Clojure

Clojure (*clojure.org*) es un dialecto moderno (surgió en 2007) del lenguaje Lisp (*LISt Processor*). Lisp fue desarrollado originalmente en 1958 por el grupo de Inteligencia Artificial del M.I.T. como un sistema para manipular expresiones que representaran oraciones y hacer deducciones. En su artículo titulado *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I* de 1960, John McCarthy menciona que después de atravesar varias etapas de simplificación durante su desarrollo, Lisp finalmente pasó a basarse en un esquema para representar funciones inspirado en el Cálculo Lambda. Esto queda bastante en evidencia al observar una función para calcular el cuadrado de un número, aplicada al número 3, que puede escribirse así en Cálculo Lambda, Lisp y Clojure:

Cálculo Lambda (con combinadores)	Lisp	Clojure
(λx.Mul x x) 3	**((lambda (x) (* x x)) 3)**	**((fn [x] (* x x)) 3)**

Algunas características de Clojure son las siguientes:

- Es *homoicónico* (los programas se escriben usando las propias estructuras de datos del lenguaje).
- Favorece la programación concurrente al ofrecer *estructuras de datos persistentes* (si un dato no existe, se lo crea, pero si ya existe, se lo aprovecha para crear otro; por ejemplo, cuando se inserta un dato en la cabeza de una lista, los datos de la lista original persisten como cola de la nueva lista) e *inmutables* (sus valores no se pueden cambiar).
- Le da preferencia a la recursividad y las *funciones de orden superior* (es decir, funciones que reciben y/o devuelven otras funciones), en lugar de la iteración basada en efectos secundarios/modificación de variables.
- Permite representar conjuntos potencialmente infinitos, dado que los elementos de las *secuencias* no se computan cuando no son necesarios (*lazy evaluation* o evaluación perezosa).
- Se integra fácilmente en entornos Java, dado que las aplicaciones escritas en Clojure corren en la JVM (*Java Virtual Machine*).
- Ofrece una consola de evaluación (REPL: *read-eval-print-loop*) para facilitar el desarrollo dinámico. No se trata de un intérprete, ya que las expresiones leídas se compilan a *bytecode* de la JVM antes de ser ejecutadas. Los ejemplos que se darán en este documento provienen de la consola REPL. Algunos ejemplos útiles referidos al uso de la consola 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 (el código fuente) de **+**
user=> (load-file "op.clj")	Evalúa secuencialmente el contenido de **op.clj**
user=> (clojure-version)	Muestra la versión de Clojure

Limitaremos el estudio de este lenguaje a las siguientes 6 partes:

1. Datos;
2. Evaluación de expresiones;
3. Formas especiales;
4. Funciones predefinidas;
5. Funciones de orden superior;
6. Macros predefinidas.

1. DATOS

Un *dato* puede ser:

- un *escalar*;
- una *colección* cuyos elementos son datos;
- una *secuencia* (una abstracción que representa una vista secuencial de una colección).

1.1. Escalares

Se caracterizan por referirse a un único elemento, que puede ser un símbolo o un valor.

a) Símbolos

Un símbolo es un dato que representa el *nombre* de algo (una función, un parámetro, etc.). Por ejemplo, **a** es un símbolo. **Clojure es sensible a mayúsculas y minúsculas**. Casi todos los caracteres están permitidos en los símbolos, siempre y cuando estos comiencen con un carácter no numérico. Para evitar que Clojure intente evaluar y resolver un símbolo (devolviendo el valor a que este se refiere), se utiliza el apóstrofo.

```
user=> a
CompilerException java.lang.RuntimeException: Unable to resolve symbol:
a in this context, compiling:(NO_SOURCE_PATH:0:0)
user=> 'a
a
```

b) Valores literales

Un valor literal puede ser de cualquiera de los siguientes tipos:

- Números: Pueden ser enteros, de punto flotante o racionales.
  I) Enteros: Por defecto **`long`**, de lo contrario (o si tienen el sufijo **`N`**) **`BigInt`**. Se les puede indicar la base. Por ejemplo, **`42`** equivale a **`2r101010`**, **`8r52`**, **`052`**, **`16r2a`**, **`0x2A`** y **`36r16`**.
  II) Punto flotante: Por defecto **`double`**, de lo contrario (o si tienen el sufijo **`M`**) **`BigDecimal`**.
  III) Racionales: representan una fracción. Por ejemplo: **`1/3`**.
- Caracteres: Pueden representarse con la barra invertida antepuesta, como el propio valor o mediante su codificación en octal o en hexadecimal (Unicode) y, en algunos casos, también mediante su denominación.
  I) Representación como el propio valor, por ejemplo: **@** es **`\@`**.
  II) Representación mediante la codificación en octal: **@** es **`\o100`**.
  III) Representación mediante la codificación en hexadecimal: **@** es **`\u0040`**.
  IV) Representación mediante su denominación: **`\newline`**, **`\return`**, **`\space`**, **`\backspace`**, **`\tab`** y **`\formfeed`**.
- Cadenas de caracteres: Se representan como series de caracteres encerradas entre comillas. Pueden extenderse por más de un renglón y soportan caracteres de escape como en Java.
  Por ejemplo: **`"La cadena \"Hola mundo\" es usada al aprender a programar"`**
- Booleanos: Son los valores de verdad **true** y **false**.
- Nulo: Es el valor **`nil`** (significa *nada* y representa una referencia nula, como **`null`** en Java).
- Constantes simbólicas: Representan valores especiales: **`##Inf`**, **`##-Inf`** y **`##NaN`** (desde Clojure 1.9).
- Palabras clave (*keywords*): Utilizadas para indexar los valores de los mapas, tienen casi la forma de los símbolos (se diferencian porque las palabras clave deben comenzar con el carácter **`:`** y porque el carácter **`.`** no está permitido). Por ejemplo, **`:primero`** es una palabra clave válida.

1.2. Colecciones

Se utilizan como contenedores de otros datos. Las principales son las listas, los vectores, las colas, los conjuntos y los mapas.

a) Listas

Se usan típicamente para representar código de Clojure. Se escriben como cero o más datos encerrados entre paréntesis, por ejemplo: `()` es la lista vacía y `(+ P 1)` es el código que suma **P** y **1**. Opcionalmente, se pueden usar comas para separar los datos. Son útiles para:

- acceder secuencialmente a los datos;
- mantener datos repetidos;
- mantener el orden de los datos a medida que se insertan;
- agregar y quitar rápidamente datos del frente (LIFO).

b) Vectores

Se usan típicamente para representar datos. Se codifican como cero o más datos encerrados entre corchetes, por ejemplo: `[]` es el vector vacío y `[0 1 1 2 3]` son los primeros 5 números de Fibonacci. Opcionalmente, se pueden usar comas para separar los datos. Son útiles para:

- acceder aleatoriamente a los datos (por índice);
- mantener datos repetidos;
- mantener el orden de los datos a medida que se insertan;
- agregar y quitar rápidamente datos del final (LIFO).

c) Colas

Se usan típicamente para representar datos que van a ser utilizados de acuerdo con el método FIFO. No poseen una codificación literal, por lo que deben construirse a partir de la cola vacía `clojure.lang.PersistentQueue/EMPTY`. Son útiles para:

- acceder secuencialmente a los datos;
- mantener datos repetidos;
- mantener el orden de los datos a medida que se insertan;
- agregar datos al final y quitar datos del frente rápidamente (FIFO).

d) Conjuntos

Se usan típicamente para representar datos no repetidos. Se codifican como cero o más datos encerrados entre llaves, estando la de apertura precedida por el carácter **#**, por ejemplo: `#{}` es el conjunto vacío y `#{2 3 5 7}` son los primeros cuatro números primos. Opcionalmente, se pueden usar comas para separar los datos. Son útiles para:

- mantener datos sin repeticiones;
- agregar o eliminar un dato dado;
- verificar si existe un dato en el conjunto.

Existen dos tipos de conjuntos en Clojure: `hash-set` y `sorted-set`. La principal diferencia es que los segundos mantienen los datos ordenados.

e) Mapas

Se usan típicamente para representar entidades. Se codifican como cero o más pares de datos encerrados entre llaves, siendo el primero la clave (que solo puede aparecer una única vez) y el segundo el valor, por ejemplo: **`{}`** es el mapa vacío y **`{:x 10, :y 15}`** es la representación de un punto en el plano. Opcionalmente, se puede usar una coma entre un par de datos y otro, para separarlos. Son útiles para:

- acceder aleatoriamente a los datos (por clave);
- agregar asociaciones de claves únicas y valores;
- eliminar pares clave-valor, dada la clave;
- verificar si existe una clave en el mapa.

Existen dos tipos de mapas en Clojure: **`hash-map`** y **`sorted-map`**. La principal diferencia es que los segundos mantienen los datos ordenados según la clave.

1.3. Secuencias

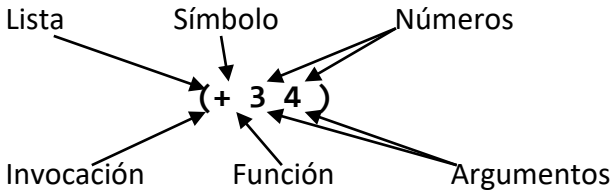
Mediante la interfaz **`ISeq`**, Clojure ofrece funciones de propósito general para trabajar con secuencias o *seqs*. Una *seq* es una abstracción que representa una vista secuencial de una colección, o sea una lista lógica que proporciona un acceso estable a una secuencia de valores. Las listas concretas implementan **`ISeq`** y son, por lo tanto, secuencias. Las demás colecciones, en cambio, no lo son, pero muchas funciones toman una o más colecciones que implementan **`Sequable`**, las convierten en secuencias y operan sobre ellas. Casi todas las funciones de la biblioteca de secuencias son perezosas (*lazy*), es decir, las funciones que devuelven *seqs* lo hacen de forma incremental y, por lo tanto, también consumen de forma incremental cualquier argumento que sea una *seq*.

2. EVALUACIÓN DE EXPRESIONES

En muchos lenguajes se hace una distinción entre *sentencias* (que al ser ejecutadas provocan algún efecto pero que no devuelven ningún valor) y *expresiones* (que al ser evaluadas devuelven un valor). En Clojure todo lo que hay son expresiones que al ser evaluadas devuelven un valor.
Al evaluar la mayoría de los datos, el resultado son ellos mismos:

```
user=> 1
1
user=> [1 2 3]
[1 2 3]
```

Las dos excepciones son los *símbolos* (que al evaluarse se resuelven como el valor a que se refieren) y las *listas* (que se evalúan como invocaciones), a no ser que estén precedidos por un apóstrofo.




```
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)
```

En una invocación, la primera posición de la lista puede estar ocupada por un símbolo (por ejemplo, el nombre de una función o una macro), una palabra clave, ciertas colecciones, etc.

Ejemplo de *macro* en la primera posición de la lista:

```
user=> (defn suma [a b] (+ a b))
#'user/suma
```

Ejemplo de *función* en la primera posición de la lista:

```
user=> (suma 5 6)
11
```

Ejemplo de *palabra clave* en la primera posición de la lista:

```
user=> (:v1 '{:v2 b, :v1 a, :v3 c})
a
```

Ejemplo de *vector* en la primera posición de la lista:

```
user=> ([0 10 20 30 40] 3)
30
```

El orden de los argumentos depende del tipo de operación: las operaciones sobre colecciones suelen tomar cierta colección como primer argumento y devolver una colección del mismo tipo. Las operaciones sobre secuencias, en cambio, suelen tomar una secuencia (o una colección que convierten en una secuencia) como último argumento y devolver una secuencia.

Ejemplo de operación sobre una *colección*:	Ejemplo de operación sobre una *secuencia*:
`user=> (conj [1 2 3] 4)` `[1 2 3 4]`	`user=> (cons 1 [2 3 4])` `(1 2 3 4)`

Algunas funciones como **`flush`** nunca toman argumentos. Otras como **`read`** a veces pueden no tomarlos:

```
user=> (do (print "Nombre: ")(flush)(let [n (read)] (print "Hola ") n))
Nombre: Diego
Hola Diego
```

3. FORMAS ESPECIALES

Las formas especiales tienen reglas de evaluación que difieren de las reglas estándar. Su listado completo se obtiene evaluando: **`(pprint (keys (. clojure.lang.Compiler specials)))`**. A continuación se describen algunas de ellas.

3.1. if

Se puede usar su estructura completa: **`(if a b c)`** y también omitir el tercer argumento: **`(if a b)`**. Si el resultado de evaluar **`a`** no es **`false`** ni **`nil`** (conocidos como valores *falsey*), se devuelve el resultado de evaluar **`b`**. De lo contrario (valores *truthy*), se devuelve el resultado de evaluar **`c`** (o **`nil`** cuando no se incluye **`c`**). Por ejemplo:

```
user=> (if (= 3 3) ([10 20 30] 2) ([40 50 60] 1))
30
user=> (if (= 3 4) ([10 20 30] 2) ([40 50 60] 1))
50
user=> (if (= 3 4) ([10 20 30] 2))
nil
```

3.2. quote

Devuelve un símbolo o una lista, sin evaluarlos. Se abrevia con el apóstrofo. Por ejemplo:

```
user=> (quote (+ 3 2))        equivale a        user=> '(+ 3 2)
(+ 3 2)                                         (+ 3 2)
```

3.3. fn

Devuelve una función. Su estructura está dada por una de las siguientes expresiones regulares:

Para funciones sin sobrecarga por aridad (cantidad de parámetros):

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

Para funciones con sobrecarga por aridad (cantidad de parámetros):

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

name: aparece cero o una vez. Es un símbolo ligado a la propia función, solo visible dentro de ella, para permitir las llamadas recursivas.

params: aparecen cero o más veces. Son los parámetros que se ligarán a los argumentos al invocar la función. Cuando el penúltimo parámetro sea **&**, el último representará una secuencia que contendrá los argumentos restantes.

condition-map: aparece cero o una vez. Se utiliza para especificar pre- y postcondiciones para la función y trabajar con aserciones.

expr: aparecen cero o más veces. Son evaluadas todas, pero la función solo devuelve el valor de la última.

En Clojure, las funciones son valores, de modo que pueden ser tratadas como cualquier otro dato, aceptadas como argumentos y devueltas como resultados por otras funciones: son elementos de *primera clase*.

Ejemplos:

Aquí la función devuelta por **fn** no se utiliza:

```
user=> (fn [])
#object[user$eval301$fn__302 0x1128536 "user$eval301$fn__302@1128536"]
```

En los siguientes casos la función devuelta por **fn** sí se utiliza, invocándola con argumentos:

Función con aridad 0 (sin parámetros):

```
user=> ((fn []))
nil
```

Función con aridad 2:

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

Función recursiva con aridad 1:

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

Función con sobrecarga por aridad (por soportar aridad indefinida se trata de una función *variádica*):

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

Función con notación abreviada (aridad 1):

```
user=> (#(* % %) 3)
9
```

Función con notación abreviada (aridad 2):

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

3.4. def

Crea y devuelve una *Var*, que es uno de los mecanismos que ofrece Clojure para mantener una referencia a un valor mutable. Una *Var* ligada a una ubicación de almacenamiento mutable puede ser ligada dinámicamente a otra ubicación de almacenamiento mutable, ya que el aislamiento de hilos garantiza el uso seguro de estas ubicaciones de almacenamiento.
Al usar **def**, además, la *Var* queda registrada en el espacio de nombres (*namespace*) global que guarda el mapeo entre símbolos y *Vars*:

El símbolo **x** no está en el espacio de nombres global:

```
user=> x
CompilerException java.lang.RuntimeException: Unable to resolve symbol:
x in this context, compiling:(NO_SOURCE_PATH:0:0)
```

Aquí se crea una *Var*, se la registra en el *namespace* global con el nombre **x** y se la devuelve:

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

Ahora el símbolo **x** ya está en el espacio de nombres global:

```
user=> x
#object[clojure.lang.Var$Unbound 0xb07f29 "Unbound: #'user/x"]
```

Sin embargo, el símbolo **x** está mapeado con una *Var* que aún no está ligada a un valor:

```
user=> (class x)
clojure.lang.Var$Unbound
```

Aquí, además de crear una *Var* y mapearla con **x**, se le liga el valor **1** antes de devolverla:

```
user=> (def x 1)
#'user/x
```

Ahora el símbolo **x** está en el espacio de nombres global, y al evaluarlo se obtiene el valor que está ligado a la *Var* mapeada con el símbolo:

```
user=> x
1
```

Aquí se consulta a qué clase pertenece el valor ligado a la *Var* mapeada con **x**:

```
user=> (class x)
java.lang.Long
```

Dado que las funciones en Clojure son valores, puede usarse **def** para darles un nombre:

```
user=> (def suma (fn [a b] (+ a b)))
user=> (suma 2 3)
5
```

No obstante, para ello es más práctico utilizar la macro **defn**:

```
user=> (defn suma [a b] (+ a b))
user=> (suma 4 5)
9
```

A veces, incluso se acostumbra combinar **defn** con **def**:

```
user=> (defn pot [n] (fn [x] (apply * (repeat n x))))
user=> (def cubo (pot 3))
user=> (cubo 2)
8
```

La función **pot** devuelve una función. La función anónima dentro de **pot** es una *closure* (clausura) porque utiliza **n** que está definida fuera de su *scope* (alcance).

3.5. var

Como se vio en el punto anterior, una vez que un símbolo fue mapeado con una *Var*, al evaluarlo se obtiene el valor ligado a esta. Para obtener la *Var*, en lugar del valor al que está ligada, se utiliza **var**:

Aquí se devuelve la misma *Var* que fue creada en el punto anterior:

```
user=> (var x)
#'user/x
```

En efecto, el símbolo **x** está mapeado con una *Var*:

```
user=> (class (var x))
clojure.lang.Var
```

3.6. do

Evalúa un grupo de expresiones, de la primera a la última, y devuelve el valor de la última (o **nil**, si no se le proporciona ninguna expresión). Ejemplos:

La forma especial **do** usada sin expresiones:

```
user=> (do)
nil
```

Aquí se evalúan tres expresiones y se devuelve el valor de la última:

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

Aquí también se evalúan tres expresiones y se devuelve el valor de la última, pero las dos primeras tienen efectos secundarios:

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

3.7. let

Evalúa un grupo de expresiones, de la primera a la última, y devuelve el valor de la última (o **nil**, si no se le proporciona ninguna expresión), usando los valores constantes de un grupo de símbolos locales definidos secuencialmente. Su estructura está dada por la siguiente expresión regular:

```
(let [binding*] expr*)
```

Ejemplos:

La forma especial **let** usada sin expresiones:

```
user=> (let [])
nil
```

Aquí se evalúan dos expresiones y se devuelve el valor de la última, pero la primera tiene efectos secundarios:

```
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)
```

3.8. try-catch-finally

Esta forma especial tiene un comportamiento análogo al de su equivalente en Java. Su estructura está dada por la siguiente expresión regular:

```
(try expreT* (catch classname name expreC*)* (finally expreF*)?)
```

Las expresiones **expreT** son evaluadas y, si no se producen excepciones, se devuelve el valor de la última expresión. Si se produce una excepción y se proporcionan cláusulas **catch**, cada una se examina a su vez y, para la primera cláusula **catch** coincidente, se evalúan sus expresiones **expreC** y el valor de la última es el valor de retorno de la función. Si no hay una cláusula **catch** coincidente, la excepción se propaga fuera de la función. Antes de retornar, normal o anormalmente, se evalúan las expresiones **expreF** de la cláusula **finally** por si tienen efectos secundarios. Por ejemplo:

```
user=> (try (/ 1 0)
            (catch Exception e (println "Exception:" (.getMessage e)))
            (finally (println "Good bye.")))
Exception: Divide by zero
Good bye.
nil
```

3.9. . (punto)

Esta forma especial es la base para el acceso a Java desde Clojure. Si el primer operando es un símbolo que se resuelve como un nombre de clase, se considera el acceso a un miembro estático de la clase nombrada. Si el segundo operando es un símbolo y no se proporciona ningún argumento, se considera que es el acceso a un atributo. En cambio, si el segundo operando es una lista, o se proporcionan argumentos, se considera como una llamada a un método. Si el método tiene un tipo de retorno **void**, el valor de la expresión es **nil**.

Ejemplos:

El primer operando **Math** es un nombre de clase y el segundo operando es un símbolo:

```
user=> (. Math PI)
3.141592653589793
```

El primer operando **System** es un nombre de clase y el segundo operando es una lista:

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

El método **println** tiene tipo de retorno **void**:

```
user=> (. System/out println "Hola")
Hola
nil
```

En general, el acceso a Java desde Clojure se lleva a cabo mediante macros que se expanden a la forma especial **.** (punto). Por ejemplo:

Uso de una macro para acceder a Java:

```
user=> (.toUpperCase "Hola")
"HOLA"
```

Uso equivalente con la forma especial **.** (punto):

```
user=> (macroexpand-1 '(.toUpperCase "Hola"))
(. "Hola" toUpperCase)
```

Uso de otra macro para acceder a Java:

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

Las posiciones se cuentan desde **0**

4. FUNCIONES PREDEFINIDAS

Clojure ofrece un gran repertorio de funciones predefinidas. Por ejemplo, una de las más utilizadas es la función **=**, que devuelve **`true`** si sus argumentos son iguales; si no, **`false`**. A continuación se muestran algunas de las demás funciones disponibles, agrupadas según los datos con que trabajan.

4.1. Símbolos

```
user=> (def a (symbol "b"))
#'user/a
```

symbol devuelve un símbolo con el nombre indicado

```
user=> a
b
```

```
user=> (symbol? a)
true
```

symbol? devuelve **`true`** si el argumento es un **Symbol**

```
user=> (class a)
clojure.lang.Symbol
```

4.2. Números

number? Devuelve **`true`** si el argumento es un **Number**; si no, **`false`**

a) Cálculos sin autopromoción

Lanzan una **`ArithmeticException`** si el resultado es incompatible con el tipo de los operandos. Por ejemplo:

```
user=> (+ 5500000000000000000 5500000000000000000)
ArithmeticException integer overflow clojure.lang.Numbers.throwInt
Overflow (Numbers.java:1501)
```

+	Suma los operandos (Pueden ser más de dos. Si no los hay, devuelve **0**)
-	Resta los operandos (Pueden ser más de dos. Si solo hay uno, le cambia el signo)
*****	Multiplica los operandos (Pueden ser más de dos. Si no los hay, devuelve **1**)
/	Divide los operandos (Pueden ser más de dos. Si solo hay uno, devuelve su recíproco)
inc	Devuelve el operando incrementado en **1**
dec	Devuelve el operando decrementado en **1**
quot	Devuelve el cociente (división entera) entre los dos operandos
rem	Devuelve el resto de la división entera entre los dos operandos

b) Cálculos con autopromoción

No lanzan una **`ArithmeticException`** si el resultado es incompatible con el tipo de los operandos, ya que estos se autopromueven para que el resultado sea de un tipo más amplio. Por ejemplo, en el siguiente caso el resultado es un **`BigInt`**:

```
user=> (+' 5500000000000000000 5500000000000000000)
11000000000000000000N
```

+'	Suma los operandos (Pueden ser más de dos. Si no los hay, devuelve **0**)
-'	Resta los operandos (Pueden ser más de dos. Si solo hay uno, le cambia el signo)
***'**	Multiplica los operandos (Pueden ser más de dos. Si no los hay, devuelve **1**)
inc'	Devuelve el operando incrementado en **1**
dec'	Devuelve el operando decrementado en **1**

c) Relacionales

`==`	Devuelve **true** si los valores de los operandos son iguales; si no, **`false`**
`=`	Devuelve **true** si los valores y los tipos de los operandos son iguales; si no, **`false`**
`not=`	Devuelve **true** si los valores o los tipos de los operandos son distintos; si no, **`false`**
`<`	Devuelve **true** si los operandos están en orden monótono creciente; si no, **`false`**
`<=`	Devuelve **true** si los operandos están en orden monótono no-decreciente; si no, **`false`**
`>`	Devuelve **true** si los operandos están en orden monótono decreciente; si no, **`false`**
`>=`	Devuelve **true** si los operandos están en orden monótono no-creciente; si no, **`false`**
`zero?`	Devuelve **true** si el valor del argumento es igual a **`0`**; si no, **`false`**
`pos?`	Devuelve **true** si el valor del argumento es mayor que **`0`**; si no, **`false`**
`neg?`	Devuelve **true** si el valor del argumento es menor que **`0`**; si no, **`false`**
`even?`	Devuelve **true** si el resto de dividir por **2** el argumento es igual a **`0`**; si no, **`false`**
`odd?`	Devuelve **true** si el resto de dividir por **2** el argumento es igual a **`1`**; si no, **`false`**

d) Fracciones

`numerator`	Devuelve el valor del numerador de un número racional
`denominator`	Devuelve el valor del denominador de un número racional

e) Selección

`min`	Devuelve el valor del menor de los argumentos (el mínimo)
`max`	Devuelve el valor del mayor de los argumentos (el máximo)
`rand-int`	Devuelve un entero al azar entre **`0`** (incl.) y el argumento (excl.)

f) Manipulación de bits

`bit-and`	Devuelve el **`and`** realizado bit a bit entre dos o más argumentos
`bit-or`	Devuelve el **`or`** realizado bit a bit entre dos o más argumentos
`bit-xor`	Devuelve el **`xor`** realizado bit a bit entre dos o más argumentos
`bit-not`	Devuelve el **`not`** realizado bit a bit sobre el argumento
`bit-shift-right`	Devuelve el primer argumento con sus bits desplazados hacia la derecha tantas veces como indica el segundo argumento
`bit-shift-left`	Devuelve el primer argumento con sus bits desplazados hacia la izquierda tantas veces como indica el segundo argumento
`bit-clear`	Devuelve el primer argumento con **`0`** en el bit indicado por el segundo
`bit-set`	Devuelve el primer argumento con **`1`** en el bit indicado por el segundo
`bit-flip`	Devuelve el primer argumento cambiando el bit indicado por el segundo
`bit-test`	Devuelve **`true`** si en el primer argumento el bit indicado por el segundo argumento es **`1`**; si no, **`false`**

g) Casteos explícitos

`int`	Devuelve el valor del argumento convertido a **`int`**
`bigdec`	Devuelve el valor del argumento convertido a **`BigDecimal`**
`bigint`	Devuelve el valor del argumento convertido a **`BigInt`**
`double`	Devuelve el valor del argumento convertido a **`double`**
`float`	Devuelve el valor del argumento convertido a **`float`**
`long`	Devuelve el valor del argumento convertido a **`long`**
`num`	Devuelve el valor del argumento convertido a **`Number`**
`short`	Devuelve el valor del argumento convertido a **`short`**

4.3. Caracteres

char	Devuelve el valor del argumento convertido a **char**
char?	Devuelve **true** si el argumento es un **Character**; si no, **false**

También se pueden usar los métodos de la clase **Character** de Java, por ejemplo:

```
user=> (Character/isLetter \q)
true
user=> (Character/digit \a 16)
10
```

Los mapas **char-name-string** y **char-escape-string** se pueden usar como si fueran funciones:

```
user=> (char-name-string \u000A)
"newline"
user=> (char-escape-string \u000A)
"\\n"
```

4.4. Cadenas de caracteres

Las cadenas de caracteres de Clojure son objetos de la clase **String** de Java. Se suelen procesar usando la forma especial **.** (punto) o macros. Por ejemplo:

```
user=> (seq (.split "Esta frase tiene cinco palabras" " "))
("Esta" "frase" "tiene" "cinco" "palabras")
```

Sin embargo, Clojure también proporciona algunas funciones propias:

str	Devuelve el valor del argumento convertido a **String**. Invocada sin argumento o con **nil** como único argumento, devuelve la cadena vacía. Con dos o más argumentos, devuelve su concatenación.
string?	Devuelve **true** si el argumento es un **String**; si no, **false**
replace	Devuelve el resultado de reemplazar, en el primer argumento, las subcadenas que coincidan con el segundo argumento por la cadena indicada en el tercero.

```
user=> (clojure.string/replace "El color es rojo" #"rojo" "azul")
"El color es azul"
```

pr imprime sin salto de línea, en el formato del lector estándar de Clojure, y devuelve **nil**:

```
user=> (pr "Hola")
"Hola"nil
```

prn imprime con salto de línea, en el formato del lector estándar de Clojure, y devuelve **nil**:

```
user=> (prn "Hola")
"Hola"
nil
```

print imprime sin salto de línea, en el formato del lector humano (sin comillas), y devuelve **nil**:

```
user=> (print "Hola")
Holanil
```

println imprime con salto de línea, en el formato del lector humano (sin comillas), y devuelve **nil**:

```
user=> (println "Hola")
Hola
nil
```

printf imprime en el formato indicado mediante el primer argumento y devuelve **nil**:

```
user=> (printf "%s\n" "Hola")
Hola
nil
```

4.5. Booleanos

boolean	Devuelve el valor del argumento convertido a **Boolean**
boolean?	Devuelve **true** si el argumento es un **Boolean**; si no, **false** (desde Clojure 1.9)
true?	Devuelve **true** si el valor del argumento es **true**; si no, **false**
false?	Devuelve **true** si el valor del argumento es **false**; si no, **false**
not	Devuelve **true** si el valor del argumento es **false** o **nil**; si no, **false**

4.6. Nulo

nil?	Devuelve **true** si el valor del argumento es **nil**; si no, **false**
some?	Devuelve **true** si el valor del argumento es distinto de **nil**; si no, **false**

4.7. Palabras clave (*keywords*)

keyword	Devuelve una palabra clave construida anteponiéndole **:** al argumento
keyword?	Devuelve **true** si el valor del argumento es una **Keyword**; si no, **false**

4.8. Colecciones

a) Creación de una colección vacía

```
user=> (list)                                    Equivale a ()
user=> (vector)                                  Equivale a []
user=> (clojure.lang.PersistentQueue/EMPTY)
user=> (hash-set)                                Equivale a #{}
user=> (sorted-set)
user=> (hash-map)                                Equivale a {}
user=> (sorted-map)
```

b) Creación de una colección con datos iniciales (con Cola/**PersistentQueue** no se puede hacer)

```
user=> (list 'a 'b 'c)
(a b c)                                          Lista
user=> (vector 'a 'b 'c)
[a b c]                                          Vector
user=> (hash-set 'a 'b 'c)
#{a c b}                                         Conjunto
user=> (sorted-set 'a 'c 'b)
#{a b c}                                         Conjunto ordenado
user=> (hash-map :v1 'a, :v2 'b, :v3 'c)
{:v2 b, :v1 a, :v3 c}                            Mapa
user=> (zipmap '[:v2 :v1 :v3] '[b a c])
{:v2 b, :v1 a, :v3 c}                            Mapa
user=> (sorted-map :v1 'a, :v3 'c, :v2 'b)
{:v1 a, :v2 b, :v3 c}                            Mapa ordenado
```

De aquí en más, para mostrar las funciones, se utilizarán las 7 colecciones definidas a continuación:

```
user=> (def L (conj (conj (conj () 'c) 'b) 'a))
user=> L
(a b c)                                                    Lista

user=> (def V (conj (conj (conj [] 'a) 'b) 'c))
user=> V
[a b c]                                                    Vector

user=> (def Q (conj (conj (conj clojure.lang.PersistentQueue/EMPTY 'a) 'b) 'c))
user=> (seq Q)
(a b c)                                                    Cola (usando seq para verla)

user=> (def HS (hash-set 'a 'b 'c))
user=> HS
#{a c b}                                                   Conjunto

user=> (def SS (sorted-set 'a 'c 'b))
user=> SS
#{a b c}                                                   Conjunto ordenado

user=> (def HM (hash-map :v1 'a, :v2 'b, :v3 'c))
user=> HM
{:v2 b, :v1 a, :v3 c}                                      Mapa

user=> (def SM (sorted-map :v1 'a, :v3 'c, :v2 'b))
user=> SM
{:v1 a, :v2 b, :v3 c}                                      Mapa ordenado
```

c) Predicados

- Verificación de clase

```
user=> (instance? clojure.lang.PersistentList L)
true

user=> (instance? clojure.lang.PersistentVector V)
true

user=> (instance? clojure.lang.PersistentQueue Q)
true

user=> (instance? clojure.lang.PersistentHashSet HS)
true

user=> (instance? clojure.lang.PersistentTreeSet SS)
true

user=> (instance? clojure.lang.PersistentHashMap HM)
true

user=> (instance? clojure.lang.PersistentTreeMap SM)
true
```

- Verificación de tipo

```
user=> (list? L)
true
user=> (vector? V)
true
user=> (list? Q)
true
user=> (set? HS)
true
user=> (set? SS)
true
user=> (map? HM)
true
user=> (map? SM)
true
user=> (coll? SM)
true
```

Devuelve **true** porque **SM** implementa **IPersistentCollection**

- Verificación de características

```
user=> (and (sequential? L) (counted? L))
true
user=> (and (sequential? V) (associative? V) (counted? V) (reversible? V))
true
user=> (and (sequential? Q) (counted? Q))
true
user=> (counted? HS)
true
user=> (and (sorted? SS) (counted? SS) (reversible? SS))
true
user=> (and (associative? HM) (counted? HM))
true
user=> (and (associative? SM) (sorted? SM) (counted? SM) (reversible? SM))
true
```

- Verificación de contenido

empty?	Devuelve **true** si el argumento es una colección vacía; si no, **false**
distinct?	Devuelve **true** si ningún argumento es igual a otro; si no, **false**
every?	Devuelve **true** si el primer argumento (un predicado) es **true** para todos los datos del segundo argumento; si no, **false**
not-every?	Devuelve **false** si el primer argumento (un predicado) es **true** para todos los datos del segundo argumento; si no, **true**
not-any?	Devuelve **false** si el primer argumento (un predicado) es **true** para algún dato del segundo argumento; si no, **true**
some	Devuelve **true** si el primer argumento (un predicado) es **true** para algún dato del segundo argumento; si no, **nil**

contains? Devuelve **true** si el primer argumento (un vector, un conjunto o un mapa) contiene un dato en la posición indicada por el segundo argumento (en el caso de los vectores), contiene el dato indicado por el segundo argumento (en el caso de los conjuntos) o contiene la clave indicada por el segundo argumento (en el caso de los mapas); si no, **false**. Por ejemplo:

```
user=> (contains? V 0)
true
user=> (contains? V 3)
false
user=> (contains? HS 'a)
true
user=> (contains? HS 'd)
false
user=> (contains? SS 'a)
true
user=> (contains? SS 'd)
false
user=> (contains? HM :v1)
true
user=> (contains? HM :v4)
false
user=> (contains? SM :v1)
true
user=> (contains? SM :v4)
false
```

d) Carga de un dato (no se modifican las colecciones originales, que son inmutables)

- **conj** (puede usarse con más de dos argumentos: (conj L 1 2) equivale a (conj (conj L 1) 2)

```
user=> (conj L 'd)
(d a b c)
```
Lista
(el dato queda primero)

```
user=> (conj V 'd)
[a b c d]
```
Vector
(el dato queda último)

```
user=> (seq (conj Q 'd))
(a b c d)
```
Cola (usando **seq** para verla)
(el dato queda último)

```
user=> (conj HS 'd)
#{a c b d}
```
Conjunto

```
user=> (conj SS 'd)
#{a b c d}
```
Conjunto ordenado

```
user=> (conj HM {:v4 'd})
{:v2 b, :v1 a, :v4 d, :v3 c}
```
Mapa
(la clave y el dato van juntos)

```
user=> (conj SM {:v4 'd})
{:v1 a, :v2 b, :v3 c, :v4 d}
```
Mapa ordenado
(la clave y el dato van juntos)

conj puede usarse con más de dos argumentos: **(conj L 1 2)** equivale a **(conj (conj L 1) 2)**

- **assoc**

  Funciona con vectores y mapas. El primer argumento es la colección, el segundo el índice (en el caso de los vectores) o la clave (en el caso de los mapas) y el tercero el dato a cargar. En vectores no se pueden usar índices menores que **0** ni mayores que el tamaño del vector. Por ejemplo:

```
user=> (assoc V 3 'd)
[a b c d]
user=> (assoc V 4 'd)
IndexOutOfBoundsException clojure.lang.PersistentVector.assocN ...
user=> (assoc HM :v4 'd)
{:v2 b, :v1 a, :v4 d, :v3 c}
user=> (assoc SM :v4 'd)
{:v1 a, :v2 b, :v3 c, :v4 d}
```

e) Remoción de un dato (no se modifican las colecciones originales, que son inmutables)

- **pop**

  Funciona con listas, vectores y colas.

  En listas y vectores se devuelve la colección sin el dato cargado por último (LIFO):

```
user=> (pop L)
(b c)
user=> (pop V)
[a b]
```

  En colas se devuelve la colección sin el dato que fue cargado primero (FIFO):

```
user=> (seq (pop Q))
(b c)
```

- **disj**

  Funciona con conjuntos. Se devuelve la colección sin el dato indicado como segundo parámetro.

```
user=> (disj HS 'a)
#{c b}
user=> (disj SS 'a)
#{b c}
```

- **dissoc**

  Funciona con mapas. Se devuelve la colección sin el dato asociado a la clave indicada como segundo parámetro.

```
user=> (dissoc HM :v1)
{:v2 b, :v3 c}
user=> (dissoc SM :v1)
{:v2 b, :v3 c}
```

f) Selección de un dato

- **peek**

  Funciona con listas, vectores y colas.

  En listas y vectores se devuelve el dato que fue cargado por último (LIFO):

```
user=> (peek L)
a
user=> (peek V)
c
```

  En colas se devuelve el dato que fue cargado primero (FIFO):

```
user=> (peek Q)
a
```

- **nth**

  Funciona con listas, vectores y colas, devolviendo el enésimo dato (se cuenta desde **0**).

```
user=> (nth L 0)
a
user=> (nth V 1)
b
user=> (nth Q 2)
c
```

- **get**

  Funciona con vectores, conjuntos y mapas, devolviendo el dato ubicado en la posición indicada por el segundo argumento (en el caso de los vectores), el dato indicado por el segundo argumento (en el caso de los conjuntos) o el dato asociado a la clave indicada por el segundo argumento (en el caso de los mapas); si no existe tal dato, **nil**. A pesar de que estas tres colecciones pueden ser usadas como si fueran funciones (sin **get**), el uso de **get** se justifica por la posibilidad de indicar como tercer argumento un valor por defecto alternativo. Por ejemplo:

```
user=> (get V 0)
a
```
Equivale a **(V 0)**

```
user=> (get HS 'a)
a
```
Equivale a **(HS 'a)**

```
user=> (get SS 'a)
a
```
Equivale a **(SS 'a)**

```
user=> (get HM :v1)
a
```
Equivale a **(HM :v1)**

```
user=> (get SM :v1)
a
```
Equivale a **(SM :v1)**

```
user=> (get V 7 'otro)
otro
user=> (get HS 'k 'otro)
otro
user=> (get SS 'k 'otro)
otro
user=> (get HM :v7 'otro)
otro
user=> (get SM :v7 'otro)
otro
```

g) Selección de múltiples datos

- **subvec**

Funciona con vectores, devolviendo los datos ubicados desde la posición indicada por el segundo argumento (inclusive). Si hay un tercer argumento, este indica hasta qué posición considerar (y el dato ubicado en esta no se devuelve). Por ejemplo:

```
user=> (subvec V 0)
[a b c]
user=> (subvec V 1)
[b c]
user=> (subvec V 0 2)
[a b]
user=> (subvec V 0 3)
[a b c]
user=> (subvec V 0 4)
IndexOutOfBoundsException   clojure.lang.RT.subvec (RT.java:1573)
```

- **select-keys**

Funciona con mapas, devolviendo los datos cuyas claves se incluyan en el segundo parámetro. Por ejemplo:

```
user=> (select-keys HM [:v1 :v3])
{:v1 a, :v3 c}
user=> (select-keys SM [:v1 :v3])
{:v1 a, :v3 c}
```

h) Combinación de colecciones

- **into**

Devuelve el resultado de aplicar la función **conj** entre la colección indicada como primer argumento y sucesivamente cada uno de los datos de la colección indicada como segundo argumento, resultando una colección del mismo tipo que la del primer argumento (o, si el primer argumento es **nil**, se devuelve una lista). Las combinaciones permitidas son:

1º↓ 2º→	L	V	Q	HS	SS	HM	SM
L	Sí	Sí	Sí	Sí	Sí	Sí	Sí
V	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Q	Sí	Sí	Sí	Sí	Sí	Sí	Sí
HS	Sí	Sí	Sí	Sí	Sí	Sí	Sí
SS	Sí	Sí	Sí	Sí	Sí	**No**	**No**
HM	**No**	**No**	**No**	**No**	**No**	Sí	Sí
SM	**No**	**No**	**No**	**No**	**No**	Sí	Sí

Por ejemplo:

```
user=> (into L '(d e f))
(f e d a b c)
user=> (into V '(d e f))
[a b c d e f]
user=> (seq (into Q '(d e f)))
(a b c d e f)
user=> (into HS '(d e f))
#{a e c b d f}
user=> (into SS '(d e f))
#{a b c d e f}
user=> (into HM '{:v4 d, :v5 e, :v6 f})
{:v2 b, :v5 e, :v1 a, :v4 d, :v3 c, :v6 f}
user=> (into SM '{:v4 d, :v5 e, :v6 f})
{:v1 a, :v2 b, :v3 c, :v4 d, :v5 e, :v6 f}
```

- Operaciones con conjuntos

```
user=> (require '[clojure.set :refer [union difference intersection]])
nil
user=> (union HS '#{c d e})
#{a e c b d}
user=> (union SS '#{c d e})
#{a b c d e}
user=> (difference SS '#{c d e})
#{a b}
user=> (intersection HS '#{c d e})
#{c}
```

- **merge**

Esta función tiene la misma aridad que **conj** (permite **0** o más argumentos) pero se comporta de manera diferente cuando se la invoca sin argumentos o cuando el primer argumento es **nil**. **merge** está diseñada para trabajar con mapas. Por ejemplo:

```
user=> (conj)
[]
user=> (merge)
nil
user=> (conj nil {:v4 'd} {:v5 'e})
({:v5 e} {:v4 d})
user=> (merge nil {:v4 'd} {:v5 'e})
{:v4 d, :v5 e}
user=> (merge HM '{:v1 d, :v2 e} '{:v0 a, :v4 b})
{:v2 e, :v1 d, :v0 a, :v4 b, :v3 c}
user=> (merge SM '{:v1 d, :v2 e} '{:v0 a, :v4 b})
{:v0 a, :v1 d, :v2 e, :v3 c, :v4 b}
```

i) Transformación de una colección (no se modifican las colecciones originales, que son inmutables)

- **empty**

  Aplicada sobre una colección, la devuelve vacía.

```
user=> (empty L)
()
user=> (empty V)
[]
user=> (empty Q)
#object[clojure.lang.PersistentQueue 0x8a60bc "clojure.lang.PersistentQueue@1"]
user=> (empty HS)
#{}
user=> (empty SS)
#{}
user=> (empty HM)
{}
user=> (empty SM)
{}
```

- **assoc**

  Se usa para cambiar un dato en un vector o en un mapa. El primer argumento es la colección, el segundo el índice (en el caso de los vectores) o la clave (en el caso de los mapas) y el tercero el nuevo dato. Por ejemplo:

```
user=> (assoc V 1 'B)
[a B c]
user=> (assoc HM :v2 'B)
{:v2 B, :v1 a, :v3 c}
```

- **update**

  Se usa para cambiar un dato en un vector o en un mapa. El primer argumento es la colección, el segundo el índice (en el caso de los vectores) o la clave (en el caso de los mapas) y el tercero la función que se aplica sobre el dato. Por ejemplo:

```
user=> (update V 1 nil?)
[a false c]
user=> (update SM :v2 clojure.string/upper-case)
{:v1 a, :v2 "B", :v3 c}
```

- **replace**

  Se usa para cambiar un conjunto de datos en un vector. El primer argumento es el conjunto de pares *antes-después* y el segundo es el vector. Por ejemplo:

```
user=> (replace '{a b, c d} V)
[b b d]
```

- **frequencies**

  Devuelve un mapa con la cantidad de veces que aparece cada dato en una colección dada.

```
user=> (frequencies '("Hola" "mundo" "Hola"))
{"Hola" 2, "mundo" 1}
```

- **shuffle**

  Si el argumento es un vector, lo devuelve mezclado. Si el argumento es un mapa, lanza una **ClassCastException**. Si es una lista, una cola o un conjunto, devuelve un vector con los datos mezclados.

```
user=> (shuffle L)
[a c b]
user=> (shuffle V)
[c b a]
user=> (shuffle SS)
[b a c]
```

- Cambio de tipo de una colección

  Siempre puede aplicarse la función **into** usando como primer argumento una colección vacía del nuevo tipo deseado y como segundo argumento la colección original. Por ejemplo:

```
user=> (into (list) V)
(c b a)
user=> (into (list) SM)
([:v3 c] [:v2 b] [:v1 a])
user=> (into [] L)
[a b c]
user=> (into [] Q)
[a b c]
user=> (into #{} SM)
#{[:v2 b] [:v1 a] [:v3 c]}
user=> (into #{} V)
#{a c b}
```

  Sin embargo, para convertir en vector o en conjunto, Clojure proporciona **vec** y **set**, respectivamente:

```
user=> (vec L)
[a b c]
user=> (vec SM)
[[:v1 a] [:v2 b] [:v3 c]]
user=> (set V)
#{a c b}
user=> (set SM)
#{[:v2 b] [:v1 a] [:v3 c]}
```

4.9. Secuencias

a) Creación de una secuencia vacía

- **concat**

```
user=> (concat)
()
user=> (class (concat))
clojure.lang.LazySeq
```

b) Creación de una secuencia a partir de una colección

- **seq**

```
user=> (seq [])
nil
user=> (seq V)
(a b c)
user=> (seq Q)
(a b c)
user=> (seq HS)
(a c b)
user=> (seq SS)
(a b c)
user=> (seq HM)
([:v2 b] [:v1 a] [:v3 c])
user=> (seq SM)
([:v1 a] [:v2 b] [:v3 c])
```

- **sequence**

```
user=> (sequence [])
()
user=> (sequence V)
(a b c)
```

- **keys**

Devuelve una secuencia con las palabras clave de un mapa. Por ejemplo:

```
user=> (keys SM)
(:v1 :v2 :v3)
```

- **vals**

Devuelve una secuencia con los datos contenidos en un mapa. Por ejemplo:

```
user=> (vals SM)
(a b c)
```

c) Creación de una secuencia a partir de constantes

- **`repeat`**

  Invocada con un único argumento, devuelve una secuencia infinita repitiéndolo. Si se usan dos argumentos, devuelve una secuencia repitiendo el segundo argumento tantas veces como indica el primer argumento. Por ejemplo:

```
user=> (take 5 (repeat 0))
(0 0 0 0 0)
user=> (repeat 5 0)
(0 0 0 0 0)
```

- **`range`**

  Con menos de dos argumentos, devuelve una secuencia, monótona creciente, de números enteros a partir del **`0`** (infinita o del tamaño indicado por el argumento). Si son dos los argumentos, la secuencia de enteros parte del valor del primer argumento y no alcanza el valor del segundo. Si son tres los argumentos, el último indica el valor del paso (*step*). Por ejemplo:

```
user=> (take 5 (range))
(0 1 2 3 4)
user=> (range 5)
(0 1 2 3 4)
user=> (range 1 6)
(1 2 3 4 5)
user=> (range 1 6 0.5)
(1 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5)
```

d) Predicado

- **`seq?`**

  Devuelve **`true`** si el argumento implementa **`ISeq`**; si no, `false`.

```
user=> (seq? L)
true
user=> (seq? V)
false
user=> (seq? Q)
false
user=> (seq? HS)
false
user=> (seq? SS)
false
user=> (seq? HM)
false
user=> (seq? SM)
false
```

e) Carga de un dato (no se modifican las secuencias originales, que son inmutables)

- **cons**

  Devuelve una secuencia con el dato indicado por el primer argumento cargado a la izquierda de la secuencia indicada por el segundo argumento:

```
user=> (class (take 5 (range)))
clojure.lang.LazySeq
user=> (cons 'd (take 5 (range)))
(d 0 1 2 3 4)
user=> (cons 'd L)
(d a b c)
user=> (cons 'd V)
(d a b c)
user=> (cons 'd Q)
(d a b c)
user=> (cons 'd HS)
(d a c b)
user=> (cons 'd SS)
(d a b c)
user=> (cons 'd HM)
(d [:v2 b] [:v1 a] [:v3 c])
user=> (cons 'd SM)
(d [:v1 a] [:v2 b] [:v3 c])
```

f) Remoción de un dato (no se modifican las secuencias originales, que son inmutables)

- **rest**

```
user=> (rest [])
()
```

Siempre se devuelve una secuencia, incluso al recibir una vacía

```
user=> (rest L)
(b c)
user=> (rest V)
(b c)
user=> (rest Q)
(b c)
user=> (rest HS)
(c b)
user=> (rest SS)
(b c)
user=> (rest HM)
([:v1 a] [:v3 c])
user=> (rest SM)
([:v2 b] [:v3 c])
```

- **next**

  Similar a **rest**, salvo cuando se recibe una secuencia vacía, en cuyo caso se devuelve **nil**.

```
user=> (next ())          Al recibir una lista vacía, se devuelve nil
nil
user=> (next L)
(b c)
```

- **butlast**

  Funciona como **next**, pero viendo la secuencia de derecha a izquierda. Por ejemplo:

```
user=> (butlast #{})      Al recibir un conjunto vacío, se devuelve nil
nil
user=> (butlast L)
(a b)
user=> (butlast V)
(a b)
user=> (butlast Q)
(a b)
```

g) Selección de un dato

- **first**

  Devuelve el dato que aparece primero a la izquierda:

```
user=> (first [])         Al recibir un vector vacío, se devuelve nil
nil
user=> (first L)          El dato que fue cargado por último en la lista
a
user=> (first V)          El dato que primero fue cargado en el vector
a
user=> (first Q)          El dato que primero fue cargado en la cola
a
user=> (first HS)         En un conjunto no ordenado aquí puede haber cualquier dato
a
user=> (first SS)         El dato que queda primero después del ordenamiento
a
user=> (first HM)         En un mapa no ordenado aquí puede haber cualquier dato
[:v2 b]
user=> (first SM)         El dato que queda primero después del ordenamiento por clave
[:v1 a]
```

- **last**

  Devuelve el dato que aparece primero a la derecha:

  ```
  user=> (last [])        Al recibir un vector vacío, se devuelve nil
  nil
  user=> (last L)         El dato que primero fue cargado en la lista
  c
  user=> (last V)         El dato que fue cargado por último en el vector
  c
  user=> (last Q)         El dato que fue cargado por último en la cola
  c
  user=> (last HS)        En un conjunto no ordenado aquí puede haber cualquier dato
  b
  user=> (last SS)        El dato que queda último después del ordenamiento
  c
  user=> (last HM)        En un mapa no ordenado aquí puede haber cualquier dato
  [:v3 c]
  user=> (last SM)        El dato que queda último después del ordenamiento por clave
  [:v3 c]
  ```

h) Operaciones dobles (no se modifican las secuencias originales, que son inmutables)

- **ffirst**

  ```
  user=> (first (first '((a b) [c d] #{e f})))
  a
  user=> (ffirst '((a b) [c d] #{e f}))
  a
  ```

- **fnext / second**

  ```
  user=> (first (next '((a b) [c d] #{e f})))
  [c d]
  user=> (fnext '((a b) [c d] #{e f}))  user=> (second '((a b) [c d] #{e f}))
  [c d]                                 [c d]
  ```

- **nfirst**

  ```
  user=> (next (first '((a b) [c d] #{e f})))
  (b)
  user=> (nfirst '((a b) [c d] #{e f}))
  (b)
  ```

- **nnext**

  ```
  user=> (next (next '((a b) [c d] #{e f})))
  (#{e f})
  user=> (nnext '((a b) [c d] #{e f}))
  (#{e f})
  ```

i) Remoción de múltiples datos (no se modifican las secuencias originales, que son inmutables)

- **drop**

```
user=> (drop 2 L)
(c)
user=> (drop 2 V)
(c)
user=> (drop 0 V)          Siempre se devuelve una secuencia...
(a b c)
user=> (drop 4 V)          ... incluso una vacía
()
```

- **drop-last**

```
user=> (drop-last 2 L)
(a)
user=> (drop-last 2 V)
(a)
user=> (drop-last 0 V)     Siempre se devuelve una secuencia...
(a b c)
user=> (drop-last 4 V)     ... incluso una vacía
()
```

- **nthrest**

```
user=> (nthrest V 0)       No siempre se devuelve una secuencia
[a b c]
user=> (nthrest V 1)
(b c)
user=> (nthrest V 2)
(c)
user=> (nthrest V 3)
()
```

- **nthnext**

```
user=> (nthnext V 0)
(a b c)
user=> (nthnext V 1)
(b c)
user=> (nthnext V 2)
(c)
user=> (nthnext V 3)
nil                        No siempre se devuelve una secuencia
```

j) Selección de múltiples datos

- **take**

```
user=> (take 5 (range))
(0 1 2 3 4)
user=> (take 0 V)
()
user=> (take 1 V)
(a)
user=> (take 2 V)
(a b)
user=> (take 3 V)
(a b c)
user=> (take 4 V)
(a b c)
```

- **take-last**

```
user=> (take-last 5 (range 20))
(15 16 17 18 19)
user=> (take-last 0 V)
nil
user=> (take-last 1 V)
(c)
user=> (take-last 2 V)
(b c)
user=> (take-last 3 V)
(a b c)
user=> (take-last 4 V)
(a b c)
```

- **take-nth**

El primer argumento indica cada cuántos datos de la secuencia original (el segundo argumento) se toman los datos de la secuencia a devolver. Por ejemplo:

```
user=> (take-nth 4 (range 30))
(0 4 8 12 16 20 24 28)
user=> (take-nth 3 (range 30))
(0 3 6 9 12 15 18 21 24 27)
```

- **re-seq**

Devuelve en una secuencia las subcadenas de una cadena (segundo argumento) que coinciden con una expresión regular (primer argumento). Por ejemplo:

```
user=> (re-seq #"\w+" "Hola, mundo!")
("Hola" "mundo")
user=> (re-seq #"\d+" "12:35-14:30")
("12" "35" "14" "30")
```

k) Tamaño de una secuencia

```
user=> (count L)
3
user=> (count V)
3
user=> (count Q)
3
user=> (count HS)
3
user=> (count SS)
3
user=> (count HM)
3
user=> (count SM)
3
```

l) Combinación de secuencias

- **concat**

  Devuelve la concatenación de dos o más secuencias de cualquier tipo

```
user=> (concat L '(d e f))
(a b c d e f)
user=> (concat V '[d e f])
(a b c d e f)
user=> (concat Q '[d e f])
(a b c d e f)
user=> (concat Q '#{d e f})
(a b c e d f)
user=> (concat HS '[d e f])
(a c b d e f)
user=> (concat SS '[d e f])
(a b c d e f)
user=> (concat HM '[d e f])
([:v2 b] [:v1 a] [:v3 c] d e f)
user=> (concat SM '[d e f])
([:v1 a] [:v2 b] [:v3 c] d e f)
user=> (concat V HM)
(a b c [:v2 b] [:v1 a] [:v3 c])
user=> (concat V SM)
(a b c [:v1 a] [:v2 b] [:v3 c])
```

m) Transformación de una secuencia (no se modifican las secuencias originales, que son inmutables)

- **reverse**

```
user=> (reverse L)
(c b a)
user=> (reverse V)
(c b a)
user=> (reverse HS)
(b c a)
user=> (reverse SM)
([:v3 c] [:v2 b] [:v1 a])
```

- **sort**

```
user=> (sort (list 4 2 5 3 1))
(1 2 3 4 5)
user=> (sort '[b c a e d])
(a b c d e)
user=> (sort HS)
(a b c)
```

- **flatten**

Devuelve una secuencia con los datos escalares contenidos en colecciones secuenciales (listas, vectores o colas) anidadas. Por ejemplo:

```
user=> (flatten '[(1) (2 3) (4 (5 [6 7 (8)]))])
(1 2 3 4 5 6 7 8)
```

- **partition**

Devuelve una secuencia con los datos (último argumento) separados en particiones del tamaño indicado por el primer argumento (y avanzando según el segundo, si este se indica). Por ejemplo:

```
user=> (partition 2 "ABCDEF")
((\A \B) (\C \D) (\E \F))
user=> (partition 3 1 "ABCDEF")
((\A \B \C) (\B \C \D) (\C \D \E) (\D \E \F))
user=> (partition 3 1 '[A B C D E F])
((A B C) (B C D) (C D E) (D E F))
```

- **replace** (Obs: ya se vio cómo funciona con vectores)

```
user=> (replace '{a b, c d} L)
(b b d)
user=> (replace '{a b, c d} Q)
(b b d)
user=> (replace '{a b, c d} SS)
(b b d)
user=> (replace '{[:v1 a] [:v1 b], [:v3 c] [:v3 d]} SM)
([:v1 b] [:v2 b] [:v3 d])
```

5. FUNCIONES DE ORDEN SUPERIOR

Se denomina *funciones de orden superior* a las funciones que aceptan funciones como argumentos y/o devuelven otras funciones como resultados.

5.1. Aplicación exacta

- **apply**

  Invoca una función de aridad *n* (el primer argumento) con los *n* datos contenidos en el segundo:

```
user=> (apply str '(a b c))
"abc"

user=> (str 'a 'b 'c)
"abc"

user=> (defn suma2 [a b] (+ a b))
#'user/suma2

user=> (defn suma3 [a b c] (+ a b c))
#'user/suma3

user=> (defn calc [f d] (apply f d))
#'user/calc

user=> (calc suma2 '(1 2))
3

user=> (calc suma3 '(1 2 3))
6
```

5.2. Mapeo

- **map**

  Devuelve la secuencia formada por el resultado de aplicar una función (el primer argumento) al conjunto formado por los primeros elementos de los demás argumentos, seguido del resultado de aplicarla al conjunto formado por los segundos elementos, y así sucesivamente:

```
user=> (map + '(1 2 3) '(4 5 6) '(7 8 9))
(12 15 18)

user=> (map inc [1 2 3])
(2 3 4)

user=> (map char "hola")
(\h \o \l \a)

user=> (map (fn [x] (inc (first x))) '((1 7)(3 9)(5 2)))
(2 4 6)
```

- **mapv**

  Funciona igual que **map**, pero su resultado es un vector:

```
user=> (mapv (fn [x] (inc (first x))) '((1 7)(3 9)(5 2)))
[2 4 6]
```

5.3. Aplicación parcial

- **partial**

  Devuelve una función de menor aridad que la indicada como primer argumento, ya que los demás argumentos se utilizan ("se fijan") en la función recibida. Por ejemplo:

```
user=> (map (partial suma2 1) [4 5 6 7 8])
(5 6 7 8 9)
```

5.4. Reducción

- **reduce**

  Aplica una función de aridad **2** (el primer argumento) al primero y al segundo dato contenidos en el segundo argumento, luego al resultado y al tercer dato contenido en el segundo argumento, y así sucesivamente, hasta agotar todos los datos. Por ejemplo:

```
user=> (reduce (fn [a b] (+ a b)) [1 2 3 4])
10

user=> (reduce (fn [accum x] (assoc accum (keyword x) (str x \- (rand-int 100))))
               {}
               ["hi" "hello" "bye"])
{:hi "hi-29" :hello "hello-42" :bye "bye-10"}

user=> (map (partial reduce +) [[1 2 3 4] [5 6 7 8]])
(10 26)
```

5.5. Selección de múltiples datos

- **filter**

  Devuelve una secuencia seleccionando los datos del segundo argumento que cumplan con el predicado indicado como primer argumento. Por ejemplo:

```
user=> (filter pos? [-3 -2 -1 0 1 -4 3])
(1 3)

user=> (filter (fn [x] (> x 0)) [-3 -2 -1 0 1 -4 3])
(1 3)
```

- **take-while**

  Devuelve una secuencia seleccionando los datos del segundo argumento, leídos desde la izquierda, mientras cumplan con el predicado indicado como primer argumento. Por ejemplo:

```
user=> (take-while neg? [-3 -2 -1 0 1 -4 3])
(-3 -2 -1)
```

5.6. Remoción de múltiples datos

- **remove**

  Devuelve una secuencia eliminando los datos del segundo argumento que cumplan con el predicado indicado como primer argumento. Por ejemplo:

```
user=> (remove pos? [-3 -2 -1 0 1 -4 3])
(-3 -2 -1 0 -4)
```

- **drop-while**

  Devuelve una secuencia eliminando los datos del segundo argumento, leídos desde la izquierda, mientras cumplan con el predicado indicado como primer argumento. Por ejemplo:

```
user=> (drop-while neg? [-3 -2 -1 0 1 -4 3])
(0 1 -4 3)
```

5.7. Predicados

- **every?**

  Devuelve **true** si el primer argumento (un predicado) es **true** para todos los datos del segundo argumento; si no, **false**. Por ejemplo:

```
user=> (every? pos? [1 2 0 4 5])
false
user=> (every? pos? [1 2 3 4 5])
true
```

- **not-every?**

  Devuelve **false** si el primer argumento (un predicado) es **true** para todos los datos del segundo argumento; si no, **true**. Por ejemplo:

```
user=> (not-every? pos? [1 2 3 4 5])
false
user=> (not-every? pos? [1 2 0 4 5])
true
```

- **not-any?**

  Devuelve **false** si el primer argumento (un predicado) es **true** para algún dato del segundo argumento; si no, **true**. Por ejemplo:

```
user=> (not-any? pos? [0 0 0 0 0])
true
user=> (not-any? pos? [0 0 1 0 0])
false
```

5.8. Composición

- **comp**

  Devuelve una función que es la composición de los argumentos recibidos. Por ejemplo:

```
user=> (first (next (next L)))
c

user=> ((comp first next next) L)
c

user=> ((comp #(reduce (fn [a b] (conj a b)) %) #(conj % ())) L)
(c b a)
```

- **iterate**

  Devuelve una secuencia infinita formada por el segundo argumento, seguido del primer argumento aplicado sobre el segundo argumento, seguido del primer argumento aplicado sobre el valor anterior, y así sucesivamente. Por ejemplo:

```
user=> (defn square [x] (* x x))

user=> (take 5 (iterate square 2))
(2 4 16 256 65536)
```

5.9. Ordenamiento

- **sort**

  Devuelve una secuencia con los datos del segundo argumento, ordenados usando una función de comparación indicada como primer argumento. Por ejemplo:

```
user=> (sort (fn [a b] (Integer/signum (- b a))) [1 5 2 4 3 2 5])
(5 5 4 3 2 2 1)
```

- **sort-by**

  Devuelve una secuencia con los datos del último argumento, ordenados según los resultados de la aplicación de una función indicada como primer argumento y usando, opcionalmente, una función de comparación indicada como segundo argumento. Por ejemplo:

```
user=> (sort second [[:a 7], [:c 13], [:b 21]])
ArityException Wrong number of args (2) passed to: core/second

user=> (sort-by second [[:a 7], [:c 13], [:b 21]])
([:a 7] [:c 13] [:b 21])
```

5.10. Desagregación

- **partition-by**

  Devuelve una secuencia con los valores del segundo argumento particionados según el primero:

```
user=> (partition-by #(> % 3) [1 2 3 4 5 1 6 1 2 3])
((1 2 3) (4 5) (1) (6) (1 2 3))
```

- **group-by**

  Devuelve una secuencia con los valores del segundo argumento agrupados según el primero:

```
user=> (group-by #(> % 3) [1 2 3 4 5 1 6 1 2 3])
{false [1 2 3 1 1 2 3], true [4 5 6]}
```

6. MACROS PREDEFINIDAS

Gracias a su homoiconicidad, Clojure ofrece la posibilidad de escribir *macros* (formas que al expandirse generan código). Además, también proporciona un gran número de ellas ya predefinidas (como **defn**, que ya ha sido usada aquí muchas veces). Algunas de las principales son las siguientes:

- **cond**

  Evalúa una serie de condiciones, hasta que alguna sea verdadera, y devuelve el resultado de la expresión ubicada a continuación de esa condición. Si ninguna es verdadera, devuelve **nil**.

```
user=> (defn cond-test [n]
         (cond
           (= n 1) (str "n is " 1)
           (> n 3) "n is over 3"
           true "n is other"))
#'user/cond-test

user=> (cond-test 2)
"n is other"

user=> (cond-test 1)
"n is 1"

user=> (cond-test 5)
"n is over 3"
```

- **case**

  Su estructura difiere de la de **cond**, ya que no evalúa condiciones, sino que compara constantes:

```
user=> (defn case-test [n]
         (case n
            1 (str "n is " 1)
            2 "n is 2"
            "n is other"))
#'user/case-test

user=> (case-test 1)
"n is 1"

user=> (case-test 2)
"n is 2"

user=> (case-test 3)
"n is other"
```

- **and**

  Evalúa una serie de expresiones, de izquierda a derecha, y devuelve **true** si ninguna es **false**.

```
user=> (and (= (count L) 3) (= 'a (first L)))
true

user=> (and (= (count L) 3) (= 'b (first L)))
false
```

- **or**

  Evalúa una serie de expresiones, de izquierda a derecha, y devuelve **true** si alguna es **true**.

```
user=> (or (= (count L) 3) (= 'b (first L)))
true
user=> (or (= (count L) 4) (= 'b (first L)))
false
```

- **for**

  Devuelve la secuencia correspondiente a una *lista por comprensión* definida a partir de un vector y otro dato. Una lista por comprensión corresponde en matemática a un conjunto de valores indicados de manera abreviada, por ejemplo: *{x | x ϵ [-10..10] y x es par}*

```
user=> (for [x (range -10 11) :when (even? x)] x)
(-10 -8 -6 -4 -2 0 2 4 6 8 10)
user=> (for [x '(0 1 2 3 4)] (+ x 1))
(1 2 3 4 5)
user=> (for [x [0 1 2 3 4 5] :let [y (* x 3)]] y)
(0 3 6 9 12 15)
user=> (for [x ['a 'b 'c] y [1 2 3]] [x y])
([a 1] [a 2] [a 3] [b 1] [b 2] [b 3] [c 1] [c 2] [c 3])
```

- **->**

  Aplica secuencialmente una serie de transformaciones sobre el primer elemento indicado, el cual es insertado en la primera posición (donde opcionalmente se usan triples comas):

```
user=> (-> () (conj ,,, 1) (conj ,,, 2) (conj ,,, 3))
(3 2 1)
user=> (-> {:pelo 'rubio, :edad 49}
           (assoc ,,, :pelo 'gris)
           (update ,,, :edad inc))
{:pelo gris, :edad 50}
```

- **->>**

  Aplica secuencialmente una serie de transformaciones sobre el primer elemento indicado, el cual es insertado en la última posición (donde opcionalmente se usan triples comas):

```
user=> (->> () (cons 1 ,,,) (cons 2 ,,,) (cons 3 ,,,))
(3 2 1)
user=> (->> ["Japan" "China" "Korea"]
            (map clojure.string/upper-case ,,,)
            (map #(str "Hello " %) ,,,))
("Hello JAPAN" "Hello CHINA" "Hello KOREA")
```

- **as->**

  Aplica secuencialmente una serie de transformaciones sobre el primer elemento indicado, el cual es insertado en la posición indicada por su alias (el segundo elemento):

```
user=> (as-> [] v (conj v 1) (cons 2 v) (conj v 3))
(3 2 1)
```

- **lazy-seq**

  Devuelve una secuencia perezosa. Por ejemplo, esta implementación de la función **tirar-dado** no funciona porque la secuencia que genera recursivamente es infinita y se desborda la pila:

```
user=> (defn tirar-dado [] (cons (+ 1 (rand-int 6)) (tirar-dado)))
#'user/tirar-dado
user=> (take 30 (tirar-dado))
StackOverflowError   clojure.lang.RT.longCast (RT.java:1267)
```

  En cambio, esta implementación de la función **tirar-dado** sí funciona porque la secuencia que genera recursivamente es perezosa y solo se la produce a medida que se la va requiriendo:

```
user=> (defn tirar-dado [] (lazy-seq (cons (+ 1 (rand-int 6)) (tirar-dado))))
#'user/tirar-dado
user=> (take 30 (tirar-dado))
(2 4 4 3 2 6 6 4 5 2 6 6 3 4 4 5 5 3 2 1 3 1 2 2 5 1 4 5 3 1)
```

- **time**

  Evalúa una expresión, muestra el tiempo que tomó hacerlo y devuelve el resultado de la evaluación. Por ejemplo:

```
user=> (time (take 30 (tirar-dado)))
"Elapsed time: 4.079701 msecs"
(2 3 3 2 5 3 2 2 5 3 5 6 6 2 1 5 1 1 6 3 1 5 6 1 4 3 4 5 6 5)
```

- **with-out-str**

  Evalúa expresiones y devuelve una cadena con la concatenación de sus salidas. Por ejemplo:

```
user=> (def transcurrio (with-out-str (time (last (range 10000)))))
#'user/transcurrio
user=> transcurrio
"\"Elapsed time: 5.765558 msecs\"\r\n"
```

- **declare**

  Su funcionamiento es prácticamente igual al de la forma especial **def**, pero permite indicar varios símbolos a la vez. Además, pone en evidencia que el autor tuvo la intención de declarar algo que se definirá luego (se utiliza para llevar a cabo *forward declarations*). Por ejemplo:

```
user=> (defn operar [a b] (operacion a b))
CompilerException java.lang.RuntimeException: Unable to resolve symbol:
operacion in this context, compiling:(NO_SOURCE_PATH:94:19)
user=> (declare operacion)
#'user/operacion
user=> (defn operar [a b] (operacion a b))
#'user/operar
user=> (defn operacion [a b] (* a b))
#'user/operacion
user=> (operar 2 3)
6
```