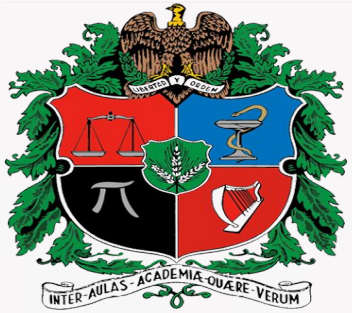


Clojure

Juan Camilo Calero

Carlos Orlando Solórzano

Oscar Dario Parra

Abril de 2016

QUE ES CLOJURE?

Clojure es un lenguaje de programación que hace énfasis en el paradigma de programación funcional. Es un lenguaje de propósito general dialecto de Lisp. Clojure puede ser ejecutado sobre la Máquina Virtual de Java y la máquina virtual de la plataforma .NET, así como compilado a JavaScript.

Características principales:

- Desarrollo dinámico con una consola de evaluación (en inglés, REPL: read eval print loop).
- Sistema integrado de estructuras de datos persistentes e inmutables.
- Interacción con java: al compilarse a bytecode de la JVM.

INSTALACIÓN.

- Ingresamos a la siguiente URL:

http://www.clojure.org/guides/getting_started

- Descargamos *clojure.jar*
-

```
java -cp clojure-1.8.0.jar clojure.main
```

-

```
user=> (+ 1 2 3)
6
user=> (javax.swing.JOptionPane/showMessageDialog nil "Hello World")
```

CLOJURE

Sintaxis

La sintaxis de Clojure es simple. Al igual que todos Lisps, emplea una estructura uniforme, un puñado de operadores especiales, y un suministro constante de paréntesis.

- Formas.
- Control de flujo.
- Nombrar valores con *def.*

Formas

Clojure reconoce dos tipos de Estructuras.

- Representaciones literales de las estructuras de datos (como números, cadenas, mapas, y vectores).

```
1
"a string"
["a" "vector" "of" "strings"]
```

- Operaciones

```
( Operador  operando1  operando2  ...  operandn )
```

Ejemplo de operaciones

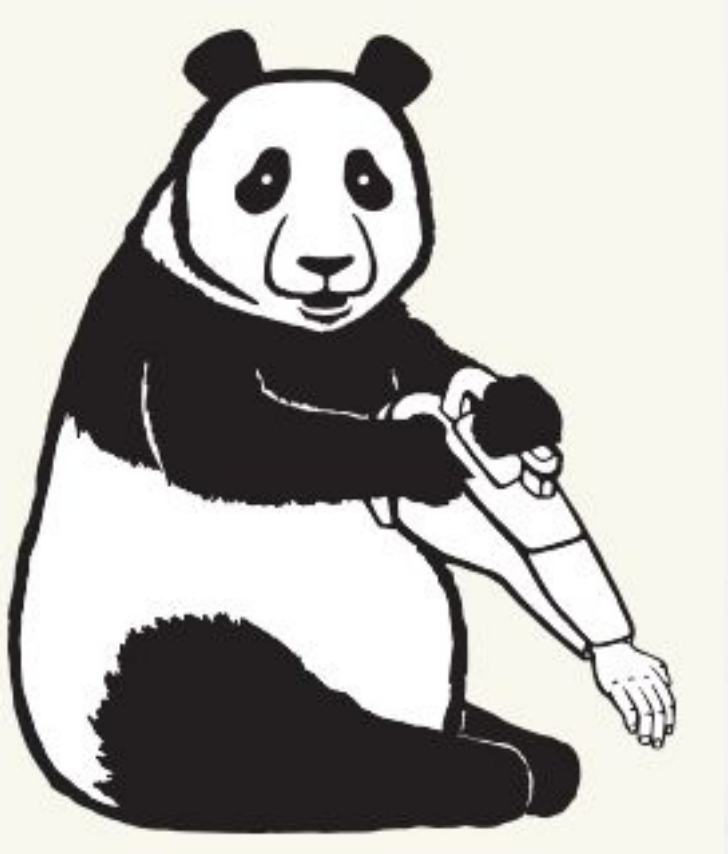
```
(+ 1 2 3)
; => 6

(str "It was the panda " "in the library " "with a dust buster")
; => "It was the panda in the library with a dust buster"
```

La siguiente no es una forma válida.

```
(+
```

No importa qué operador está utilizando o qué tipo de datos que está en funcionamiento, la estructura es la misma.



Control de flujo

Veamos tres operadores de control de flujo básicos: ***if, do*** y ***when.***

if.

```
(if boolean-form
  then-form
  optional-else-form)
```

do.

Permite envolver múltiples formas de paréntesis, y ejecutar cada una de ellas.

```
(if true
  (do (println "Success!")
      "By Zeus's hammer!")
  (do (println "Failure!")
      "By Aquaman's trident!"))
; => Success!
; => "By Zeus's hammer!"
```

when.

El operador ***when*** es una combinación de ***if, do,*** pero no ***else.***

```
(when true
  (println "Success!")
  "abra cadabra")
; => Success!
; => "abra cadabra"
```

nil, true, false, Truthiness, igualdad y Expresiones boolenas.

nil se utiliza para indicar ningún valor.

```
(nil? 1)
; => false

(nil? nil)
; => true
```

Truthy and falsey

Truthy y falsey se refiere a cómo un valor se trata en una expresión booleana.

```
(if "bears eat beets"
  "bears beets Battlestar Galactica")
; => "bears beets Battlestar Galactica"

(if nil
  "This won't be the result because nil is falsey"
  "nil is falsey")
; => "nil is falsey"
```

- El operador de igualdad de clojure es `=` .

- Clojure usa los operadores booleanos **or** y **and.**

```
(or false nil :large_I_mean_venti :why_cant_I_just_say_large)
; => :large_I_mean_venti

(or (= 0 1) (= "yes" "no"))
; => false

(or nil)
; => nil
```

```
(and :free_wifi :hot_coffee)
; => :hot_coffee

(and :feelin_super_cool nil false)
; => nil
```

Nombrar valores con *def*.

Se usa ***def*** *para* ***enlazar*** *un nombre con un valor.*

```
(def failed-protagonist-names
  ["Larry Potter" "Doreen the Explorer" "The Incredible Bulk"])

failed-protagonist-names
; => ["Larry Potter" "Doreen the Explorer" "The Incredible Bulk"]
```

Un ejemplo en Ruby.

```ruby
severity = :mild
error_message = "OH GOD! IT'S A DISASTER! WE'RE "
if severity == :mild
  error_message = error_message + "MILDLY INCONVENIENCED!"
else
  error_message = error_message + "DOOOOOOOMED!"
end
```

Estamos tentados a hacer esto en clojure.

```clojure
(def severity :mild)
(def error-message "OH GOD! IT'S A DISASTER! WE'RE ")
(if (= severity :mild)
  (def error-message (str error-message "MILDLY INCONVENIENCED!"))
  (def error-message (str error-message "DOOOOOOOMED!")))
```

```clojure
(def severity :mild)
(def error-message "OH GOD! IT'S A DISASTER! WE'RE ")
(if (= severity :mild)
  (def error-message (str error-message "MILDLY INCONVENIENCED!"))
  (def error-message (str error-message "DOOOOOOOMED!")))
```

Cambiar el valor asociado con nombre, puede que sea más difícil de entender el comportamiento programa. Clojure tiene un conjunto de herramientas para hacer frente al cambio.

```clojure
(defn error-message
  [severity]
  (str "OH GOD! IT'S A DISASTER! WE'RE "
       (if (= severity :mild)
         "MILDLY INCONVENIENCED!"
         "DOOOOOOOMED!")))

(error-message :mild)
; => "OH GOD! IT'S A DISASTER! WE'RE MILDLY INCONVENIENCED!"
```

Estructuras de datos

Todas las estructuras de datos en clojure son inmutables.

Numbers

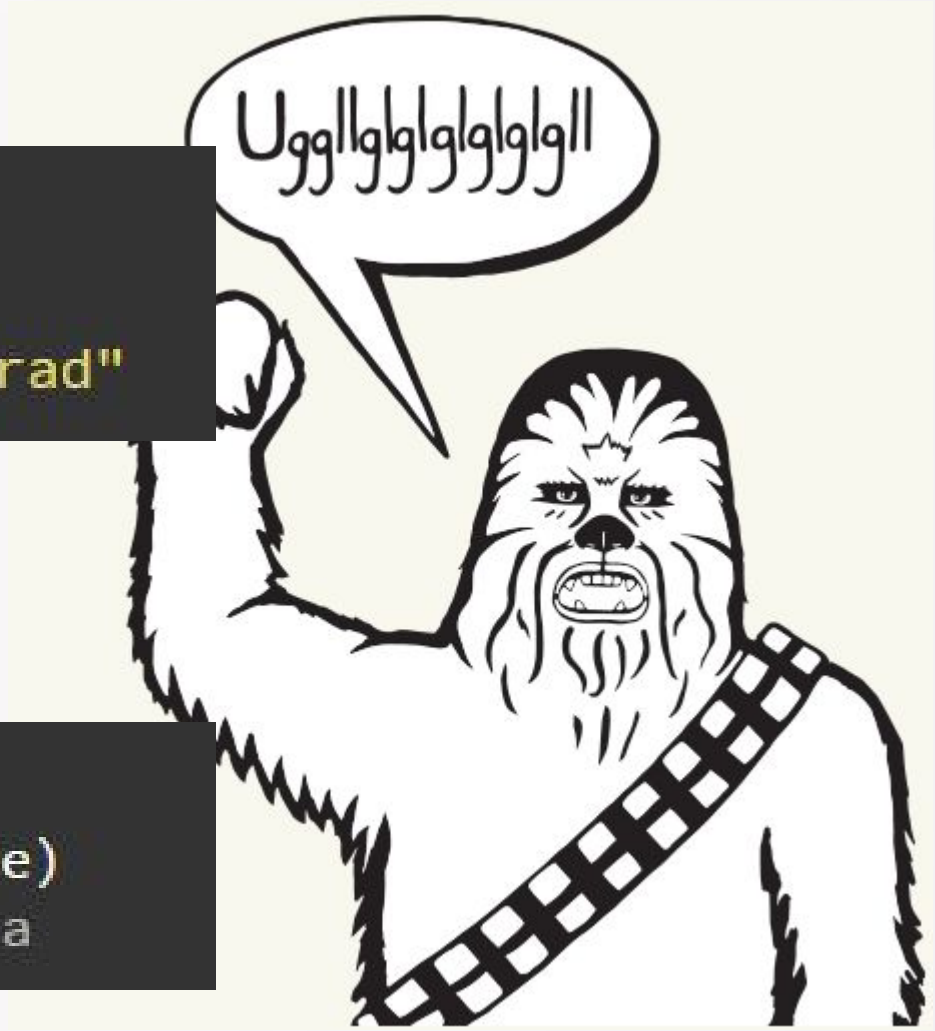
Aquí es un número **integer,** un **float** y un **ratio**, respectivamente:

```
93
1.2
1/5
```

String

```
"Lord Voldemort"
"\"He who must not be named\""
"\"Great cow of Moscow!\" - Hermes Conrad"
```

```
(def name "Chewbacca")
(str "\"Uggllglglglglglglglll\" - " name)
; => "Uggllglglglglglglglll" - Chewbacca
```



Maps

Son similares a los diccionarios a otros lenguajes. Es una manera de asociar valores con otros valores.

```
{}
```

```
{:first-name "Charlie"
 :last-name "McFishwich"}
```

```
{"string-key" +}
```

Podemos usar **hash-map** para crear **maps.**

```
(hash-map :a 1 :b 2)
; => {:a 1 :b 2}
```

```
(get {:a 0 :b 1} :b)
; => 1

(get {:a 0 :b {:c "ho hum"}} :b)
; => {:c "ho hum"}
```

```
(get-in {:a 0 :b {:c "ho hum"}} [:b :c])
; => "ho hum"
```

Keywords

```
:a
:rumplestiltsken
:34
:_?
```

```
(:a {:a 1 :b 2 :c 3})
; => 1
```

This is equivalent to:

```
(get {:a 1 :b 2 :c 3} :a)
; => 1
```

```
(:d {:a 1 :b 2 :c 3} "No gnome knows homes like Noah knows")
; => "No gnome knows homes like Noah knows"
```

Vectors

```
[3 2 1]
```

```
(get [3 2 1] 0)
; => 3
```

```
(vector "creepy" "full" "moon")
; => ["creepy" "full" "moon"]
```

```
(conj [1 2 3] 4)
; => [1 2 3 4]
```

Lists

```
'(1 2 3 4)
; => (1 2 3 4)
```

```
(nth '(:a :b :c) 0)
; => :a

(nth '(:a :b :c) 2)
; => :c
```

```
(list 1 "two" {3 4})
; => (1 "two" {3 4})
```

Sets

```
#{"kurt vonnegut" 20 :icicle}
```

```
(hash-set 1 1 2 2)
; => #{1 2}
```

```
(conj #{:a :b} :b)
; => #{:a :b}
```

Funciones

Llamada de funciones

```
(+ 1 2 3 4)
(* 1 2 3 4)
(first [1 2 3 4])
```

```
((or + -) 1 2 3)
;; => 6
```

Definición de funciones

```
1 (defn too-enthusiastic
2   "Return a cheer that might be a bit too enthusiastic"
3   [name]
4   (str "OH. MY. GOD! " name " YOU ARE MOST DEFINITELY LIKE THE BEST "
  "MAN SLASH WOMAN EVER I LOVE YOU AND WE SHOULD RUN AWAY SOMEWHERE"))

(too-enthusiastic "Zelda")
; => "OH. MY. GOD! Zelda YOU ARE MOST DEFINITELY LIKE THE BEST MAN SLASH WOMAN EVE
```

Las funciones también soportan *arity overloading*

```clojure
(defn multi-arity
  ;; 3-arity arguments and body
  ([first-arg second-arg third-arg]
     (do-things first-arg second-arg third-arg))
  ;; 2-arity arguments and body
  ([first-arg second-arg]
     (do-things first-arg second-arg))
  ;; 1-arity arguments and body
  ([first-arg]
     (do-things first-arg)))
```

Las funciones también soportan *arity overloading*

```
(defn x-chop
  "Describe the kind of chop you're inflicting on someone"
  ([name chop-type]
     (str "I " chop-type " chop " name "! Take that!"))
  ([name]
     (x-chop name "karate")))
```

```
(x-chop "Kanye West" "slap")
; => "I slap chop Kanye West! Take that!"
```

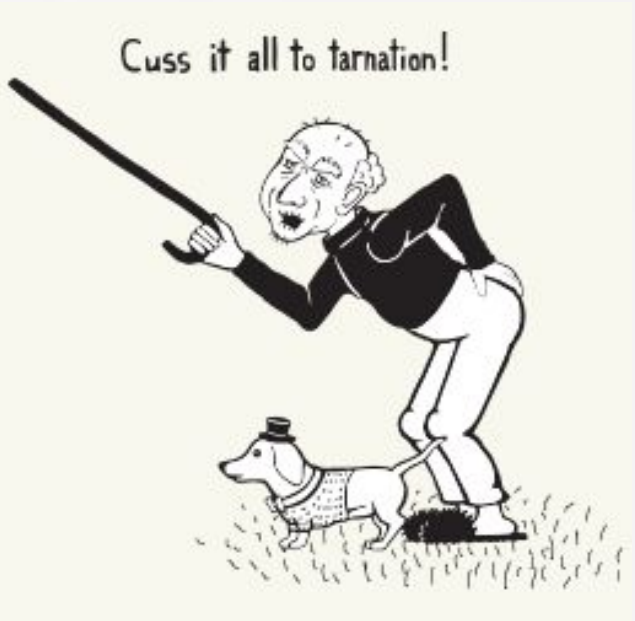
```
(x-chop "Kanye East")
; => "I karate chop Kanye East! Take that!"
```

rest parameter

```clojure
(defn codger-communication
  [whippersnapper]
  (str "Get off my lawn, " whippersnapper "!!!"))

(defn codger
  [& whippersnappers]
  (map codger-communication whippersnappers))

(codger "Billy" "Anne-Marie" "The Incredible Bulk")
; => ("Get off my lawn, Billy!!!"
      "Get off my lawn, Anne-Marie!!!"
      "Get off my lawn, The Incredible Bulk!!!")
```




```clojure
(defn favorite-things
  [name & things]
  (str "Hi, " name ", here are my favorite things: "
       (clojure.string/join ", " things)))

(favorite-things "Doreen" "gum" "shoes" "kara-te")
; => "Hi, Doreen, here are my favorite things: gum, shoes, kara-te"
```

Destructuring

Permite enlazar de forma concisa nombres a los valores dentro de una colección.
Ejemplo básico.

```
;; Return the first element of a collection
(defn my-first
  [[first-thing]] ; Notice that first-thing is within a vector
  first-thing)

(my-first ["oven" "bike" "war-axe"])
; => "oven"
```

También podemos desestructurar ***maps.***

```clojure
(defn announce-treasure-location
  [{lat :lat lng :lng}]
  (println (str "Treasure lat: " lat))
  (println (str "Treasure lng: " lng)))

(announce-treasure-location {:lat 28.22 :lng 81.33})
; => Treasure lat: 100
; => Treasure lng: 50
```

Cuerpo de la Función

El cuerpo de la función puede contener formas de cualquier tipo. Clojure automáticamente retorna la última forma evaluada.

```
(defn number-comment
  [x]
  (if (> x 6)
    "Oh my gosh! What a big number!"
    "That number's OK, I guess"))

(number-comment 5)
; => "That number's OK, I guess"

(number-comment 7)
; => "Oh my gosh! What a big number!"
```

Funciones anónimas

El clojure las funciones no necesitan tener nombre.

```
(fn [param-list]
  function body)
```

```
#(* % 3)
```

```
(#(str %1 " and " %2) "cornbread" "butter beans")
; => "cornbread and butter beans"
```

Retornando funciones

Las funciones retornadas son clausuras, significa que pueden acceder a todas las variables que resultaron en su alcance cuando se creó la función.

```
(defn inc-maker
  "Create a custom incrementor"
  [inc-by]
  #(+ % inc-by))

(def inc3 (inc-maker 3))

(inc3 7)
; => 10
```

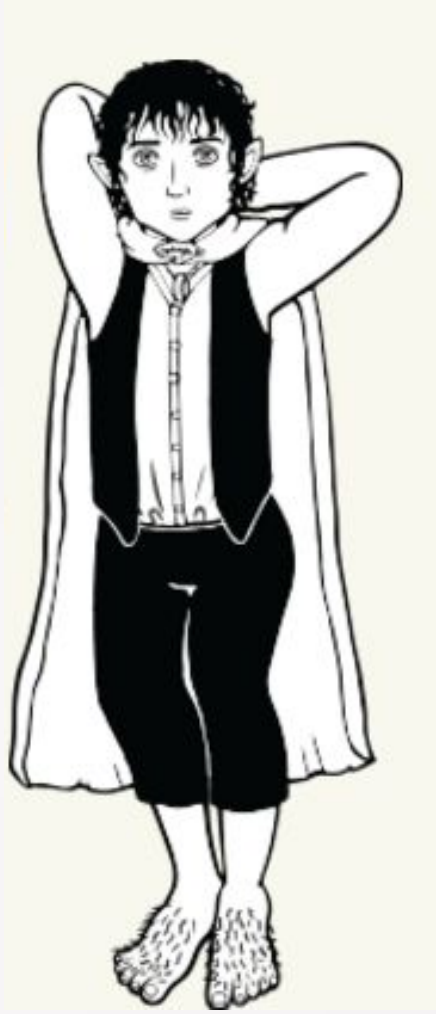
Uniendo Todo

Vamos a completar al hobbit.

El modelo ideal.

Nos vamos a centrar en las características físicas del hobbit.

```
(def asym-hobbit-body-parts [{:name "head" :size 3}
                             {:name "left-eye" :size 1}
                             {:name "left-ear" :size 1}
                             {:name "mouth" :size 1}
                             {:name "nose" :size 1}
                             {:name "neck" :size 2}
                             {:name "left-shoulder" :size 3}
                             {:name "left-upper-arm" :size 3}
                             {:name "chest" :size 10}
                             {:name "back" :size 10}
                             {:name "left-forearm" :size 3}
                             {:name "abdomen" :size 6}
                             {:name "left-kidney" :size 1}
                             {:name "left-hand" :size 2}
                             {:name "left-knee" :size 2}
                             {:name "left-thigh" :size 4}
                             {:name "left-lower-leg" :size 3}
                             {:name "left-achilles" :size 1}
                             {:name "left-foot" :size 2}])
```



The matching-part and symmetrize-body-parts functions

```clojure
(defn matching-part
  [part]
  {:name (clojure.string/replace (:name part) #"^left-" "right-")
   :size (:size part)})

(defn symmetrize-body-parts
  "Expects a seq of maps that have a :name and :size"
  [asym-body-parts]
  (loop [remaining-asym-parts asym-body-parts
         final-body-parts []]
    (if (empty? remaining-asym-parts)
      final-body-parts
      (let [[part & remaining] remaining-asym-parts]
        (recur remaining
               (into final-body-parts
                     (set [part (matching-part part)])))))))
```

```clojure
(symmetrize-body-parts asym-hobbit-body-parts)
; => [{:name "head", :size 3}
      {:name "left-eye", :size 1}
      {:name "right-eye", :size 1}
      {:name "left-ear", :size 1}
      {:name "right-ear", :size 1}
      {:name "mouth", :size 1}
      {:name "nose", :size 1}
      {:name "neck", :size 2}
      {:name "left-shoulder", :size 3}
      {:name "right-shoulder", :size 3}
      {:name "left-upper-arm", :size 3}
      {:name "right-upper-arm", :size 3}
      {:name "chest", :size 10}
      {:name "back", :size 10}
      {:name "left-forearm", :size 3}
      {:name "right-forearm", :size 3}
      {:name "abdomen", :size 6}
      {:name "left-kidney", :size 1}
      {:name "right-kidney", :size 1}
      {:name "left-hand", :size 2}
      {:name "right-hand", :size 2}
      {:name "left-knee", :size 2}
      {:name "right-knee", :size 2}
      {:name "left-thigh", :size 4}
      {:name "right-thigh", :size 4}
      {:name "left-lower-leg", :size 3}
      {:name "right-lower-leg", :size 3}
      {:name "left-achilles", :size 1}
      {:name "right-achilles", :size 1}
      {:name "left-foot", :size 2}
      {:name "right-foot", :size 2}]
```

Let

Se usa let para crear un binding (asociación) temporal

```
(let [a 1 b 2]
  (> a b)) ; => false
```

```
(let [x 3]
  x)
; => 3

(def dalmatian-list
  ["Pongo" "Perdita" "Puppy 1" "Puppy 2"])
(let [dalmatians (take 2 dalmatian-list)]
  dalmatians)
; => ("Pongo" "Perdita")
```

Expresiones Regulares

```
#"regular-expression"
```

```
(re-find #"^left-" "left-eye")
; => "left-"

(re-find #"^left-" "cleft-chin")
; => nil

(re-find #"^left-" "wongleblart")
; => nil
```

Loop

Bucle con recurrencia

```
(loop [iteration 0]
  (println (str "Iteration " iteration))
  (if (> iteration 3)
    (println "Goodbye!")
    (recur (inc iteration))))
; => Iteration 0
; => Iteration 1
; => Iteration 2
; => Iteration 3
; => Iteration 4
; => Goodbye!
```

Explicación del método

```
❶ (defn symmetrize-body-parts
  "Expects a seq of maps that have a :name and :size"
  [asym-body-parts]
❷   (loop [remaining-asym-parts asym-body-parts
           final-body-parts []]
❸     (if (empty? remaining-asym-parts)
       final-body-parts
❹       (let [[part & remaining] remaining-asym-parts]
❺         (recur remaining
                (into final-body-parts
                      (set [part (matching-part part)])))))))
```

Reduce

Procesa cada elemento en la secuencia y construye un resultado

```
;; sum with reduce
(reduce + [1 2 3 4])
; => 10
```

```
(+ (+ (+ 1 2) 3) 4)
```

```
(reduce + 15 [1 2 3 4])
```

Bibliografía

- Clojure. http://clojure.org/guides
- Clojure for the brave and true http://www.braveclojure.com/do-things/