# CAP - *Closures* - Model d'Entorns

**Jordi Delgado, Gerard Escudero,**

Tema 3


# *Closures*: Introducció

## Definició

*In programming languages, a* ***closure****, also lexical closure or function closure, is a technique for* ***implementing lexically scoped name binding in a language with first-class functions****. Operationally, a closure is a record storing a* ***function together with an environment****. The environment is a mapping associating each free variable of the function (variables that are used locally, but defined in an enclosing scope) with the value or reference to which the name was bound when the closure was created. Unlike a plain function,* ***a closure allows the function to access those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope****.*

Font: Wikipedia

# *Closures*: Introducció

Definim una funció `sumador`:

```
(defn sumador [n] (fn [x] (+ n x)))
```

Invocar aquesta funció implica retornar una funció:

```
(def suma_5 (sumador 5))
```

Ara `suma_5` està vinculat a una funció `(fn [x] (+ n x))` amb un nom ***lliure***, `n`, tot i que:

```
(suma_5 1000) 👉 1005
(suma_5 -5)   👉 0
(suma_5 25)   👉 30
```

és a dir, `n` està vinculat al valor 5. Així, `n` és lliure respecte a la funció `(fn [x] (+ n x))` (`n` no és un paràmetre de la funció, tampoc és local), però aquesta funció `suma_5` *d'alguna manera sap* que existeix un lligam entre `n` i el valor `5`.

**D'on treu `suma_5` aquest lligam entre `n` i el valor `5`?**

# *Closures*: Introducció

A Clojure la ***visibilitat*** dels diferents símbols ve determinada pel seu ***context lèxic*** (***lexical scope***), que és fàcilment determinat a partir del *text* del programa (per això s'anomena *lèxic*).

En l'exemple de la plana anterior, el context lèxic de `(fn [x] (+ n x))` conté el símbol `n`, i qualsevol lligam que aquest pugui tenir amb un valor.

Així, el que ha retornat la funció `sumador` **NO** és *només* una funció, és **una funció que ha capturat el seu context lèxic**.

Això és el que anomenarem ***closure***: *funció que captura el seu context lèxic*.

Els paràmetres d'una funció formen part del context lèxic del cos de la funció. La definició de símbols locals fent servir `let` també afegeix símbols al context lèxic del cos del `let`.

```
(def sumador #(let [n %] (fn [x] (+ n x)))) 👉 #'user/sumador

(def suma_10 (sumador 10)) 👉 #'user/suma_10

(suma_10 50) 👉 60
```

# *Closures*: Introducció

Cal entendre bé el concepte de *captura del context lèxic*, ja que una funció es ***crea*** dins un determinat context lèxic i es pot ***cridar*** (***invocar***) dins d'altres contextos lèxics. Veiem-ne un exemple senzill: En aquest codi,

```
(def foo (let [bar 1]
           (fn [] bar))) 👉 #'user/foo

(foo) 👉 1

(let [bar 5] (foo)) 👉 ???
```

Quin valor retorna `(let [bar 5] (foo))`?

# *Closures*: Introducció

Cal entendre bé el concepte de *captura del context lèxic*, ja que una funció es ***crea*** dins un determinat context lèxic i es pot ***cridar*** (***invocar***) dins d'altres contextos lèxics. Veiem-ne un exemple senzill: En aquest codi,

```
(def foo (let [bar 1]
           (fn [] bar))) 👉 #'user/foo

(foo) 👉 1

(let [bar 5] (foo)) 👉 ???
```

Quin valor retorna `(let [bar 5] (foo))`?

```
(def foo (let [bar 1]
           (fn [] bar))) 👉 #'user/foo

(foo) 👉 1

(let [bar 5] (foo)) 👉 1
```

Ja que el nom `bar` (i el corresponent lligam al valor `1`) capturat per `foo` és el que pertany al ***context lèxic en el moment de crear la funció***, no al context lèxic en el moment de *cridar* la funció.

# *Closures*: Funcions d'ordre superior

Les *closures* poden ***compartir*** el context lèxic:

```
(defn sumador_amb_consulta
  "Retorna dues closures que comparteixen el mateix context lèxic"
  [n]
  [#(+ % n), #(identity n)])

(let [[suma, què-val-n] (sumador_amb_consulta 100)]
  (println (suma 100))
  (println (suma 10))
  (què-val-n)) 👉 100
👁️ 200
👁️ 110
```

Les dues funcions `suma` i `què-val-n` han capturat el context lèxic del moment en que han estat ***creades***, dins la invocació de `sumador_amb_consulta`. Aquest context lèxic, però, és el ***mateix*** per a les dues funcions.

Fixem-nos que aquest context lèxic capturat prové de paràmetres de funcions o bé de noms locals que lliguem amb `let`. Això implica que aquests contextos lèxics són ***immutables***, és a dir, un cop lligats els noms a uns valors, no podem modificar aquest lligam. Els valors lligats als noms capturats estan "*congelats*".

De moment, continuarem explorant les possibilitats de les *closures* que capturen un context lèxic immutable...

# *Closures*: Estructures associatives amb funcions

Si tenim funcions d'ordre superior podem implementar estructures de dades. Perdem en eficiència, però aquest és un exercici, un cop més, en expressivitat.

Veiem un exemple: Estructures associatives. Imaginem que volem crear una estructura que em permeti associar claus a valors, i fer cerques sobre aquestes claus per recuperar els valors associats. Operacions que volem sobre aquesta estructura: `crear`/`cercar`/`afegir`.

Volem construir una estructura així fent servir ***només*** funcions.

```
(defn crear []
    (fn [_] :default)) ;; retornem una funció que retorna un valor per defecte

(defn cercar [estructura clau]
    (estructura clau))

(defn afegir [estructura [clau valor]]
    (fn [clau']
        (if (= clau' clau)
            valor
            (cercar estructura clau'))))

(def d (reduce afegir (crear) [[:a 1], [:b 2], [:c 3], [:d 4]]))

(cercar d :b) 👉 2
(cercar d :d) 👉 4
(cercar d :e) 👉 :default
```

# *Closures*: Estructures associatives amb funcions

Podem fer quelcom de similar d'una manera més compacta (i més difícil d'entendre 😀)

```
(defn crear
  ([] (crear (fn [_] :default)))
  ([prev]
   (letfn [(afegir [k v]
             (letfn [(cercar [k2]
                       (if (= k k2) v
                           (prev k2)))]
               [cercar, (crear cercar)]))]
     afegir)))

(let [ afegir1 (crear)
      [cercar2, afegir2] (afegir1 :a 1)
      [cercar3, afegir3] (afegir2 :b 2)
      [cercar4, afegir4] (afegir3 :c 3)
      [cercar5, afegir5] (afegir4 :d 4)]
  (println (cercar5 :a))
  (println (cercar5 :c))
  (println (cercar5 :b))
  (println (cercar5 :d))
  (println (cercar3 :c))) 👉 nil
```

👁 1

👁 3

👁 2

👁 4

👁 :default

# *Closures*: Un altre exemple

Veiem un altre exemple: Implementeu `repetits`, que és una funció que retorna una funció per detectar arguments repetits. Com a efecte secundari de les crides repetides, s'escriu cada argument que s'ha utilitzat abans en una seqüència de crides repetides. Per tant, si un argument apareix $n$ vegades, s'escriu $n - 1$ vegades en total, cada cop que el trobi (diferent del primer cop). La funció `detector` forma part de la implementació de `repetits`, cal determinar com s'utilitza*. Important: *no podeu utilitzar cap llista, conjunt o qualsevol altre col·lecció*

```
(defn detector
  [f]
  (letfn [(g [i]
            (when (f i) (println i))
            (detector #(or (= % i) (f %))))]
    g))
(defn repetits
  [k]
  ??)

(def f ((((((((((repetits 1) 7) 7) 3) 4) 2) 5) 1) 6) 5) 1)) 👉 #'user/f
👁️ 7
👁️ 1
👁️ 5
👁️ 1
```

* Font

# *Closures*: Un altre exemple

Veiem un altre exemple: Implementeu `repetits`, que és una funció que retorna una funció per detectar arguments repetits. Com a efecte secundari de les crides repetides, s'escriu cada argument que s'ha utilitzat abans en una seqüència de crides repetides. Per tant, si un argument apareix $n$ vegades, s'escriu $n - 1$ vegades en total, cada cop que el trobi (diferent del primer cop). La funció `detector` forma part de la implementació de `repetits`, cal determinar com s'utilitza*. Important: *no podeu utilitzar cap llista, conjunt o qualsevol altre col·lecció*

```
(defn detector
  [f]
  (letfn [(g [i]
            (when (f i) (println i))
            (detector #(or (= % i) (f %))))]
    g))
(defn repetits
  [k]
  ((detector (fn [_] false)) k))

(def f ((((((((((repetits 1) 7) 7) 3) 4) 2) 5) 1) 6) 5) 1)) 👉 #'user/f
👁️ 7
👁️ 1
👁️ 5
👁️ 1
```

* Font

# *Closures*: Un altre exemple

Fixeu-vos que el que retorna la crida repetida a `repetits` té "*memòria*" (!!)

```
(defn detector
  [f]
  (letfn [(g [i]
            (when (f i) (println i))
            (detector #(or (= % i) (f %))))]
    g))

(defn repetits
  [k]
  ((detector (fn [_] false)) k))

(def f ((((((((((repetits 1) 7) 7) 3) 4) 2) 5) 1) 6) 5) 1)) 👉 #'user/f
👁 7
👁 1
👁 5
👁 1

(def ff (f 2))      👉 #'user/ff
👁 2

(def fff (ff 8))    👉 #'user/fff

(def ffff (fff 8)) 👉 #'user/ffff
👁 8
```

# *Closures*: Un patró d'ús

Aquests dos darrers exemples, la versió compacta de `crear` i el `repetits`, tenen un *patró comú*, freqüent en l'ús del retorn de *closures*. Mirem de generalitzar (informalment):

```
(defn fun
  [f]
  (letfn [(g [params de g]
             ... crear una nova funció nova_f, que inclogui alguna crida a f
             ... crida a (fun nova_f))]
    g))
```

Això permet anar explotant els diferents contextos lèxics que s'han anat capturant a mida que es va cridant la funció retornada `g` en diversos estadis del procés a realitzar. Així, podem anar *emmagatzemant* informació, malgrat la immutabilitat dels contextos capturats.

La crida `(fun nova_f)` en realitat només serveix per donar més context a la nova `g` que serà retornada, a partir de la funció `nova_f` que ja fa servir la `f` que s'ha utilitzat com a argument en la crida anterior.

Estudiem amb detall l'exemple del `repetits`, per il·lustrar el patró...

# *Closures*: Un patró d'ús

```
;; (repetits 1) en realitat crida a ((detector (fn [_] false)) 1)
;; Pas 1        f1
;;        -------------
(detector (fn [_] false)) ➡️  g1
;; on (g1 1) és
 (when (f1 1) (println 1))
 (detector #(or (= % 1) (f1 %)))

;; Pas 2           f2
;;        --------------------
(detector #(or (= % 1) (f1 %))) ➡️  g2
;; on (g2 7) és (suposem que la següent crida es fa amb 7)
 (when (f2 7) (println 7))
 (detector #(or (= % 7) (f2 %)))

;; Pas 3           f3
;;        --------------------
(detector #(or (= % 7) (f2 %))) ➡️  g3
;; on (g3 1) és (suposem que la següent crida es fa amb 1)
 (when (f3 1) (println 1))
 (detector #(or (= % 1) (f3 %)))

;; Pas 4           f4
;;        --------------------
(detector #(or (= % 1) (f3 %))) ➡️  g4
;; on (g4 5) és (suposem que la següent crida es fa amb 5)
 (when (f4 5) (println 5))
 (detector #(or (= % 5) (f4 %)))

;; ...
```

# *Closures*: Un patró d'ús

Fixem-nos què passa en avaluar les diferents `g` que van apareixen, concretament, en les crides a `(fk i)` que determinen si un argument `i` s'escriu o no:

```
;; Pas 1
(f1 1) = ((fn [_] false) 1) = false  ;; per tant l'1 no s'escriu

;; Pas 2
(f2 7) = (#(or (= % 1) (f1 %)) 7) = (f1 7) =
                                  = ((fn [_] false) 7) = false
;; per tant el 7 no s'escriu

;; Pas 3
(f3 1) = (#(or (= % 7) (f2 %)) 1) = (f2 1) =
                                  = (#(or (= % 1) (f1 %)) 1) = true
;; per tant el 1 sí s'escriu

;; Pas 4
(f4 5) = (#(or (= % 1) (f3 %)) 5) = (f3 5) =
                                  = (#(or (= % 7) (f2 %)) 5) = ... = false
;; per tant el 5 no s'escriu

;; ...
```

Com que cada `fk` és un *closure*, ha capturat l'ús de `f{k-1}` i així, *implícitament*, la invocació de totes les funcions que s'han definit fins el moment, sent capaç de *memoritzar* (en els contextos lèxics capturats) cada un dels elements que han aparegut.

# *Closures*: El model d'entorns

Una manera de copsar aquest mecanisme de captura del context lèxic és mitjançant el ***model d'entorns***[1], que ens permet entendre: **Què passa quan es crida una funció?**[2].

Quan es ***crida*** una funció es ***crea un nou entorn*** on s'aparellen els paràmetres (formals) i els arguments de la crida a la funció. L'aparellament és posicional si no hi intervenen mecanismes de *destructuring*, en altre cas l'aparellament ve determinat per l'esquema de *destructuring* proporcionat.

Aquest nou entorn creat en cridar la funció serà l'entorn on s'ubiquen els noms nous que apareixen en el cos de la funció. Aquest entorn nou s'ha de vincular a l'***entorn de creació de la funció cridada*** (anomenat ***entorn pare*** del nou entorn creat), que és on començarà la cerca dels noms *lliures* que apareixen en el cos de la funció.

Així es crea una ***cadena d'entorns*** que comença en l'entorn pare de la funció-*closure* (l'entorn on s'ha *creat*), i és essencialment el ***context lèxic*** que la funció-*closure* captura.

1 Cal tenir clar que és un *model*, és a dir, no és una descripció acurada de com ho implementa Clojure. És una versió simplificada que ens permet entendre què succeeix, sense entrar en tot detall.

2 Veure SICP, capítol 3 (encara que s'explica fent servir Scheme) o Composing Programs, seccions 1.3 i 1.6, per a una explicació fent servir Python (sense mencionar el `let`)

# *Closures*: El model d'entorns

I per què parlem només de les crides a funció? Per què no parlem dels contextos lèxics que es creen en un `let`?

***Un `let` és essencialment una sèrie de crides a funció.***

Fer (ignorem el *destructuring*, per simplificar, però no perdem generalitat)

```
(let [x1 expr1, x2 expr2, ..., xn exprn] cos-del-let)
```

és el mateix (en el que pertoca el que estem tractant, poden haver diferències en aspectes que no són importants ara en aquest context) que:

```
((fn [x1]
     ((fn [x2]
          ...
          ((fn [xn] cos-del-let) exprn)...) expr2) expr1))
```

**Exercici**: Un `let` podria ser equivalent a:

```
((fn [x1, x2, ..., xn] cos-del-let) expr1, expr2, ..., exprn)
```

Per quina raó NO ho és?

# *Closures*: El model d'entorns

Veiem un exemple no trivial d'aquesta equivalència entre el `let` i les crides a funcions. Recordeu la plana 9?

Transformem aquell `let` en crides a funcions:

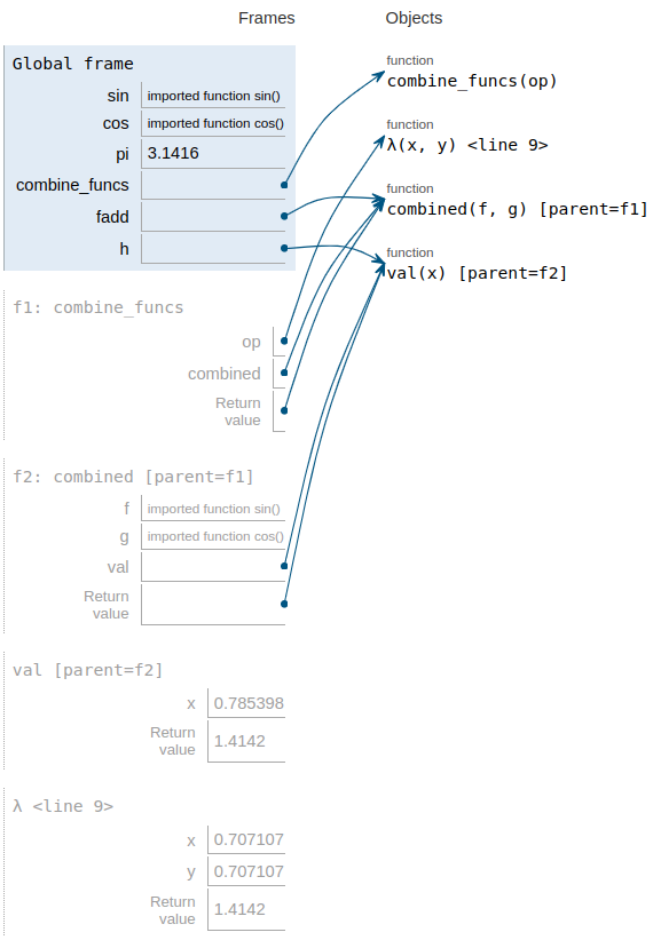
```
((fn [afegir1]
   ((fn [[cercar2,afegir2]]
      ((fn [[cercar3,afegir3]]
         ((fn [[cercar4,afegir4]]
            ((fn [[cercar5,afegir5]]
               (println (cercar5 :a))
               (println (cercar5 :c))
               (println (cercar5 :b))
               (println (cercar5 :d))
               (println (cercar3 :c)))
             (afegir4 :d 4)))
          (afegir3 :c 3)))
       (afegir2 :b 2)))
    (afegir1 :a 1)))
 (crear)) 👉 nil
👁 1
👁 3
👁 2
👁 4
👁 :default
```

# *Closures*: El model d'entorns

Veiem un exemple en Python, fent servir el Python Tutor:

```
from math import sin, cos, pi
def combine_funcs(op):
    def combined(f,g):
        def val(x):
            return op(f(x),g(x))
        return val
    return combined

fadd = combine_funcs(lambda x,y: x + y)
h    = fadd(sin, cos)
print(h(pi / 4))
```

```
(require '[clojure.math :as m])
(defn combine_funcs [op]
  (letfn [(combined [f g]
            (letfn [(val [x]
                      (op (f x) (g x)))]
              val))]
    combined))

(let [fadd (combine_funcs #(+ %1 %2))
      h    (fadd m/sin m/cos)]
  (println (h (/ m/PI 4))))
```



# *Closures*: Exercicis Recapitulatoris

- (**Z73720**) Implementeu la funció `aplicacio-condicional`, una funció que té dues funcions com a paràmetres, `f` i `condicio`. `f` és una funció que pren dos arguments, i `condicio` és una funció-predicat que acceptarà un sol argument i retornarà `true` o `false`. `aplicacio-condicional` retorna una funció `fr` tal que:

```
                                  |sí ---> retorna gr
fr(x) --és cert (condicio x)?---|
                                  |no ---> Ignorem x i retornem fr

                                  |sí ---> retorna (f x y)
gr(y) --és cert (condicio y)?---|
                                  |no ---> Ignorem y i retornem gr
```

  Així, podem passar tants arguments com calgui fins que n'hi hagi dos que satisfan condicio. Obviament el darrer argument que fem servir cal que satisfaci la condició; en altre cas intentaria aplicar un nombre com si fos una funció.

```
(def suma-si-parell (aplicacio-condicional + even?)) 👉 #'user/suma-si-parell
((suma-si-parell 2) 4)                               👉 6
(((((suma-si-parell 2) 3) 5) 7) 6)                   👉 8
```

# *Closures*: Exercicis Recapitulatoris

- Feu una funció `(maximitzador f)` que, donada una funció com a paràmetre (anomenem-la $f$; suposarem que aquesta funció $f$ rep un nombre i retorna un nombre), retorni una altra funció.
  Aquesta funció retornada ha de ser una funció tal que, després de les invocacions amb $n-1$ paràmetres, quan s'invoca amb l'$n$-èssim paràmetre escrigui al *stdout* el màxim dels $f(x_i)$ trobats fins el moment (és a dir, el màxim de $f(x_1),...,f(x_n)$ per a tots els $x_i$ amb que aquesta funció, retornada per `(maximitzador f)`, ha estat invocada).
  No podeu utilitzar cap llista, diccionari, conjunt o cap altra estructura de dades. Només nombres i funcions.

```
(def h (maximitzador #(* % %))) 👉 #'user/h
((((((h 2) 3) 2) 1) 5) 3)       👉 #object[user$maximitzador$...]
;; la crida ha retornat una representació interna de la funció retornada
👁 4
👁 9
👁 9
👁 9
👁 25
👁 25
```

# *Closures*: Exercicis Recapitulatoris

- Escriure una funció `ordre(ops)` que, donat un **vector** no buit `ops` de **funcions de dos paràmetres** retorna **una nova funció de dos paràmetres**. Aquesta funció retornada el que ha de fer és, quan és invocada, ha d'escriure al *stdout* el resultat d'aplicar la primera funció d'`ops` als arguments que ha rebut, després ha de retornar una funció que aplicarà la segona funció d'`ops` i així successivament. Quan la funció retornada hagi aplicat la darrera funció d'`ops`, caldrà que retorni una funció que torni a començar aplicant la primera funció d'`ops`.

```
(def funcions [+ * -])   👉 #'user/funcions ;; [suma, producte, resta]
(def f (ordre funcions)) 👉 #'user/f
(def f (f 1.0 2.0)) 👉 #'user/f
👁  3.0              ;; 1.0 + 2.0
(def f (f 1.0 2.0)) 👉 #'user/f
👁  2.0              ;; 1.0 * 2.0
(def f (f 1.0 2.0)) 👉 #'user/f
👁  -1.0             ;; 1.0 - 2.0
(def f (f 1.0 2.0)) 👉 #'user/f
👁  3.0              ;; 1.0 + 2.0
(def f (f 1.0 2.0)) 👉 #'user/f
👁  2.0              ;; 1.0 * 2.0
(def f (f 1.0 2.0)) 👉 #'user/f
👁  -1.0             ;; 1.0 - 2.0
;; etc...
```

# *Closures*: Exercicis Recapitulatoris

- Escriure una funció `aplica` que, donada una funció `f` d'un paràmetre numèric positiu, que retorna un nombre, retorna una altra funció que, donat un nombre positiu `x` aplica `f` a `x`, `(f x)`, el *guarda* i retorna una funció que repeteix el procés. Si aquesta funció rep com a argument `-1`, la funció acaba i retorna un vector amb el resultat d'haver aplicat `f` a tots els arguments que ha rebut aquesta funció (l'argument de la darrera crida cal que sigui `-1`, en altre cas retornarà una funció).

```
(require '[clojure.math :as m])
(def h (aplica m/sqrt)) 👉 #'user/h
((((((h 2) 4) 6) 8) 10) -1) 👉
[1.414213562373 2.0 2.44948974278 2.828427124746 3.162277660168]
(def h (aplica m/cos)) 👉 #'user/h
((((h (/ m/PI 4)) (/ m/PI 2)) m/PI) -1) 👉
[0.7071067811865476 6.123233995736766E-17 -1.0]
```

  No podeu fer servir llistes ni vectors (ni cap altre estructura de dades) més que passant-ho com a argument a una funció. No sabem encara treballar amb *closures* amb estat mutable, per tant, aquest problema no és tan senzill com pot semblar.

# *Closures*: Exercicis Recapitulatoris

- Escriure una funció `filtra` que, donada una funció-predicat `pred`, retorna una funció d'un paràmetre numèric positiu tal que *guarda* aquest si satisfà la funció-predicat `pred`, en altre cas l'ignora i retorna una funció d'un paràmetre que fa el mateix procés. Aquesta funció retornada, en rebre com a argument `-1`, s'aturarà escrivint al *stdout* un vector amb tots els arguments rebuts que han satisfet `pred` (l'argument de la darrera crida cal que sigui `-1`, en altre cas retornarà una funció) i *retornant* el *keyword* `:done`.

```
(def h (filtra even?)) 👉 #'user/h
((((((h 2) 4) 6) 8) 10) -1) 👉 :done
👁 [2 4 6 8 10]
((((((h 2) 3) 6) 5) 10) -1) 👉 :done
👁 [2 6 10]
```

  No podeu fer servir llistes ni vectors (ni cap altre estructura de dades) més que passant-ho com a paràmetre a una funció. No sabem encara treballar amb *closures* amb estat mutable, per tant, aquest problema no és tan senzill com pot semblar.

- Canvia la funció `filtra` per a que retorni el vector, enlloc d'escriure'l i retornar `:done`

# Closures

I si el context lèxic conté entitats ***mutables***?

***One of the conclusions that we reached was that the "object" need not be a primitive notion in a programming language; one can build objects and their behaviour from little more than assignable value cells and good old lambda expressions.***

Guy Steele, comentant el disseny d'Scheme, citat a Let Over Lambda

I sí, efectivament,

(...) *assignable value cells and good old lambda expressions* ➡️ *Closures*

# *Closures*: Estat mutable en el context lèxic

Fins ara hem vist *closures* que capturen un context lèxic format per noms de paràmetres de funcions i per noms definits per `let` i/o `letfn`. Aquests noms no admeten cap canvi en el seu lligam a un valor, per tant el context lèxic està format per *lligams (entre noms i valors) immutables*.

Clojure, però, no és un llenguatge funcional *pur*. Així, és possible que una *closure* capturi i manipuli entitats ***mutables***. A Clojure, però, cal fer-ho ***explícitament***. Clojure té varies maneres de manipular entitats mutables fent servir `vars`, `atoms`, `refs` o `agents`*, anomenats *reference types*.

Ja hem treballat amb `vars`. Les creem amb `def` o `defn`. La seva consideració com a *reference type* té a veure amb que poden tenir abast dinàmic (*dynamic scope*) i canviar de valor en diferents fils d'execució. No tractarem aquest aspecte de les `vars` en aquest curs. Per a nosaltres són essencialment la manera de definir lligams globals, tot i que la seva utilitat real va força més enllà. `refs` i `agents` tenen un paper molt important a l'hora de gestionar la concurrència a Clojure, tema que nosaltres no tractem.

El mecanisme que nosaltres farem servir per treballar amb estat mutable seran els `atoms`. Així podrem tenir les *assignable value cells* que menciona la cita de Guy Steele a la transparència anterior.

* Vegeu aquest article o el capítol 10 de *The Joy of Clojure*

# *Closures*: Atoms

Els `atoms`* a Clojure són objectes que "*contenen*" *referències* a valors. Quan vinculem un nom a un `atom` estem creant un lligam immutable entre el nom i aquest objecte. La diferència ara és que *podem accedir al valor referenciat* i *modificar aquesta referència*. Fixem-nos que això no afecta la immutabilitat del lligam entre el nom i l'objecte que "*conté*" aquesta referència.

Les funcions principals que farem servir són `atom`, la *read-macro* `@` (alternativa a `deref`), `swap!`, `reset!`.

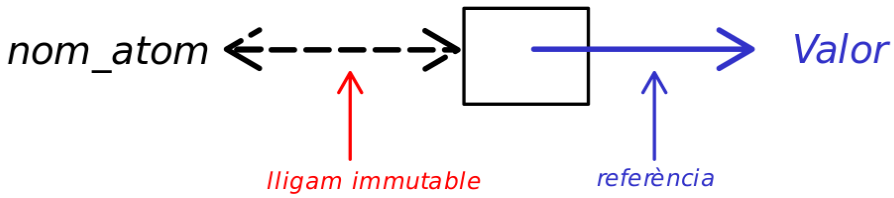
```
(def nom_atom (atom 0)) 👉 #'user/nom_atom

(deref nom_atom ) 👉 0
@nom_atom         👉 0

(swap! nom_atom #(+ % 10)) 👉 10
@nom_atom                  👉 10

(reset! nom_atom 5) 👉 5
@nom_atom           👉 5
```

Els `atoms` poden tenir *validators* (funcions associades que controlen que els valors referenciats compleixin uns determinats requeriments), però no en farem gaire ús.

* Vegeu la secció 10.4 de *The Joy of Clojure*

# *Closures*: Atoms

Gràficament:




- `(atom valor)` - crea un `atom` amb una referència al `valor` proporcionat
- `(deref nom_atom)` - retorna el valor referenciat per l'objecte vinculat a `nom_atom` (`@nom_atom` és el mateix que `(deref nom_atom)`)
- `(reset! nom_atom valor)` - fa que `valor` sigui referenciat per l'objecte vinculat a `nom_atom`
- `(swap! nom_atom f x y & args)` - `f` ha de ser *pura*. Retorna el nou valor.

| versió de `swap!` | fa que el nou valor de `nom_atom` sigui |
|---|---|
| `(swap! nom_atom f)` | `(apply f ( @nom_atom ))` |
| `(swap! nom_atom f x)` | `(apply f @nom_atom ( x ))` |
| `(swap! nom_atom f x y)` | `(apply f @nom_atom x ( y ))` |
| `(swap! nom_atom f x y & args)` | `(apply f @nom_atom x y args)` |

# *Closures*: Objectes

Tornem a les *closures*...

En el moment que podem tenir `atoms` com a part del context lèxic capturat per una *closure* ja podem tenir un "*estat intern*" mutable del que treure'n profit:

```
(def comptador-funcs (let [comptador (atom 0)] [#(identity @comptador)
                                                #(reset! comptador 0)
                                                #(swap! comptador + %)
                                                #(swap! comptador - %)]))
(def consulta   (get comptador-funcs 0)) 👉 #'user/consulta
(def reset      (get comptador-funcs 1)) 👉 #'user/reset
(def incrementa (get comptador-funcs 2)) 👉 #'user/incrementa
(def decrementa (get comptador-funcs 3)) 👉 #'user/decrementa

(reset)         👉 0
(incrementa 3)  👉 3
(incrementa 3)  👉 6
(incrementa 5)  👉 11
(consulta)      👉 11
(decrementa 9)  👉 2
(consulta)      👉 2
```

Vinculem a `comptador-funcs` un vector amb quatre *closures*. Les quatre *closures* han capturat el **mateix** context lèxic, en aquest cas l'`atom` anomenat `comptador`.

# *Closures*: Objectes

**Exercici**: Donades aquestes dues funcions, expliqueu per què s'observa el que s'observa en executar-les:

```clojure
(defn codi-a []
 (let [res (atom [])
       estat (atom 0)]
   (loop [i 1]
     (if (> i 3)  @res
       (do
         (swap! estat (fn [_] i))
         (swap! res
          #(conj %
             (fn [] (println @estat))))
         (recur (inc i)))))))
```

```clojure
(defn codi-b []
 (let [res (atom [])]
   (loop [i 1]
     (let [estat (atom 0)]
       (if (> i 3)  @res
         (do
           (swap! estat (fn [_] i))
           (swap! res
            #(conj %
               (fn [] (println @estat))))
           (recur (inc i))))))))
```

```clojure
(def c-a (codi-a)) 👉  #'user/c-a

;; Executem c-a...

((c-a 0)) 👉 nil
👁  3
((c-a 1)) 👉 nil
👁  3
((c-a 2)) 👉 nil
👁  3
```

```clojure
(def c-b (codi-b)) 👉  #'user/c-b

...i c-b

((c-b 0)) 👉 nil
👁  1
((c-b 1)) 👉 nil
👁  2
((c-b 2)) 👉 nil
👁  3
```

# *Closures*: Estructures de dades

Que diverses *closures* puguin capturar i compartir entitats mutables és una idea molt potent. De fet, res ens allunya del concepte clàssic d'*objecte*, no en el sentit d'*instància d'una classe*, sinò en el sentit d'un *estat* mutable i ***privat*** al que només es pot accedir via uns "*mètodes*" (les *closures* que comparteixen aquest *estat*).

Veiem-ne un exemple:

```
(defn generador_piles
  []
  (let [ed  (let [pila (atom []),
                  mida (atom 0)]
              {:reset    #(do (swap! pila (fn [_] [])) (swap! mida (fn [_] 0))),
               :push     #(do (swap! pila conj %) (swap! mida inc)),
               :top      #(when (> @mida 0) (peek @pila)),
               :pop      #(when (> @mida 0)
                            (let [r (peek @pila)]
                              (swap! pila pop) (swap! mida dec) r)),
               :empty?   #(= @mida 0),
               :size     #(identity @mida)
               })]
    (fn [& cmd]
      (let [instr (keyword (first cmd)),
            argum (rest cmd)
            f     (instr ed)]
        (apply f argum)))))
```

# *Closures*: Estructures de dades

`generador_piles` retorna una funció per manipular un `map` de parelles *keyword-closure*, que alhora capturen i comparteixen dos `atoms`, amb valors un vector i un enter, per tal de manipular-los amb les operacions que caracteritzen una pila.

```
(def p (generador_piles)) 👉 #'user/p
(p 'reset) 👉 0

(p 'push 2) 👉 1
(p 'push \t) 👉 2
(p 'push "hello") 👉 3

(p 'size) 👉 3
(p 'empty?) 👉 false

(p 'top) 👉 "hello"
(p 'pop) 👉 "hello"

(p 'top) 👉 \t
(p 'pop) 👉 \t

(p 'top) 👉 2
(p 'pop) 👉 2

(p 'empty?) 👉 true
```

# *Closures*: Iteradors

La mateixa idea ens permet crear el que en altres llenguatges s'anomenen ***Iteradors*** sobre seqüències:

```
(defn creaIterador [s]
  "Creem un iterador sobre la seqüència s"
  (let [vct   (vec s)   ;; converteixo s en un vector, per millorar l'eficiència
        index (atom 0)]
    (fn [cmd]
      (if (= cmd :reset)   ;; si :reset no importa en quin estat està l'iterador
        (reset! index 0)
        (let [elem   (get vct @index)]
          (if (nil? elem)
            (throw (Exception. "StopIteration"))  ;; si acabo d'iterar, excepció
            (cond
              (= cmd :last?) (= (count vct) (inc @index))
              (= cmd :peek)  elem
              (= cmd :next)  (do (swap! index inc) elem))))))))

(def it (creaIterador '(10 20 30))) 👉 #'user/it
(it :peek) 👉 10
(it :last?)👉 false
(it :next) 👉 10
(it :next) 👉 20
(it :last?)👉 true
(it :next) 👉 30
(it :next) 👉 Execution error at user/creaIterador$fn (REPL:9).
              StopIteration
```

# *Closures*, exemples: Composició de funcions

Tot i que tenim `comp` a Clojure, tenint *closures* en realitat no ens cal.

Veiem un exemple senzill, on fem una funció `my-comp` per composar funcions d'un paràmetre tals que és possible fer-ne la composició (els dominis i els rangs corresponents són els *adequats*):

```
(defn my-comp [& funcs]
  "funcs: llista de funcions d'un paràmetre tals que 'té sentit' composar-les"
  (fn [arg]
      "arg ha de pertànyer al domini de la darrera funció de funcs"
      (fold #(%1 %2) arg funcs)))  ;; recordeu el fold?

(def foo (my-comp reverse sort (partial map #(* 2 %)) (partial filter even?)))
👉 #'user/foo
(foo (range 20)) 👉 (36 32 28 24 20 16 12 8 4 0)

((my-comp reverse) (range 10)) 👉 (9 8 7 6 5 4 3 2 1 0)

((my-comp) (range 20)) 👉 (0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19)
```

Òbviament un `my-comp` general requeriria una implementació més sofisticada.

# *Closures*, exemples: Decoradors

Amb *Closures* podem escriure el que s'anomena en Python **decoradors**, que són funcions per modificar d'alguna manera el context en el que s'executa una funció.

Per exemple, podem afegir el càlcul del temps d'execució a qualsevol funció:

```
(defn timer-decorate [f]
  (fn [& args]
    (time (apply f args))))

(def tmap (timer-decorate map)) 👉 #'user/tmap
(tmap inc (range 10)) 👉 (1 2 3 4 5 6 7 8 9 10)
👁️ "Elapsed time: 0.052528 msecs"

(def tfilter (timer-decorate filter)) 👉 #'user/tfilter
(tfilter even? (range 20)) 👉 (0 2 4 6 8 10 12 14 16 18)
👁️ "Elapsed time: 0.012717 msecs"

(def treduce (timer-decorate reduce)) 👉 #'user/treduce
(treduce       ;; primers 15 nombres de Fibonacci
  (fn [a b] (conj a (+' (last a) (last (butlast a)))))
  [0 1]
  (repeat 13 1)) 👉 [0 1 1 2 3 5 8 13 21 34 55 89 144 233 377]
👁️ "Elapsed time: 0.064126 msecs"
```

# *Closures*, exemples: Decoradors

També podem fer que abans i després de la crida a la funció es realitzin determinades accions, com per exemple escriure o guardar informació rellevant que tingui a veure amb els arguments. Un exemple simplificat on només escrivim un missatge pot ser:

```
(defn logging-decorate [f]
  (fn [& args]
    (println "...comunicar/guardar info abans de la crida a la funció...")
    (let [resultat (apply f args)]
      (println "...comunicar/guardar info després de la crida a la funció...")
      resultat)))

(def lmap (logging-decorate map)) 👉 #'user/lmap
(lmap inc (range 10)) 👉 (1 2 3 4 5 6 7 8 9 10)
👁️ ...comunicar/guardar info abans de la crida a la funció...
👁️ ...comunicar/guardar info després de la crida a la funció...

(def lfilter (logging-decorate filter)) 👉 #'user/lfilter
(lfilter even? (range 10)) 👉 (0 2 4 6 8)
👁️ ...comunicar/guardar info abans de la crida a la funció...
👁️ ...comunicar/guardar info després de la crida a la funció...
```

La possibilitat de guardar informació sobre l'execució d'una funció, en particular de *memoritzar* els resultats per a no haver-los de tornar a calcular (això que en anglés s'anomena *memoize*) s'implementa també fent servir un decorador. Que entengueu el decorador que *memo(r)itza* és un exercici de la sessió de laboratori dedicada a les *Closures*.

# *Closures*: Exercici Recapitulatori

Hi ha una manera interessant d'implementar una cua, que és mitjançant dues llistes (*llistes* perquè és poc costós afegir elements al davant de la llista amb `cons`). Una llista conté part de la cua, i l'altra conté el que resta de la cua, però *invertit*.

```
;; No Clojure => Cua[:a :b :c :d :e] = ((:a :b) (:e :d :c))
```

Feu una funció `generador_cues`, amb les operacions habituals (`reset`, `push`, `top`, `pop`, `empty?`, `equal` i una que ens ensenyi el contingut de la cua, `contents`), que implementi una cua d'aquesta manera fent servir *closures*.

Un joc de proves podria ser:

```
(def q (generador_cues)) 👉 #'user/q
(q 'reset) 👉 ()
(q 'push 2) 👉 nil
(q 'push 20) 👉 nil
(q 'push 200) 👉 nil
(q 'contents) 👉 (2 20 200)
(q 'empty?) 👉 false
(q 'pop) 👉 2
(q 'contents) 👉 (20 200)
(q 'top) 👉 20
(q 'contents) 👉 (20 200)
```

# *Closures*

Teniu més exemples de *Closures* en els exercicis de laboratori.

Fins aquí la introducció a les ***funcions d'ordre superior***, on hem tractat les possibilitats que ofereix el fet de poder passar funcions com a paràmetre d'altres funcions (Funcions *first-class*), i el fet de poder retornar funcions des d'altres funcions (*Closures*).

En el proper tema del curs veurem algunes de les tècniques que podem fer servir gràcies al fet de tenir ***funcions d'ordre superior***.