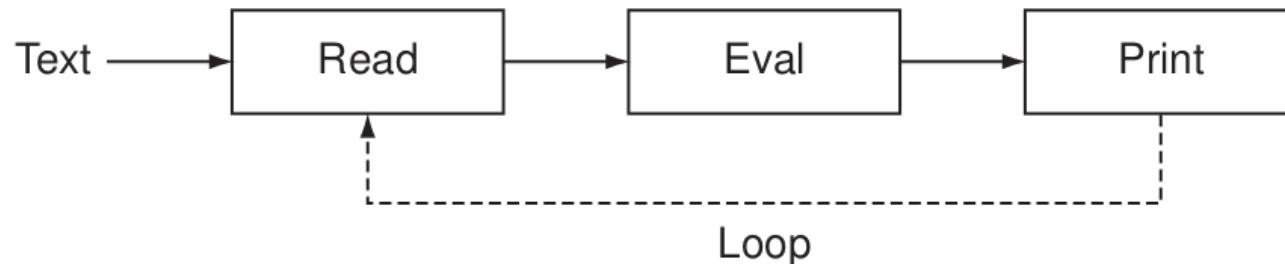
*A language is homoiconic if a program written in it can be manipulated as data using the language. The program's internal representation can thus be inferred just by reading the program itself. This property is often summarized by saying that the language treats **code as data**.*

Aquesta possibilitat de metaprogramació que les macros (de Lisp) ens permeten està darrera la metodologia de programació associada als llenguatges de la família de Lisp. Simplificant, hom resol problemes en Lisp/Clojure creant **Domain-Specific Languages** (DSL's) associats al problema en qüestió, en els que sigui senzill formular (un programa que porti a) una solució del problema.

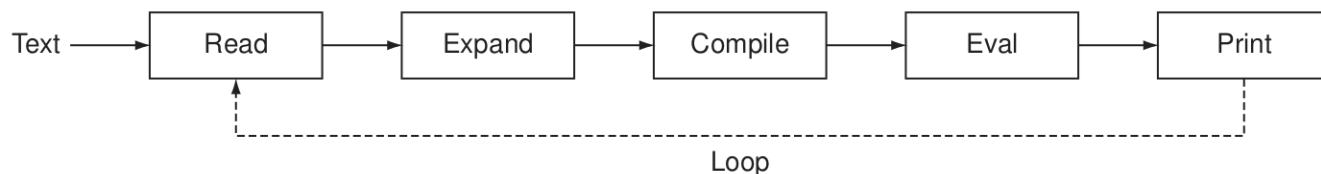
# Macros: Motivació

Us recordieu del primer dia de classe? Dèiem...

És habitual fer servir el **REPL** (*Read, Eval, Print Loop*) en treballar amb Clojure. Provem les funcions que definim fent-ne prototipus i les testem. Accedirem al **REPL** via terminal o via editor/IDE. Sigui com sigui, nosaltres el farem servir molt.



encara que en realitat el que fa és:

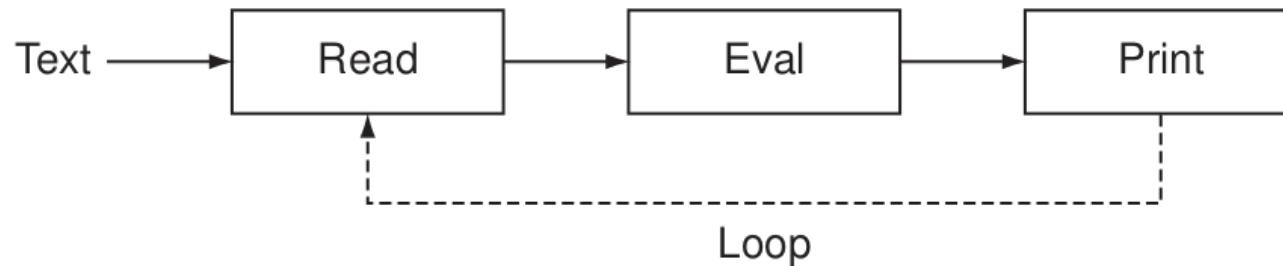


Font: *The Joy of Clojure*, p. 15

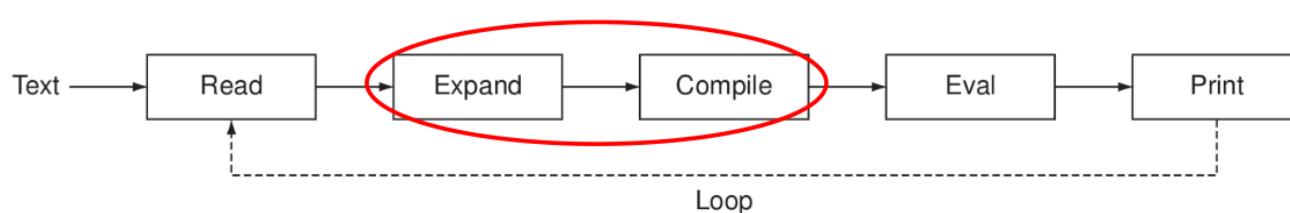
# Macros: Motivació

En aquest tema ens centrarem en el què passa en la part assenyalada...

És habitual fer servir el **REPL** (*Read, Eval, Print Loop*) en treballar amb Clojure. Provem les funcions que definim fent-ne prototipus i les testem. Accedirem al **REPL** via terminal o via editor/IDE. Sigui com sigui, nosaltres el farem servir molt.



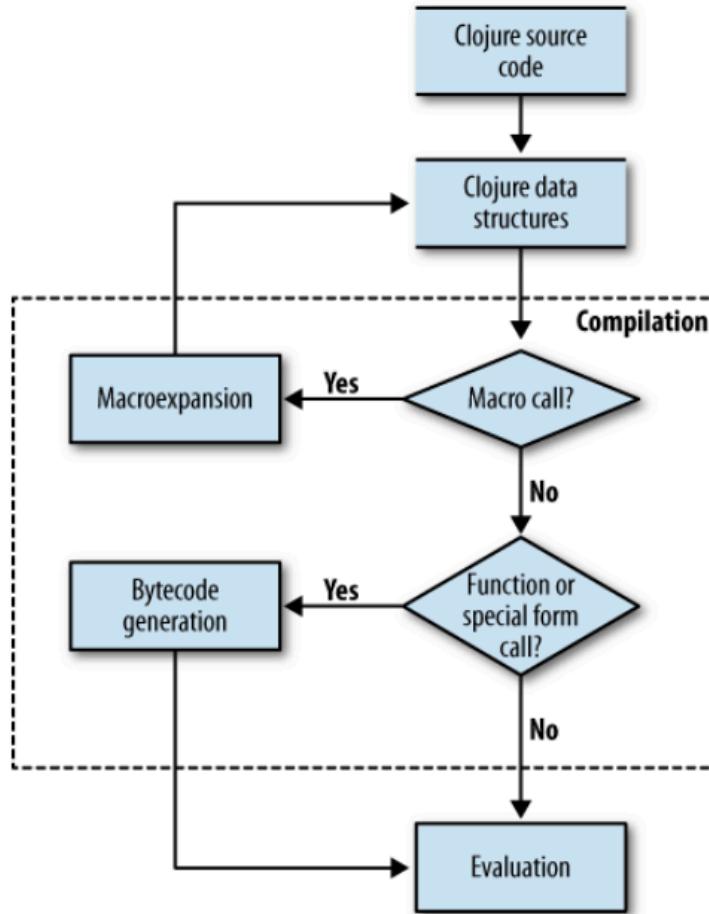
encara que en realitat el que fa és:



Font: *The Joy of Clojure*, p. 15

# Macros: Motivació

Veiem-ho una mica millor representat:



Font: *Clojure Programming*, secció *What is a Macro*, figure 5-1, p. 231

# Macros: Motivació

Podem analitzar què passa en les fases de `read` i `eval` que ja coneixem:

En Clojure la instrucció `read-string` permet obtenir una estructura de dades Clojure a partir d'una *string* de caràcters:

```
(read-string "(+ 1 2)") ➡ (+ 1 2)
```

```
(read-string "(map inc (range 10))") ➡ (map inc (range 10))
```

i `eval` ens permet avaluar una estructura de dades Clojure (interpretant-la com a codi):

```
(eval (read-string "(+ 1 2))) ➡ 3
```

```
(eval (read-string "(map inc (range 10)))) ➡ (1 2 3 4 5 6 7 8 9 10)
```

;; però fixem-nos que el pas de `read-string` és necessari...

```
(eval "(+ 1 2)") ➡ "(+ 1 2)"
```

```
(eval "(map inc (range 10)))" ➡ "(map inc (range 10))"
```

# Macros: Motivació

In computer programming, a macro (short for "macro instruction"; from Greek μακρο- 'long, large') is a rule or pattern that specifies how a certain input should be mapped to a replacement output. Applying a macro to an input is known as macro expansion. ([Wikipedia](#))

Ara bé, a la fase d'**Expansió** de Clojure és on passen *coeses interessants*. Podem veure-ho amb `macroexpand` i variants:

```
(macroexpand (read-string "(+ 1 2)")) ➡ (+ 1 2)  
(macroexpand (read-string "(map inc (range 10))")) ➡ (map inc (range 10))
```

però... no sembla que passi res.

Clojure té moltes macros definides. Una d'elles és `cond`:

```
;; Fem (def x 10) i (def y 15) abans...  
  
(read-string "(cond (> x y) 1000 :else -1000)")  
➡ (cond (> x y) 1000 :else -1000)  
  
(eval (read-string "(cond (> x y) 1000 :else -1000)")) ➡ -1000
```

Tampoc sembla que passi res d'especial!

# Macros: Motivació

A més de `macroexpand`, podem fer servir `macroexpand-all`:

```
;; Hem de fer (use 'clojure.walk) abans, que és on està macroexpand-all  
(macroexpand-all (read-string "(cond (> x y) 1000 :else -1000)"))  
👉 (if (> x y) 1000 (if :else -1000 nil))
```

Ara sí. El resultat de `macroexpand-all` no és el que retorna `read-string`!

S'ha transformat una llista de Clojure (que representa codi) en una altra llista de Clojure (que també representa codi).

La *macroexpansió* és una transformació `Codi font`  $\Rightarrow$  `Codi font` que té lloc *abans* de l'avaluació.

S'acostuma a dir que la macroexpansió té lloc *en temps de compilació*.

Així doncs, l'avaluador de Clojure no "sap" res de cap `cond`, aquest queda transformat en una sèrie de `if` imbricats *abans que el codi arribi a l'avaluador!*

# Macros: Motivació

La transformació de codi font en codi font que fan les macros té lloc *abans* d'avaluar les expressions que contenen les invocacions a les macros.

En realitat, el que fa Clojure és `(eval (macroexpand-all (read-string "...")))*`

```
(eval (macroexpand-all (read-string "(cond (> x y) 1000 :else -1000)")))
```

El procés de *macroexpansió* pot requerir diversos passos. L'exemple que estem treballant n'és un cas. `macroexpand-1` en fa un d'aquests passos:

```
(macroexpand-1 (read-string "(cond (> x y) 1000 :else -1000)"))
👉 (if (> x y) 1000 (clojure.core/cond :else -1000))
;;
-----  
;; Aquí hi ha un 'cond' encara per expandir
;;
;; en canvi:
(macroexpand-all (read-string "(cond (> x y) 1000 :else -1000)"))
👉 (if (> x y) 1000 (if :else -1000 nil))
```

És clar, doncs, que la definició de les macros pot ser *recursiva*.

\* En realitat el comportament intern de *macroexpansió* no queda fidelment reproduït per `macroexpand-all` (de `clojure.walk`) al 100%, però és la variant de `macroexpand` que més s'hi apropa

# Macros: Motivació

Les macros formen part d'un sistema per ampliar un llenguatge de programació fent servir el mateix llenguatge de programació. El mateix *core* de Clojure està ple de macros. Veiem alguns exemples (del [codi font](#) de Clojure):

```
(defmacro when
  "Evaluates test. If logical true, evaluates body in an implicit do."
  {:added "1.0"}
  [test & body]
  (list 'if test (cons 'do body)))

(...)

(defmacro cond
  "Takes a set of test/expr pairs. It evaluates each test one at a
  time. If a test returns logical true, cond evaluates and returns
  the value of the corresponding expr and doesn't evaluate any of the
  other tests or exprs. (cond) returns nil."
  {:added "1.0"}
  [& clauses]
  (when clauses
    (list 'if (first clauses)
          (if (next clauses)
              (second clauses)
              (throw (IllegalArgumentException.
                      "cond requires an even number of forms"))))
    (cons 'clojure.core/cond (next (next clauses)))))))
```

# Macros

Si ens fixem en `when` i `cond`, són exemples on la transformació de codi font en codi font es senzillament manipulació de llistes i els seus continguts, jugant amb `quote` quan cal. Aquesta transformació es fa en Clojure.

Hi ha macros una mica més complicades, que fan servir funcions i macros de Clojure que encara no hem vist:

```
;; Ara resulta que a Clojure teníem 'while'!! 😊
```

```
(defmacro while
  "Repeatedly executes body while test expression is true. Presumes
  some side-effect will cause test to become false/nil. Returns nil"
  [:added "1.0"]
  [test & body]
  `(~(loop []
        (when ~test
          ~@body
          (recur))))
```

Què són aquests símbols `~` o `~@`?

El millor de les macros és que *el programador en pot definir de pròpies*.

Ara veurem com... ➔

# Macros: `defmacro` i els paràmetres

Per definir macros fem servir `defmacro`. Quan s'invoca una macro cal tenir present una propietat importantíssima:

**Els paràmetres de la macro NO s'avaluen**

Cap paràmetre dels que passem a una macro (expressions, llistes, símbols, el que sigui) **NO** s'avalua.

Veiem un exemple. Recordem la definició de `when`:

```
(defmacro when
  "Evaluates test. If logical true, evaluates body in an implicit do."
  {:added "1.0"}
  [test & body]
  (list 'if test (cons 'do body)))

(macroexpand-all '(when (f x) (map g (h x)) nil))
👉 (if (f x) (do (map g (h x)) nil))
```

No s'avalua l'expressió `(f x)`, ni la resta de paràmetres `(map g (h x))` o `nil`. Quan s'invoca la macro `(when (f x) (map g (h x)) nil)`, aquesta invocació queda *textualment* substituïda per `(if (f x) (do (map g (h x)) nil))`.

# Macros: defmacro i els paràmetres

A l'expressió `(when (f x) (map g (h x)) nil)` es fa el que diu la definició de la macro sense avaluar els paràmetres, és a dir, els paràmetres són estructures de dades Clojure i es queden com a tals.

Quan executem el cos del `when`: `(list 'if test (cons 'do body))`, `test` és *literalment* el que hem passat a la invocació de `when`, és a dir, `(f x)`, i `body` és la seqüència amb la resta de paràmetres (com correspon a `&`) sense avaluar: `((map g (h x)) nil)`.

Així, `(list 'if test (cons 'do body))` construeix la llista `(if (f x) (do (map g (h x)) nil))`, que és el que substitueix `(when (f x) (map g (h x)) nil)` i que serà *posteriorment* evaluat.

Veiem, doncs, com amb les funcions per manipular llistes de Clojure (aquí `cons` i `list`) més `quote` podem manipular i *transformar* codi font en codi font. El mateix Clojure ens permet executar una expressió per transformar codi font *abans* de l'execució/avaluació en sí del codi.

El problema és que amb aquestes funcions no en tenim prou.

# Macros: syntax-quote et al

Voldríem tenir una mena de quote selectiu, on es pogués *triar* què s'avalua o no. Això és precisament el que fa `syntax-quote`, altrament conegut pel *back-tick* ` . Com diem el que volem que sigui evaluat dins un `syntax-quote`? Fent servir l'`unquote`, o ~. Amb ` i ~ tenim el que necessitem:

```
user=> (def x 1001)
#'user/x

user=> x    ;; avaliació "normal"
1001

user=> `x   ;; amb syntax-quote
user/x

user=> 'x   ;; amb quote
x

user=> `(list :a :b :c x :d)  ;; syntax-quote actua (gairebé) com un quote...
(clojure.core/list :a :b :c user/x :d)

user=> `(list :a :b :c ~x :d)  ;;; però podem fer servir l'unquote
(clojure.core/list :a :b :c 1001 :d)
```

Quan fem servir `syntax-quote` els símbols no evaluats són *namespace-qualified*. Es fa explícita la seva pertinença a un *namespace* determinat.

# Macros: syntax-quote et al

Hi ha una utilitat més que ens farà la vida més fàcil: l'`unquote-splicing`, o `~@`.

De vegades volem inserir els elements d'una llista en una altra llista:

```
user=> (def lst '(:e :f :g))
#'user/lst

;; si fem servir unquote...
user=> `(:a :b :c :d ~lst :h)
(:a :b :c :d (:e :f :g) :h)

;; no és això el que volem... però si faig servir l'unquote-splicing
user=> `(:a :b :c :d ~@lst :h)
(:a :b :c :d :e :f :g :h)
```

L'`unquote-splicing` és precisament el que fa, si el resultat de l'avaluació és una llista insereix els elements de la llista allà on hem fet servir l'`~@`.

De fet, allò que avaluem amb `~@` ha de resultar obligatòriament en una llista.  
En altre cas obtindrem un `Don't know how to create ISeq from:...`

# Macros: syntax-quote et. al

Ara ja podem entendre el `while`:

```
(defmacro while  ;; sense comentaris
  [test & body]  ;; ni metadades
  `(loop []
    (when ~test
      ~@body
      (recur))))
```

```
(let [n (atom 2)]
  (while (not (zero? @n))
    (println @n)
    (swap! n dec))) ⚡ nil
🕒 2
🕒 1
```

```
(macroexpand-all
  '(let [n (atom 2)] (while (not (zero? @n)) (println @n) (swap! n dec)))

  ⚡ (let* [n (atom 2)]
    (loop* []
      (if (not (zero? (clojure.core/deref n)))
        (do
          (println (clojure.core/deref n))
          (swap! n dec)
          (recur)))))
```

`loop*` i `let*` són versions *internes* de `loop` i `let` que el programador no hauria de fer servir. Tot i això, l'expansió de la macro sí les fa servir. Fixem-nos que també s'ha expandit el `when`.

# Macros: syntax-quote et al

Hem vist l'expansió completa, però podem entendre-la millor per passos (ignorarem el `loop*` i el `let*`):

```
;; primer macroexpandim el while:  
(let [n (atom 2)]           (let [n (atom 2)]  
    (while (not (zero? @n))   ↙ ↙ ↙  
        (println @n)  
        (swap! n dec)))  
    (loop []  
        (when (not (zero? (clojure.core/deref n)))  
            (println (clojure.core/deref n))  
            (swap! n dec)  
            (recur))))  
  
;; ara macroexpandim el when:  
(let [n (atom 2)]  
    (loop []  
        (when (not (zero? (clojure.core/deref n)))  
            (println (clojure.core/deref n))  
            (swap! n dec)  
            (recur))))  
  
    ↗ ↗ ↗  
  
(let [n (atom 2)]  
    (loop []  
        (if (not (zero? (clojure.core/deref n)))  
            (do  
                (println (clojure.core/deref n))  
                (swap! n dec)  
                (recur)))))
```

# Macros: Captura de símbols

El món de les macros és ple de subtileses, que anirem veient poc a poc. La primera que tractarem és el que s'anomena *symbol capture*.

Comencem pel problema d'escriure símbols dins de `syntax-quote`.

```
(defmacro cubs [s] `(map (fn [x] (* x x x)) ~s)) ⚡ #'user/cubs
(cubs (range 10)) ⚡ Error ;; es queixa perquè no sap què és user/x
(macroexpand '(cubs (range 10))) ⚡
(clojure.core/map (clojure.core/fn [user/x]
                                     (clojure.core/* user/x user/x user/x)) (range 10))
```

En fer servir `cubs` ens trobem que els símbols que fem servir, `x` en particular, és interpretat pel `syntax-quote` com a un símbol del *namespace user*, `user/x`, i no ho és. És el paràmetre d'una funció, i així hauríem d'escriure'l.

Haurem de fer que determinats símbols que ens interessen no siguin processats per `syntax-quote`: Si fem `(unquote (quote x))`, és a dir `~'x`, aconseguim que el codi generat escrigui senzillament `x`:

```
(defmacro cubs [s] `(map (fn [~'x] (* ~'x ~'x ~'x)) ~s)) ⚡ #'user/cubs
(cubs (range 10)) ⚡ (0 1 8 27 64 125 216 343 512 729)
(macroexpand '(cubs (range 10))) ⚡
(clojure.core/map (clojure.core/fn [x] (clojure.core/* x x x)) (range 10))
```

# Macros: Captura de símbols

Ara fem una altra macro: `crea-multiplicador`:

```
(defmacro crea-multiplicador [x] `(fn [~'y] (* ~'y ~x)))
👉 #'user/crea-multiplicador
;; (crea-multiplicador x) macroexpandirà a la closure (fn [y] (* x y))
(def per3 (crea-multiplicador 3)) 👉 #'user/per3
;; per3 és en realitat el resultat d'avaluar (fn [y] (* 3 y))
(per3 10) 👉 30
```

Tot sembla correcte. Anem, però, a fer un experiment:

```
(def y 100) 👉 #'user/y
(def per103 (crea-multiplicador (+ y 3))) 👉 #'user/per103
(per103 10) 👉 130 ;; !!!!
```

Aquest no és el resultat correcte! Esperàvem `1030`. Què ha passat?  
*Macroexpandim "a mà"...*

```
(crea-multiplicador (+ y 3)) 👉 (fn [y] (* (+ y 3) y))
((fn [y] (* (+ y 3) y)) 10) ≡ (* 13 10) ≡ 130
```

Direm que `y` (és a dir, `user/y`) ha estat *capturada* (pel codi generat per la macro).

# Macros: Captura de símbols

Així doncs, què podem fer per evitar la captura de símbols (a banda d'anar amb molt de compte)?

Clojure ens proporciona la funció `gensym`. La seva aplicació ens proporciona un símbol nou, que no ha estat utilitzat fins el moment. Es pot proporcionar un prefix:

```
(gensym) ➡ G__3
```

```
(gensym "y") ➡ y6
```

```
(gensym "nou") ➡ nou9
```

Dins d'una macro podem disposar de l'*auto-gensym*. Podem afegir un sufix `#` a un símbol i es generarà un símbol únic amb el símbol com a prefix:

```
(defmacro crea-multiplicador [x] `(fn [y#] (* y# ~x)))
```

```
➡ #'user/crea-multiplicador
```

```
(def y 100) ➡ #'user/y
```

```
(def per103 (crea-multiplicador (+ y 3))) ➡ #'user/per103
```

```
(per103 10) ➡ 1030
```

# Macros: Captura de símbols

Ara podem entendre altres macros, per exemple l'`and` i l'`or`, del **codi font** de Clojure:

```
(defmacro and
  "Evaluates exprs one at a time, from left to right. If a form
  returns logical false (nil or false), and returns that value and
  doesn't evaluate any of the other expressions, otherwise it returns
  the value of the last expr. (and) returns true."
  {:added "1.0"}
  ([] true)
  ([x] x)
  ([x & next]
   `(~(let [and# ~x]
        (if and# (and ~@next) and#)))))

(defmacro or
  "Evaluates exprs one at a time, from left to right. If a form
  returns a logical true value, or returns that value and doesn't
  evaluate any of the other expressions, otherwise it returns the
  value of the last expression. (or) returns nil."
  {:added "1.0"}
  ([] nil)
  ([x] x)
  ([x & next]
   `(~(let [or# ~x]
        (if or# or# (or ~@next))))))
```

# Macros: Captura de símbols

Així doncs, un exemple d'utilització de l'*auto-gensym*:

```
(macroexpand-all '(and a b))
👉
(let* [and_5579_auto_ a]
  (if and_5579_auto_
      b
      and_5579_auto_))

(macroexpand-all '(and (or x y) (or z t)))
👉
(let* [and_5579_auto_ (let* [or_5581_auto_ x]
                           (if or_5581_auto_
                               or_5581_auto_
                               y))]
  (if and_5579_auto_
      (let* [or_5581_auto_ z]
        (if or_5581_auto_
            or_5581_auto_ t)
        and_5579_auto_)))
```

Fixem-nos com hi ha (*gen*)símbols que s'han "reciclat" on és legítim fer-ho

Cal tenir en compte que els lligams establerts per *special forms* com `let`, `letfn` o la clàusula `catch` de `try` tenen el mateix requeriment que els paràmetres de funcions, de manera que normalment cal utilitzar *auto-gensym* per a aquestes situacions, també.

# Macros: Petita parada, Resum

Form	Description
<code>foo#</code>	Auto-gensym: Inside a syntax-quoted section, create a unique name prefixed with <code>foo</code> .
<code>(gensym prefix?)</code>	Create a unique name, with optional prefix.
<code>(macroexpand form)</code>	Expand form with <code>macroexpand-1</code> repeatedly until the returned form is no longer a macro.
<code>(macroexpand-1 form)</code>	Show how Clojure will expand form.
<code>(list-frag? ~@form list-frag?)</code>	Splicing unquote: Use inside a syntax quote to splice an unquoted list into a template.
<code>'form</code>	Syntax quote: Quote form, but allow internal unquoting so that <code>form</code> acts as a template. Symbols inside <code>form</code> are resolved to help prevent inadvertent symbol capture.
<code>~form</code>	Unquote: Use inside a syntax quote to substitute an unquoted value.

\*Font: *Programming Clojure, 3rd ed.*, Alex Miller with Stuart Halloway and Aaron Bedra, Pragmatic 2018, p. 194

# Macros: &form &&env

Dins de les macros (i només allà) puc disposar de dos variables:

- `&env`: diccionari on les claus són els noms en l'entorn local on expandeixo la macro (els valors són instàncies de classes que Clojure fa servir internament)
- `&form`: l'expressió amb que s'ha invocat la macro

```
(defmacro write-form-and-env []
  (println (str "&form és " &form))
  (println (str "&env és " &env))) ⏪ #'user/write-form-and-env

(write-form-and-env) ⏪ nil

🕒 &form és (write-form-and-env)
🕒 &env és

(let [a "a", b "b"] (write-form-and-env)) ⏪ nil

🕒 &form és (write-form-and-env)
🕒 &env és {a #object[clojure.lang.Compiler$LocalBinding 0x68809cc7...],
            b #object[clojure.lang.Compiler$LocalBinding 0x703feacd...]}
```

# Macros: Les macros NO són *first class*

Les macros **no són valors** en Clojure. No són *ciutadans de primera classe*.

Veiem un exemple:

```
(defn cub [x] (* x x x)) ⚡ #'user/cub  
  
(cub 4) ⚡ 64  
  
(map cub (range 10)) ⚡ (0 1 8 27 64 125 216 343 512 729)  
  
(defmacro cub' [x] `(* ~x ~x ~x)) ⚡ #'user/cub'  
  
(cub' 4) ⚡ 64  
(macroexpand-1 '(cub' 4)) ⚡ (clojure.core/* 4 4 4)  
  
(map cub' (range 10)) ⚡ Syntax error (...).  
Can't take value of a macro: #'user/cub'
```

Si la macro és prou senzilla, hi ha un possible *pegat* per arreglar això:

```
(map (fn [n] (cub' n)) (range 10)) ⚡ (0 1 8 27 64 125 216 343 512 729)
```

però no sempre serveix. Dependrà de la macro.

# Macros: Avaluacions múltiples indesitjades

Ja hem vist com Clojure implementa la macro `and`. Anem a fer una versió pròpia incorrecte:

```
(defmacro and'  ;; sense metadades ni comentari
  ([] true)
  ([x] x)
  ([x & next]
   `(~(if ~x (and' ~@next) ~x))))
```

Veieu on és l'error?

# Macros: Avaluacions múltiples indesitjades

Ja hem vist com Clojure implementa la macro `and`. Anem a fer una versió pròpia incorrecte:

```
(defmacro and'  ;; sense metadades ni comentari
  ([] true)
  ([x] x)
  ([x & next]
   `(~(if ~x (and' ~@next) ~x))))
```

Fem una prova:

```
(and' true true) ➡ true
(and' nil 4) ➡ nil

;; sembla que està bé... però:
(and (do (println "yuju") (= 100 101)) true) ➡ false
👁 yuju

(and' (do (println "yuju") (= 100 101)) true) ➡ false
👁 yuju
👁 yuju
```

Veiem que els `~x` provoquen una doble avaluació, que el `let` de l'`and` original evita.

# Macros: Gimnàstica de Macros

- Implementa una macro per proporcionar una versió senzilla de `defn`. La podem anomenar `defn'` (es pot fer una solució en dues línies de codi).
- Implementa una macro per proporcionar l'estructura de control `do-while` a Clojure i fes-la servir per fer una funció `jugar-a-endevinar-número` que implementi un joc d'endevinar un nombre entre 1 i 100 (inclosos), amb pistes:

```
(jugar-a-endevinar-número ) ⏴ nil
  ⚡ Número? 50
  ⚡ És més gran
  ⚡ Número? 75
  ⚡ És més gran
  ⚡ Número? 87
  ⚡ És més petit
  ⚡ Número? 81
  ⚡ És més petit
  ⚡ Número? 78
  ⚡ És més petit
  ⚡ Número? 76
  ⚡ Trobat!
```

Us anirà bé saber què fan `print`, `flush`, `Integer/valueOf` i `read-line`.

# Macros: Gimnàstica de Macros

- Explica per quina raó observem aquest comportament:

```
(defmacro incognita []
  (println "Quan s'executa aquest println?")
  `(println "Quan s'executa aquest altre println?")) ⏪ #'user/incognita

(incognita) ⏪ nil
  ⚡ Quan s'executa aquest println?
  ⚡ Quan s'executa aquest altre println?

(defn utilitza-la-macro [] (incognita)) ⏪ #'user/utilitza-la-macro
  ⚡ Quan s'executa aquest println?

(utilitza-la-macro) ⏪ nil
  ⚡ Quan s'executa aquest altre println?
```

- Feu una macro `rand-expr` que, donades dues expressions `e1` i `e2` es *macroexpandeixi* a una d'elles **a l'atzar**. Cada cop que hi ha una *macroexpansió* té lloc la tria aleatòria.

```
(rand-expr (+ 1 2) (* 3 4)) ⏪ 12
(rand-expr (+ 1 2) (* 3 4)) ⏪ 12
(rand-expr (+ 1 2) (* 3 4)) ⏪ 3
(macroexpand '(rand-expr (+ 1 2) (* 3 4))) ⏪ (* 3 4)
(macroexpand '(rand-expr (+ 1 2) (* 3 4))) ⏪ (+ 1 2)
```

# Macros: Gimnàstica de Macros

- Suposem que volem fer un `fold` amb una macro: Donada una funció de dos arguments  $\oplus$ , cal que faci un `fold` explícit, és a dir, la crida a `macro-fold` s'expandeixi en les corresponents aplicacions de  $\oplus$ .

```
;; Fem el següent:  
(defmacro macro-fold [f x0 s]  
  (if (empty? s)  
      `x0  
      `(~f ~(~first s) (macro-fold ~f ~x0 ~(next s))))))  
  
;; Ho provem...  
(macro-fold (fn [x a] (+ x (* 2 a))) 0 [1 2 3 4]) ➡ 49  
(macroexpand-all '(macro-fold (fn [x a] (+ x (* 2 a))) 0 [1 2 3 4])) ➡  
((fn* ([x a] (+ x (* 2 a))))  
 1  
  ((fn* ([x a] (+ x (* 2 a))))  
  2  
   ((fn* ([x a] (+ x (* 2 a))))  
  3  
   ((fn* ([x a] (+ x (* 2 a))))  
  4  
    0))))  
(macroexpand-all '(macro-fold ⊕ x0 [x1,x2,x3,x4,x5])) ➡  
(~⊕ x1 (~⊕ x2 (~⊕ x3 (~⊕ x4 (~⊕ x5 x0)))))
```

Sembla que `macro-fold` funciona bé... però no, no és correcte. Per què?  
Suggereix un contra-exemple.

# Macros: Usos

Quan convé fer servir macros?

Mirem primer el principal consell sobre quan NO fer servir macros:

**NO fer macros si podem fer servir funcions**

Dit això, existeixen situacions on SÍ convé fer servir macros.

Un ús típic de les macros és el cas de voler afegir **noves estructures de control** a Clojure.

Per exemple, pels que enyorin el `for` de "tota la vida", podríem voler afegir a Clojure una construcció `for-loop` que funcionés de la següent manera:

```
;; Volem fer quelcom similar a for (i = 0; i < 5; ++i) ...
```

```
(for-loop [i 0, (< i 5), (inc i)] (println i)) ➡ nil
```

- 👁 0
- 👁 1
- 👁 2
- 👁 3
- 👁 4

# Macros: Usos

Fixem-nos, com a entrada tenim un vector [`i 0, (< i 5), (inc i)`] i una col·lecció d'expressions que cal avaluar com a *cos del bucle*, en aquest cas només n'hi ha una (`println i`).

Cal recordar, i entendre, que en el procés de *macroexpansió* podem fer servir Clojure de manera completa. Per exemple, a les macros podem fer *destructuring* a l'hora de gestionar els paràmetres:

```
(defmacro for-loop [[simbol inicial condicio canvi :as params] & cos]
  ...)
```

Així, `simbol` ≡ `i`, `inicial` ≡ `0`, `condicio` ≡ `(< i 5)`, `canvi` ≡ `(inc i)` i finalment `cos` ≡ `((println i))` (atenció al &). També, com tenim `:as, params` ≡ `[i 0, (< i 5), (inc i)]`.

Ara cal transformar aquesta informació en una expressió Clojure amb `loop/recur`:

```
`(loop [~simbol ~inicial valor# nil]    ; utilitzem l'auto-gensym
  (if ~condicio
    (let [nou-valor# (do ~@cos)]
      (recur ~canvi nou-valor#))
    valor#))
```

# Macros: Usos

Si ho posem tot junt:

```
(defmacro for-loop [[simbol inicial condicio canvi :as params] & cos]
  `(loop [~simbol ~inicial valor# nil]   ;; utilitzem l'auto-gensym
    (if ~condicio
        (let [nou-valor# (do ~@cos)]
          (recur ~canvi nou-valor#))
        valor#)))
```

Provem-ho:

```
(for-loop [i 0, (< i 5), (inc i)] (println i)) 👉 nil
  ⚡ 0
  ⚡ 1
  ⚡ 2
  ⚡ 3
  ⚡ 4

(macroexpand '(for-loop [i 0, (< i 5), (inc i)] (println i)))
  👉
  (loop* [i 0 valor__3__auto__ nil]   ;; pretty-printed
    (if (< i 5)
        (clojure.core/let [nou-valor__4__auto__ (do (println i))]
          (recur (inc i) nou-valor__4__auto__))
        valor__3__auto__))
```

# Macros: Usos

Opcionalment podem fer la comprovació d'alguns possibles errors, en cas de no fer servir de manera adequada la macro:

```
(defmacro for-loop [[simbol inicial condicio canvi :as params] & cos]
  (cond
    (not (vector? params))
    (throw (Error. "El 1r argument ha de ser un vector amb el format adequat"))

    (not= 4 (count params))
    (throw (Error. "Calen exactament 4 elements per definir el for"))

    :else
    `(~(loop [~simbol ~inicial valor# nil]      ;; utilitzem l'auto-gensym
         (if ~condicio
             (let [nou-valor# (do ~@cos)]
               (recur ~canvi nou-valor#))
             valor#)))))

(for-loop [i 0, (< i 5)] (println i)) ;; recordeu que les comes són espais
👉
Unexpected error (Error) macroexpanding for-loop at (REPL:1:1).
Calen exactament 4 elements per definir el for
```

# Macros: Usos

Alguns usos habituals de macros tenen a veure amb aspectes de Clojure que no hem explicat durant el curs. Un d'ells és la gestió implícita de **variables amb lligam dinàmic** (de vegades s'anomenen **variables dinàmiques**).

Podem veure un exemple amb variables dinàmiques que Clojure defineix per defecte. Una d'elles és l'equivalent al `stdout` dels sistemes Unix que tots coneixem: `*out*` (les variables amb lligam dinàmic s'acostumen a escriure amb dos `*`, al principi i al final del nom).

Funcions que sí hem fet servir sovint, com `println`, escriuen en realitat a `*out*`. Si redefinim `*out*`, redefinirem el lloc on `println` envia el seu *input*. Al **codi font** de Clojure trobem:

```
(defmacro with-out-str
  "Evaluates exprs in a context in which *out* is bound to a fresh
  StringWriter. Returns the string created by any nested printing
  calls."
  [& body]
  `(~@body
    (binding [*out* (new java.io.StringWriter)] ; Interop amb Java, no ho hem vist
      (str s#))))))

(with-out-str (println "Això hauria de retornar com a string"))
  "Això hauria de retornar com a string\n" ; fixem-nos que no ha retornat nil
```

# Macros: Usos

Un altre dels usos habituals de les macros és estalviar processos en temps d'execució. Amb les macros podem aprofitar informació que ja tenim en temps de compilació. Un exemple senzill és el de calcular la mitjana d'una sèrie de nombres:

```
(defn mitjana-f [& args]
  (/ (apply + args) (count args)))
(defmacro mitjana-m [& args]
  `(/ (+ ~@args) ~(count args)))
(mitjana-f 0 1 2 3 4 5 6 7 8 9) 9/2
(mitjana-m 0 1 2 3 4 5 6 7 8 9) 9/2
```

Què hi guanyem amb la macro? El nombre d'arguments es coneix en temps de compilació i no cal cridar `count` en temps d'execució:

```
(macroexpand '(mitjana-m 0 1 2 3 4 5 6 7 8 9))
👉
(clojure.core// (clojure.core/+ 0 1 2 3 4 5 6 7 8 9) 10)
```

però...

```
(apply mitjana-f (range 10)) 9/2
(apply mitjana-m (range 10)) 👈
Syntax error ( ...) Can't take value of a macro: #'user/mitjana-m
```

# Macros: Modismes Habituals

- Quan una macro introduceix lligams locals (à la `let`) hauria de fer-ho en un vector de noms i valors (inicials). Exemples: `for`, `if-let`, `with-open`, etc.
- Les macros no haurien d'"*amagar*" comportament complex. El mateix Clojure, però, trenca aquesta norma: El cas del `for`.
- Si la macro defineix una `var`, cal fer-ho d'acord a unes normes: nom que comença amb `def`, el nom del `var` com a primer argument, definir un `var` per crida a la macro.

# Macros: Exemple ->>>

Ja hem conegit les *threading macros*, la macro *threading first* -> i la macro *threading last* ->>. Veiem tot seguit com fer una *threading macro* més general\*: ->>>.

Amb aquesta macro composarem resultats igual que fèiem amb -> i ->>, però en aquest cas podrem indicar amb un *underscore* \_ on volem el resultat (en lloc de posar-lo sempre com a primer argument o com a darrer argument):

```
(->>> 1      ; valor inicial d'1
  (+ _ 2)    ; amb el resultat previ d'1, això és: (+ 1 2)
  (+ 3 _)    ; resultat anterior de 3, (+ 3 3)
  (- 50 _)   ; resultat anterior de 6, (- 50 6)
  (/ _ 2))   ; resultat anterior de 44, (/ 44 2) => resultat final: 22
👉 22
```

El resultat que volem després de *macroexpandir* seria quelcom similar a:

```
(let [init 1]
  (let [res0 (+ init 2)]
    (let [res1 (+ 3 res0)]
      (let [res2 (- 50 res1)]
        (let [res3 (/ res2 2)]
          res3))))
```

\*Font: Entrada blog *The Anatomy of a Clojure Macro*, de Bryan Gilbert, 30 Juliol 2013

# Macros: Exemple - >>>

Definirem algunes senzilles funcions auxiliars:

```
(defn substituir-si-underscore [element val]
  (if (= element '_)
      val
      element))

(substituir-si-underscore '_ 1) ➡ 1
(substituir-si-underscore '+ 1) ➡ +

(defn substituir-underscores [expressio val]
  (map #(substituir-si-underscore % val) expressio))

(substituir-underscores '(+ _ 2) 1) ➡ (+ 1 2)
(substituir-underscores '(+ 3 2) 1) ➡ (+ 3 2)
```

**Exercici:** La funció `substituir-underscores` només substitueix l'*underscore* quan apareix en una expressió en el, diguem-ne, "*primer nivell*":

```
(substituir-underscores '(* (- (+ _ _)) (* 2 _)) (/ 10 _)) 2)
➡ (* (- (+ _ _)) (* 2 _)) (/ 10 _)); No és això el que volem!
```

Modifiqueu la funció per a que faci la substitució en qualsevol subexpressió de l'expressió que rep com a argument. Podeu fer servir `postwalk`, explicat a la *Frikada Final*

# Macros: Exemple - >>>

Aquesta funció auxiliar que ara definirem, `convertir-expressions`, *sempre* una macro però no ho és. No hi ha res que impedeixi una funció retornar una llista que sigui codi en realitat.

```
(defn convertir-expressions [val [propera-expressio & altres-expressions]]
  (if (nil? propera-expressio)
    val
    (let [next-val (gensym)]
      `(let [~next-val ~(substituir-underscores propera-expressio val)]
        ~(convertir-expressions next-val altres-expressions)))))

(convertir-expressions 2 '((+ _ 1) (+ 4 _)))
👉
(clojure.core/let [G_409 (+ 2 1)]
  (clojure.core/let [G_410 (+ 4 G_409)]
    G_410))

(convertir-expressions 1 '((+ _ 2) (+ 3 _) (- 50 _) (/ _ 2)))
👉
(clojure.core/let [G_413 (+ 1 2)]
  (clojure.core/let [G_414 (+ 3 G_413)]
    (clojure.core/let [G_415 (- 50 G_414)]
      (clojure.core/let [G_416 (/ G_415 2)]
        G_416))))
```

# Macros: Exemple ->>>

Definir ara la macro ->>> és senzill:

```
(defmacro ->>> [inicial & expressions]
  (convertir-expressions inicial expressions))
```

Fixem-nos que la macro crida a una funció que s'executa retornant el codi en el que la crida a la macro és re-escriu. És una combinació que no havíem vist fins ara, però possible.

Podem ara recuperar els exemples de -> i ->>:

```
(->>> {})
  (assoc _ :clau1 24)
  (assoc _ :clau2 36)
  (assoc _ :clau3 48))
👉 {:_clau1 24, :_clau2 36, :_clau3 48}
```

```
(->>> (range 10)
  (filter odd? _)
  (map #(* % %) _)
  (reduce + _))👉 165
```

```
(->>> []
  (conj _ 3)
  (conj _ 3)
  (conj _ 7))
👉 [3 5 7]
```

```
(defn prod-of-evens' [s]
  (->>> s
    (filter even? _)
    (apply * _)))
```

```
(prod-of-evens' (range 1 21))
👉 3715891200
```

# Macros: Més Gimnàstica de Macros

- Hem vist les macros de Clojure per implementar les operacions lògiques `and` i `or`. Feu una macro que implementi l'operació `nand` (si no sabeu què és, mireu [aquí](#))
- Generalitzem la macro `rand Expr` d'un problema anterior. Feu una macro `randomly` que, donada una col·lecció d'expressions, triï una a l'atzar per avaluar. Al tanto que la tria s'ha de fer en temps d'execució, no en temps de compilació. En aquest problema us poden ajudar l'estructura condicional `case` i la funció `interleave` de Clojure, tot i que no les hem vist durant el curs.
- Feu una macro `do-until`, amb una estructura sintàctica com la de `cond`, que executa totes les seves clàusules que avaluuen a un valor *truthy*, fins que en troba una que és *falsy*. Us podeu inspirar en la macro `cond`.

```
(do-until
  (even? 2)  (println "Aquesta l'escric")
  (odd? 3)   (println "Aquesta també")
  (zero? 1)   (println "Aquesta no la veuràs")
  :whatever   (println "Aquesta tampoc")) ⏵ nil
```

- 👁 Aquesta l'escric
- 👁 Aquesta també

# Macros: *Frikada Final*

El mòdul on trobem `macroexpand-all`, anomenat `clojure.walk`, també té altres funcions. Una d'elles és `postwalk`, que et permet recórrer llistes imbricades, aplicant una funció a cada element trobat:

```
(use '[clojure.walk :as w])

(w/postwalk #(if (symbol? %) (println %) (if (number? %) (println (inc %)))
              '(map inc [1 2 3])))

map
inc
2
3
4

(w/postwalk #(if (symbol? %) (println %) (if (number? %) (println (inc %)))
              '(map inc ['a 'b 'c])))

map
inc
quote
a
quote
b
quote
c
```

# Macros: *Frikada* Final

Fent servir `postwalk` podem fer una macro... curiosa:

```
(use '[clojure.string :as s])

(defmacro misteri
  [form]
  (w/postwalk
    #(if (symbol? %)
       (symbol (s/reverse (name %)))
       %)
    form))
```

Penseu què fa aquesta macro...

Veiem-ne un exemple:

```
(misteri (pool [i 5]
  (fi (ton (?orez i))
    (od (nltnirp i)
      (rucer (ced i))))))
```



nil

- 5
- 4
- 3
- 2
- 1

# Macros: *Frikada* Final

La solució ens la dóna `macroexpand`:

```
(macroexpand '(misteri (pool [i 5]
                               (fi (ton (?orez i)))
                               (od (nltnirp i)
                                    (rucer (ced i))))))
👉

(loop [i 5] ;; en realitat és loop*
  (if (not (zero? i))
    (do (println i)
        (recur (dec i)))))
```

I tenim una macro que ens permet escriure els símbols de Clojure a l'inrevés\*!



\*Font: *Clojure Programming*, secció *Writing Your First Macro*, p. 236

# Macros

Hi ha molt que no hem explicat sobre les macros: macros que generen funcions, macros que generen macros, etc. Es podria fer un curs sencer només sobre macros.

Per saber-ne més caldrà recòrrer a fonts en Common Lisp, ja que és on més s'ha desenvolupat la difusió de les capacitats de les *macros de Lisp*.

Dos llibres destaquen:

- **On Lisp**, de Paul Graham, Prentice Hall 1993. ISBN 9780130305527. El podeu descarregar (legalment) [aquí](#).
- **Let Over Lambda**, de Doug Hoyte, Lulu 2008. ISBN 9781435712751. [Web](#).

Us caldrà, però, aprendre Common Lisp per entendre'ls. Les meves recomanacions són:

- **Practical Common Lisp**, de Peter Seibel, Apress 2005. ISBN 9781590592397. [Web](#)
- **ANSI Common Lisp**, de Paul Graham, Prentice Hall 1995. ISBN 9780133708752. [Web](#).