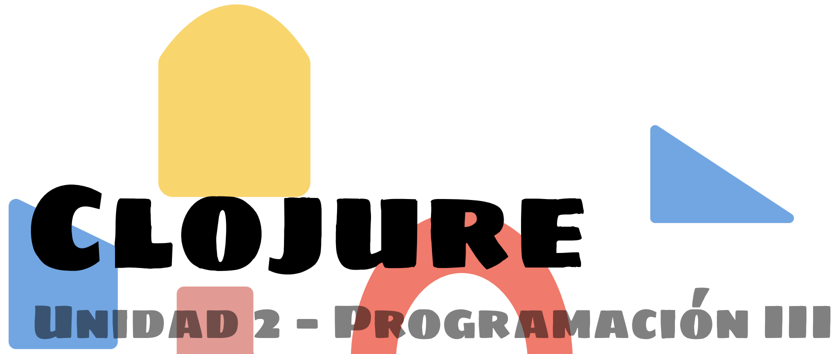


Clojure

Unidad 2 - Programación III

INSPT - UTN - Ciclo Lectivo 2026

Rich Hickey

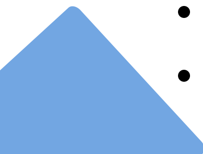
Programador y conferencista estadounidense.

- Creador de **Clojure** (2007)
- También creó **Datomic** y **ClojureScript**
- Background en C++, Java y C#
- CTO de Cognitect (2013–2020)
- Filosofía centrada en la **simplicidad** y la gestión del **estado**

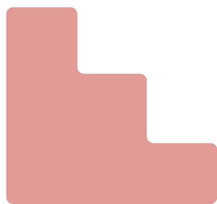
"Programmers know the benefits of everything and the tradeoffs of nothing."

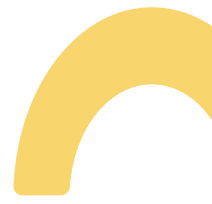
Fundamentos

Lenguaje



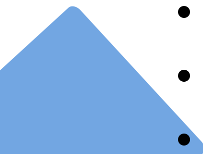
- Programación funcional (no puro)
- Dialecto moderno de **Lisp**
- Compila a bytecode **JVM**
- Concurrencia segura — modelo de **STM**
- Interop nativa con **Java**

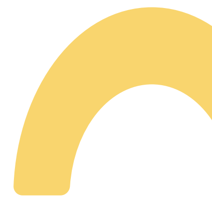




Características

- **Homoicónico** — el código es una estructura de datos
- Datos persistentes — **Inmutables**
- Recursividad — Funciones de orden superior
- **Lazy evaluation**
- Entornos Java — JVM
- **REPL**: read-eval-print-loop



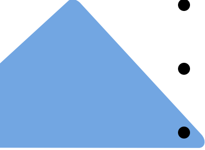


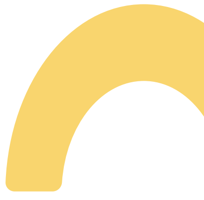
Ejemplos - REPL

```
user⇒ (doc list)                ; Muestra la documentación de list
user⇒ (find-doc "trim")         ; Muestra las documentaciones que incluyan trim
user⇒ (apropos "?")             ; Muestra los nombres que incluyan ?
user⇒ (dir clojure.string)      ; Muestra los nombres definidos en clojure.string
user⇒ (source +)                ; Muestra la definición (código fuente) de +
user⇒ (load-file "op.clj")     ; Evalúa secuencialmente el contenido de op.clj
user⇒ (clojure-version)         ; Muestra la versión de Clojure
```

Elementos

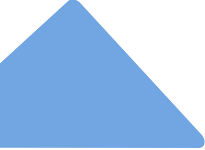
- Datos
- Evaluación de expresiones
- Formas especiales
- Funciones predefinidas
- Funciones de orden superior
- Macros predefinidas

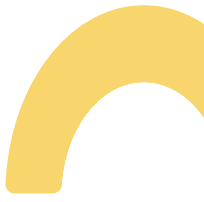




Datos - Clasificación

- Un **escalar** (letras, cadenas, números), menos las palabras reservadas
- Una **colección** de datos
- Una **secuencia** (abstracción de una colección)





Escalares - Símbolos

- Nombre de: función, parámetro, variable, etc.
- Case sensitive
- Siempre comienzan con carácter
- `'` para evitar que se evalúe

```
user⇒ a
CompilerException java.lang.RuntimeException:
  Unable to resolve symbol: a in this context

user⇒ 'a
a
```

Escalares - Valores literales

Números: `42` · `42.5` · `1/3`

Caracteres: `\@` · `\o100` · `\u0040` · `\newline`

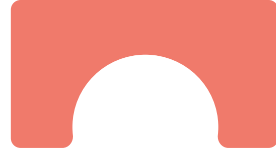
Cadenas: `"Hola \"mundo\"."`

Booleanos: `true` / `false`

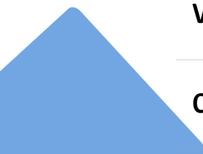
Nulo: `nil` → `null` en Java

Constantes simbólicas: `##Inf` · `##-Inf` · `##NaN`

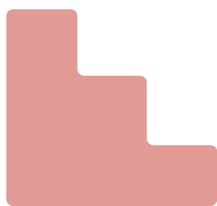
Palabras clave: en mapas `:` como key

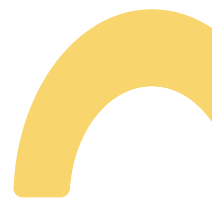


Colecciones



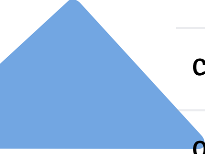
Tipo	Sintaxis	Ejemplo
Listas — código en Clojure	`()`	`(+ P 1)`
Vectores — datos (LIFO)	`[]`	`[0 1 1 2 3]`
Colas — datos (FIFO)	`PersistentQueue/EMPTY`	—
Conjuntos — datos no repetidos	`#{}`	`#{2 3 5 7}`
Mapas — entidades key-value	`{}`	`{:x 10, :y 15}`

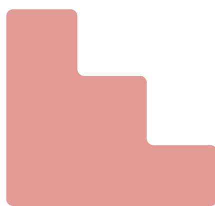


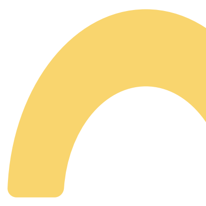


Colecciones - Comparación

	Listas	Vectores	Colas	Conjuntos	Mapas
Acceso	Secuencial	Aleatorio	Secuencial	Secuencial	Aleatorio
Repetidos	Sí	Sí	Sí	No	No
Contiene	Operaciones	Datos	Datos	Datos	Entidades
Orden	Inserción	Inserción	Inserción	hash-set / sorted-set	Clave-valor
Tipo	LIFO	LIFO	FIFO	Por dato	Por clave







Secuencias

- Interfaz **ISeq** — abstracción que representa una vista secuencial de una colección
- Métodos fundamentales: `first`, `rest`, `cons`
- **Lazy evaluation** — los elementos se computan bajo demanda

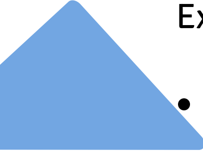
```
user=> (first [10 20 30])
10

user=> (rest [10 20 30])
(20 30)

user=> (cons 0 [10 20 30])
(0 10 20 30)

user=> (take 5 (range))
(0 1 2 3 4)
```

Evaluación de expresiones

Sentencias vs Expresiones

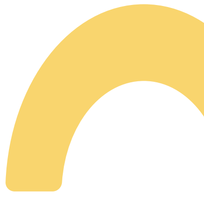
Todo es una expresión a evaluar:

- **Dato** → dato
- **Expresión** → expresión o dato



Excepciones:

- **Símbolos** (valor al que se refieren)
- **Listas** (invocaciones)



Ejemplo - Evaluación de expresiones

```
user⇒ 1
1
```

```
user⇒ [1 2 3]
[1 2 3]
```

```
user⇒ (+ 3 4)
7
```

```
user⇒ (1 2 3)
ClassCastException java.lang.Long cannot be cast to clojure.lang.IFn
```

```
user⇒ '(1 2 3)
(1 2 3)
```

Evaluación - Primer elemento

En una lista, el primer elemento es lo que va a ser evaluado y lo que lo prosigue, son sus argumentos.

```
;; Macro en la primera posición
user⇒ (defn suma [a b] (+ a b))
#'user/suma
```

```
;; Función en la primera posición
user⇒ (suma 5 6)
11
```

```
;; Palabra clave en la primera posición
user⇒ (:v1 '{:v2 b, :v1 a, :v3 c})
a
```

```
;; Vector en la primera posición
user⇒ ([0 10 20 30 40] 3)
30
```

Evaluación - Orden de argumentos

El orden de los argumentos depende del tipo de elemento y el tipo de operación.

```
;; Operación sobre una colección
user=> (conj [1 2 3] 4)
[1 2 3 4]
```

```
;; Operación sobre una secuencia
user=> (cons 1 [2 3 4])
(1 2 3 4)
```

```
;; Otras no toman argumentos
user=> (do (print "Nombre: ")
          (flush)
          (let [n (read)] (print (str "Hola ")) n))
```

Formas especiales

IF · QUOTE

```
;; IF ⇒ condicional ⇒ (if a b c) o (if a b)
user⇒ (if (= 3 4) ([10 20 30] 2) ([40 50 60] 1))
50

user⇒ (if (= 3 4) ([10 20 30] 2))
nil
```

```
;; QUOTE ⇒ no evalúa ⇒ (quote expresion) o 'expresion
user⇒ (quote (+ 3 2))
(+ 3 2)

user⇒ '(+ 3 2)
(+ 3 2)
```

FN

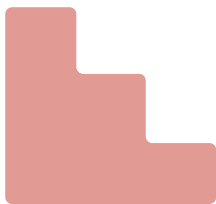
Sin sobrecarga por aridad:

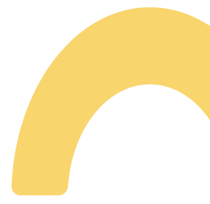
```
(fn name? [params*] condition-map? expr*)
```

Con sobrecarga por aridad:

```
(fn name? ([params*] condition-map? expr*)+)
```

- `name?` — nombre, permite llamadas recursivas
- `params*` — parámetros de la función
- `condition-map?` — pre- y postcondiciones
- `expr*` — son evaluadas, pero solo devuelve el valor de la última





FN - Ejemplos

```
user=> ((fn [a b] (+ a b)) 3 5)
8
```

```
user=> ((fn fact [n] (if (zero? n) 1 (* n (fact (- n 1))))) 5)
120
```

```
user=> ((fn ([] 0)
            ([x] x)
            ([x y] (+ x y))
            ([x y & more] (+ x y (reduce + more)))) 2 3 5 2)
12
```

```
;; Funciones anónimas abreviadas
user=> (#(* % %) 3)
9

user=> (#(+ (* %1 %1) (* %2 %2)) 3 4)
25
```

DEF

Crea y devuelve una **Var** (referencia) y registra en el namespace.

```
user⇒ x
CompilerException java.lang.RuntimeException:
  Unable to resolve symbol x in this context
```

```
user⇒ (def x)
#'user/x

user⇒ x
#object[clojure.lang.Var$Unbound 0×b07f29 "Unbound: #'user/x"]
```

```
user⇒ (def x 1)
#'user/x

user⇒ x
1

user⇒ (class x)
java.lang.Long
```

VAR · DO

```
;; VAR ⇒ definición
user⇒ (var x)
#'user/x

user⇒ (class (var x))
clojure.lang.Var
```

```
;; DO ⇒ evalúa grupo de expresiones, devuelve la última
user⇒ (do)
nil

user⇒ (if (= 2 (+ 1 1)) (do 1 2 3) 4)
3

user⇒ (if (= 2 (+ 1 1)) (do (println 1) (println 2) 3) 4)
1
2
3
```

LET · TRY-CATCH-FINALLY

```
;; LET ⇒ evalúa expresiones con constantes locales
;; (let [binding*] expr*)
user⇒ (let [a [1 2 3], b 4]
        (println (list a b b a))
        (list b a a b))
([1 2 3] 4 4 [1 2 3])
(4 [1 2 3] [1 2 3] 4)
```

```
;; TRY-CATCH-FINALLY ⇒ ídem que en Java
;; (try expreT* (catch classname name expreC*)* (finally expreF*)?)
user⇒ (try (/ 1 0)
        (catch Exception e
          (println "Exception:" (.getMessage e)))
        (finally (println "Good bye.")))
Exception: Divide by zero
Good bye.
nil
```

. (punto) - Interop con Java

Permite acceder a las funciones de Java. 1er argumento: nombre de clase. 2do argumento: símbolo (atributo) o lista (método).

```
;; Acceso a atributo
user=> (. Math PI)
3.141592653589793
```

```
;; Llamada a método
user=> (. (. System (getProperties)) (get "java.runtime.version"))
"1.8.0_60-b27"
```

```
;; Lo más común es usar macros
user=> (.toUpperCase "Hola")
"HOLA"

user=> (.indexOf '(a b c d) 'c)
2
```

Funciones predefinidas

Categoría	Funciones
Aritméticas	`+`, `-`, `*`, `/`, `mod`, `inc`, `dec`, `max`, `min`
Comparación	`=`, `<`, `>`, `≤`, `≥`, `not=`, `compare`
Predicados	`nil?`, `zero?`, `pos?`, `neg?`, `even?`, `odd?`, `empty?`
Colecciones	`count`, `conj`, `assoc`, `get`, `first`, `rest`, `last`, `nth`
Orden superior	`map`, `filter`, `reduce`, `apply`, `partial`, `comp`
Strings	`str`, `subs`, `clojure.string/upper-case`, `clojure.string/trim`
Conversión	`int`, `float`, `str`, `keyword`, `symbol`, `vec`, `set`

Referencia completa: clojure.org/api

Recursos

Oficiales

- clojure.org — página oficial del lenguaje
- Guía de instalación — incluye instrucciones para Windows, macOS y Linux
- Referencia del lenguaje — especificación y referencia oficial

Documentación y práctica

- ClojureDocs — documentación con ejemplos de la comunidad
- TryClj — REPL online para probar Clojure en el navegador
- Clojure for the Brave and True — tutorial completo y gratuito

Resumen

- **Origen**: Rich Hickey, 2007 — dialecto de Lisp sobre la JVM
- **Datos**: escalares, colecciones (listas, vectores, mapas, conjuntos) y secuencias lazy
- **Evaluación**: todo es una expresión; el primer elemento se evalúa como operador
- **Formas especiales**: if, quote, fn, def, var, do, let, try-catch-finally
- **Interop Java**: acceso directo a clases y métodos de la JVM

¡Muchas Gracias!