무료 전자 책

배우기

clojure

Free unaffiliated eBook created from
Stack Overflow contributors.

#clojure

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: clojure

It is an unofficial and free clojure ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official clojure.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

1: Clojure

Clojure **Lisp** .

. Clojure . . . . Clojure .

Clojure . . . .

Clojure .

- (STM)
- (atom, agent)
- ( ),
- ()

(,) Clojure . `core.typed` `core.async` `core.logic` .

. JVM Java , Common Language Runtime ClojureCLR JavaScript ( ) ClojureScript . JVM .

1.8		2016-01-19
1.7	1.7	2015-06-30
1.6	1.6	2014-03-25
1.5.1	1.5.1	2013-03-10
1.4	1.4	2012-04-15
1.3	1.3	2011-09-23
1.2.1		2011-03-25
1.2		2010-08-19
1.1		2010-01-04
1.0		2009-05-04

Examples

1 : Leiningen

JDK 6 .

Clojure Clojure Leiningen **`lein repl`** REPL .

```
curl https://raw.githubusercontent.com/technomancy/leiningen/stable/bin/lein > ~/bin/lein
export PATH=$PATH:~/bin
chmod 755 ~/bin/lein
```

OS X

Linux macOS .

Homebrew

```
brew install leiningen
```

MacPorts

Clojure .

```
sudo port -R install clojure
```

Clojure `Leiningen`

```
sudo port -R install leiningen
```

```
lein self-install
```

Windows

.

2 :

JRE 6 .

Clojure JAR JVM . Clojure .

1. http://clojure.org Clojure .
2. ZIP .
3.

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

`clojure-1.8.0.jar` JAR .

REPL (:) `rlwrap` . `rlwrap java -cp clojure-1.8.0.jar clojure.main`

3 :

JDK 7 .

Boot Clojure . Clojure . ( ) .

`PATH` Clojure REPL `boot repl` .

", !" REPL

Clojure Clojure REPL (read-eval-print-loop) . Clojure , , , .

Clojure REPL . . :

```
(println "Hello, world!")
```

`Enter □□ □□□□ . Hello, world! Hello, world! nil .`

clojure REPL . : http://www.tryclj.com/ .

Leiningen .

```
lein new <project-name>
```

Clojure `<project-name>` Leiningen . Leiningen . "app" jar ( ) . .

```
lein new app <project-name>
```

app-template .

```
lein run
```

`Hello, World! Hello, World!` .

jar .

```
lein uberjar
```

", !"

: Boot . .

Boot **shebang** (#!) Clojure . (" " `hello.clj` `hello.clj`).

```
#!/usr/bin/env boot

(defn -main [& args]
  (println "Hello, world!"))
```

(`chmod +x hello.clj` `chmod +x hello.clj`).
... (`./hello.clj`) .

"Hello, world!" . .

( )

```
boot -d seancorfield/boot-new new -t app -n <appname>
```

`boot-new` https://github.com/seancorfield/boot-new `app` ( ). Clojure `<appname>` .
README .

: `boot run` . `build.boot` README .

Clojure : https://riptutorial.com/ko/clojure/topic/827/clojure-

2: clj-time

.

project.clj : dependencies [clj-time "<version_number>"] .

Examples

```
(clj-time/date-time 2017 1 20)
```

2017 1 20 00:00:00 Joda .

, .

```
(clj-time/date-time year month date hour minute second millisecond)
```

-

```
(require '[clj-time.core :as t])

(def example-time (t/date-time 2016 12 5 4 3 27 456))

(t/year example-time) ;; 2016
(t/month example-time) ;; 12
(t/day example-time) ;; 5
(t/hour example-time) ;; 4
(t/minute example-time) ;; 3
(t/second example-time) ;; 27
```

```
(require '[clj-time.core :as t])

(def date1 (t/date-time 2016 12 5))
(def date2 (t/date-time 2016 12 6))

(t/equal? date1 date2) ;; false
(t/equal? date1 date1) ;; true

(t/before? date1 date2) ;; true
(t/before? date2 date1) ;; false

(t/after? date1 date2) ;; false
(t/after? date2 date1) ;; true
```

.

```
(require '[clj-time.core :as t])

(def date1 (t/date-time 2016 11 5))
(def date2 (t/date-time 2016 12 5))
```

```
(def test-date1 (t/date-time 2016 12 20))
(def test-date2 (t/date-time 2016 11 15))

(t/within? (t/interval date1 date2) test-date1) ;; false
(t/within? (t/interval date1 date2) test-date2) ;; true
```

interval . , interval . :

```
(t/within? (t/interval date1 date2) date2) ;; false
(t/within? (t/interval date1 date2) date1) ;; true
```

joda -

clj-time.coerce - joda (clj-time.core / date-time) . **Java long** format, **String** , **Date** , **SQL Date** .

from .

```
(require '[clj-time.coerce :as c])

(def string-time "1990-01-29")
(def epoch-time 633571200)
(def long-time 633551400)

(c/from-string string-time) ;; #<DateTime 1990-01-29T00:00:00.000Z>
(c/from-epoch epoch-time) ;; #<DateTime 1990-01-29T00:00:00.000Z>
(c/from-long 633551400) ;; #<DateTime 1990-01-29T00:00:00.000Z>
```

- -

cljs-time / . , , , .

```
(require '[clj-time.core :as t])

(def example-date (t/date-time 2016 1 1)) ;; #<DateTime 2016-01-01T00:00:00.000Z>

;; Addition
(t/plus example-date (t/months 1))        ;; #<DateTime 2016-02-01T00:00:00.000Z>
(t/plus example-date (t/years 1))         ;; #<DateTime 2017-01-01T00:00:00.000Z>

;; Subtraction
(t/minus example-date (t/days 1))          ;; #<DateTime 2015-12-31T00:00:00.000Z>
(t/minus example-date (t/hours 12))        ;; #<DateTime 2015-12-31T12:00:00.000Z>
```

clj-time : https://riptutorial.com/ko/clojure/topic/9127/clj-time

3: Clojure destructuring

Examples

.

```
(def my-vec [1 2 3])
```

let .

```
(let [[x y] my-vec]
 (println "first element:" x ", second element: " y))
;; first element: 1 , second element: 2
```

.

```
(def my-map {:a 1 :b 2 :c 3})
```

, let .

```
(let [{x :a y :c} my-map]
  (println ":a val:" x ", :c val: " y))
;; :a val: 1 , :c val: 3
```

.

.

```
(let [{:keys [a c]} my-map]
  (println ":a val:" a ", :c val: " c))
;; :a val: 1 , :c val: 3
```

.

```
(let [{:strs [foo bar]} {"foo" 1 "bar" 2}]
  (println "FOO:" foo  "BAR: " bar ))
;; FOO: 1 BAR: 2
```

.

```
(let [{:syms [foo bar]} {'foo 1 'bar 2}]
  (println "FOO:" foo "BAR:" bar))
;; FOO: 1 BAR: 2
```

.

```
(def data
```

```
  {:foo {:a 1
         :b 2}
   :bar {:a 10
         :b 20}})

(let [{{:keys [a b]} :foo
       {a2 :a b2 :b} :bar} data]
  [a b a2 b2])
;; => [1 2 10 20]
```

.

```
(def my-vec [1 2 3 4 5 6])
```

. .

```
(let [[x y z & remaining] my-vec]
 (println "first:" x ", second:" y "third:" z "rest:" remaining))
;= first: 1 , second: 2 third: 3 rest: (4 5 6)
```

.

```
(def my-vec [[1 2] [3 4]])

(let [[[a b][c d]] my-vec]
  (println a b c d))
;; 1 2 3 4
```

. :

```
(def my-map {:a 3 :b 4})
(let [{a :a
       b :b
       :keys [c d]
       :or {a 1
            c 2}} my-map]
  (println a b c d))
  ;= 3 4 2 nil
```

fn params

Destructurling fn param .

```
(defn my-func [[_ a b]]
  (+ a b))

(my-func [1 2 3]) ;= 5
(my-func (range 5)) ;= 3
```

destructuring param `& rest` .

```
(defn my-func2 [& [_ a b]]
```

```
  (+ a b))

(my-func2 1 2 3) ;= 5
(apply my-func2 (range 5)) ;= 3
```

Destructuring .

```
(def my-vec [:a 1 :b 2])
(def my-lst '("smthg else" :c 3 :d 4))

(let [[& {:keys [a b]}] my-vec
      [s & {:keys [c d]} my-lst]
  (+ a b c d)) ;= 10
```

.

```
(defn my-func [a b & {:keys [c d] :or {c 3 d 4}}]
  (println a b c d))

(my-func 1 2) ;= 1 2 3 4
(my-func 3 4 :c 5 :d 6) ;= 3 4 5 6
```

Destructuring . (`a` = `"a"` , b = `"b"` & c) .

		data /
vec	`(let [[abc] data ...)`	`["a" "b" "c"]`
vec	`(let [[[ab] [cd]] data ...)`	`[["a" "b"] ["c" "d"]]`
map	`(let [{a :ab :bc :c} data ...)`	`{:a "a" :b "b" :c "c"}`
- :	`(let [{:keys [abc]} data ...)`	.

:

- `:or` `:or` nil
- `& rest seq` `rest` ,
- .
- (_).

Key

. destructuring .

.

```
(def john {:lastname "McCarthy" :firstname "John" :country "USA"})
```

:

```
(let [{lastname :lastname firstname :firstname country :country} john]
    (str firstname " " lastname ", " country))
;;"John McCarthy, USA"
```

3 (, ,) .

, `:keys` ( ) bind :

```
(let [{:keys [firstname lastname country]} john]
    (str firstname " " lastname ", " country))
;;"John McCarthy, USA"
```

(`:syms`) (`:strs`) `:strs`

```
;; using strings as keys
(def john {"lastname" "McCarthy" "firstname" "John" "country" "USA"})
;;#'user/john

;; destructuring string-keyed map
(let [{:strs [lastname firstname country]} john]
    (str firstname " " lastname ", " country))
;;"John McCarthy, USA"

;; using symbols as keys
(def john {'lastname "McCarthy" 'firstname "John" 'country "USA"})

;; destructuring symbol-keyed map
(let [{:syms [lastname firstname country]} john]
    (str firstname " " lastname ", " country))
;;"John McCarthy, USA"
```

.

```
(defn print-some-items
   [[a b :as xs]]
  (println a)
  (println b)
  (println xs))

(print-some-items [2 3])
```

.

```
2
3
[2 3]
```

`2 3 a b. [2 3] xs .`

Clojure destructuring : https://riptutorial.com/ko/clojure/topic/1786/clojure-destructuring

4: clojure.core

clojure . org.clojure .

Examples

```
(defn x [a b]
        (* a b)) ;; public function

=> (x 3 2) ;; 6
=> (x 0 9) ;; 0

(defn- y [a b]
        (+ a b)) ;; private function

=> (x (y 1 2) (y 2 3)) ;; 15
```

Assoc - /

.

.

```
(def userData {:name "Bob" :userID 2 :country "US"})
(assoc userData :age 27)            ;; { :name "Bob" :userID 2 :country "US" :age 27}
```

.

```
(assoc userData :name "Fred")       ;; { :name "Fred" :userID 2 :country "US" }
(assoc userData :userID 3 :age 27)  ;; {:name "Bob" :userID 3 :country "US" :age 27}
```

Y .

```
(assoc [3 5 6 7] 2 10)              ;; [3 5 10 7]
(assoc [1 2 3 4] 6 6)               ;; java.lang.IndexOutOfBoundsException
```

Clojure

. `(2>1)` `(> 2 1)` . clojure .

1.

```
(> 2 1) ;; true
(> 1 2) ;; false
```

2.

```
(< 2 1) ;; false
```

3.

```
(>= 2 1) ;; true
(>= 2 2) ;; true
(>= 1 2) ;; false
```

4.

```
(<= 2 1) ;; false
(<= 2 2) ;; true
(<= 1 2) ;; true
```

5.

```
(= 2 2) ;; true
(= 2 10) ;; false
```

6.

```
(not= 2 2) ;; false
(not= 2 10) ;; true
```

Dissoc -

- . .

```
(dissoc {:a 1 :b 2} :a) ;; {:b 2}
```

.

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

clojure.core : https://riptutorial.com/ko/clojure/topic/9585/clojure-core

5: clojure.spec

- :: . user :: foo user / foo .
- # : # - map-literal

Clojure 1.9 clojure / .

, , .

Examples

. .

```
(clojure.spec/valid? odd? 1)
;;=> true

(clojure.spec/valid? odd? 2)
;;=> false
```

`valid?` spec true false .

:

```
(s/valid? #{:red :green :blue} :red)
;;=> true
```

fdef :

.

```
(defn nat-num-count [nums] (count (remove neg? nums)))
```

.

```
(clojure.spec/fdef nat-num-count
        :args (s/cat :nums (s/coll-of number?))
        :ret integer?
        :fn #(<= (:ret %) (-> % :args :nums count)))
```

`:args` . spec `:args` arities . `:ret` .

`:fn` `:args` `:ret` . test.check . , : `:args` ( ) `:ret` ( ) .

`clojure.spec/def` . `def` namespace-qualified .

```
(clojure.spec/def ::odd-nums odd?)
;;=> :user/odd-nums
```

```
(clojure.spec/valid? ::odd-nums 1)
;;=> true
(clojure.spec/valid? ::odd-nums 2)
;;=> false
```

Clojure .

`::odd-nums` user :user/odd-nums . :: .

`valid?` `valid?` , .

clojure.spec / & clojure.spec /

`clojure.spec/and` & `clojure.spec/or` .

```
(clojure.spec/def ::pos-odd (clojure.spec/and odd? pos?))

(clojure.spec/valid? ::pos-odd 1)
;;=> true

(clojure.spec/valid? ::pos-odd -3)
;;=> false
```

`or` . an `or` spec . .

```
(clojure.spec/def ::big-or-small (clojure.spec/or :small #(< % 10) :big #(> % 100)))

(clojure.spec/valid? ::big-or-small 1)
;;=> true

(clojure.spec/valid? ::big-or-small 150)
;;=> true

(clojure.spec/valid? ::big-or-small 20)
;;=> false
```

`or` .

```
(clojure.spec/conform ::big-or-small 5)
;; => [:small 5]
```

.

```
(clojure.spec/def ::name string?)
(clojure.spec/def ::age pos-int?)
(clojure.spec/def ::occupation string?)

(defrecord Person [name age occupation])

(clojure.spec/def ::person (clojure.spec/keys :req-un [::name ::age ::occupation]))

(clojure.spec/valid? ::person (->Person "john doe" 25 "programmer"))
;;=> true
```

```
(clojure.spec/valid? ::person (->Person "john doe" "25" "programmer"))
;;=> false
```

, . .

.

```
(clojure.spec/def ::name string?)
(clojure.spec/def ::age pos-int?)
(clojure.spec/def ::occupation string?)

(clojure.spec/def ::person (clojure.spec/keys :req [::name ::age ::occupation]))

(clojure.spec/valid? ::person {::name "john" ::age 25 ::occupation "programmer"})
;; => true
```

`:req` . `:opt` , .

. . `clojure.spec` . req : opt : `:req-un` and `:opt-un` . .

```
(clojure.spec/def ::name string?)
(clojure.spec/def ::age pos-int?)
(clojure.spec/def ::occupation string?)

(clojure.spec/def ::person (clojure.spec/keys :req-un [::name ::age ::occupation]))

(clojure.spec/valid? ::person {:name "john" :age 25 :occupation "programmer"})
;; => true
```

`:req-un` . clojure.spec .

. :

```
(clojure.spec/def ::name string?)
(clojure.spec/def ::age pos-int?)
(clojure.spec/def ::occupation string?)

(clojure.spec/def ::person (clojure.spec/keys :req [::name ::age ::occupation]))

(clojure.spec/valid? ::person #:user{:name "john" :age 25 :occupation "programmer"})
;;=> true
```

`#:` reader . . .

. coll-of . .

```
(clojure.spec/valid? (clojure.spec/coll-of int?) [1 2 3])
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int?) '(1 2 3))
;; => true
```

/ . `:kind` .

```
(clojure.spec/valid? (clojure.spec/coll-of int? :kind vector?) [1 2 3])
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int? :kind vector?) '(1 2 3))
;; => false
```

false.

```
(clojure.spec/valid? (clojure.spec/coll-of int? :kind list?) '(1 2 3))
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int? :kind set?) #{1 2 3})
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int? :kind set?) #{1 "2" 3})
;; => false
```

int false.

.

```
(clojure.spec/valid? (clojure.spec/coll-of int? :kind vector? :count 3) [1 2 3])
;; => true

    (clojure.spec/valid? (clojure.spec/coll-of int? :kind vector? :count 3) [1 2])
;; => false

(clojure.spec/valid? (clojure.spec/coll-of int? :min-count 3 :max-count 5) [1 2 3])
;; => true

    (clojure.spec/valid? (clojure.spec/coll-of int? :min-count 3 :max-count 5) [1 2])
;; => false
```

`:distinct` :

```
(clojure.spec/valid? (clojure.spec/coll-of int? :distinct true) [1 2])
;; => true

(clojure.spec/valid? (clojure.spec/coll-of int? :distinct true) [2 2])
;; => false
```

`coll-of` . . `every` `coll-of` , ' . . .

```
(clojure.spec/valid? (clojure.spec/every int? :distinct true) [1 2 3 4 5])
;; => true
```

`map-of` `coll-of` . .

```
(clojure.spec/valid? (clojure.spec/map-of keyword? string?) {:red "red" :green "green"})
;; => true
```

`coll-of` `map-of` / . . `coll-of` , supply `map-of` `every-kv` :

```
(clojure.spec/valid? (clojure.spec/every-kv keyword? string?) {:red "red" :green "green"})
;; => true
```

spec . .

```
(clojure.spec/valid? (clojure.spec/cat :text string? :int int?) ["test" 1])
;;=> true
```

`cat` . cat .

`alt` . :

```
(clojure.spec/valid? (clojure.spec/cat :text-or-int (clojure.spec/alt :text string? :int
int?)) ["test"])
;;=> true
```

`alt` .

. .

```
(clojure.spec/def ::complex-seq (clojure.spec/+ (clojure.spec/cat :num int? :foo-map
(clojure.spec/map-of keyword? int?))))
(clojure.spec/valid? ::complex-seq [0 {:foo 3 :baz 1} 4 {:foo 4}])
;;=> true
```

`::complex-seq` . int int .

clojure.spec : https://riptutorial.com/ko/clojure/topic/2325/clojure-spec

6: clojure.test

Examples

~.

is clojure.test . .

```
(defn square [x]
  (+ x x))

(require '[clojure.test :as t])

(t/is (= 0 (square 0)))
;;=> true

(t/is (= 1 (square 1)))
;;
;; FAIL in () (foo.clj:1)
;; expected: (= 1 (square 1))
;;   actual: (not (= 1 2))
;;=> false
```

testing deftest unit .

```
(deftest add-nums
  (testing "Positive cases"
    (is (= 2 (+ 1 1)))
    (is (= 4 (+ 2 2))))
  (testing "Negative cases"
    (is (= -1 (+ 2 -3)))
    (is (= -4 (+ 8 -12)))))
```

. testing deftest .

deftest

deftest .

.

```
(deftest add-nums
  (is (= 2 (+ 1 1)))
  (is (= 3 (+ 1 2))))
```

+ add-nums . .

.

```
(run-tests)
```

.

```
(run-tests 'your-ns)
```

are clojure.test . .

:

```
(are [x y] (= x y)
   4 (+ 2 2)
   8 (* 2 4))
=> true
```

, (= xy) , is .

is :

```
(do
  (is (= 4 (+ 2 2)))
  (is (= 8 (* 2 4))))
```

.

use-fixtures deftest . .

Fixture (/ ,) .

```
(ns myapp.test
  (require [clojure.test :refer :all])

(defn stub-current-thing [body]
  ;; with-redefs stubs things/current-thing function to return fixed
  ;; value for duration of each test
  (with-redefs [things/current-thing (fn [] {:foo :bar})]
    ;; run test body
    (body)))

(use-fixtures :each stub-current-thing)
```

:once .

```
(defn database-for-tests [all-tests]
  (setup-database)
  (all-tests)
  (drop-database))

(use-fixtures :once database-for-tests)
```

Leiningen

Leiningen `lein test`

clojure.test : https://riptutorial.com/ko/clojure/topic/1901/clojure-test

7: core.async

Examples

: , , , .

core.async .

```
(require [clojure.core.async :as a])
```

chan

chan .

```
(def chan-0 (a/chan)) ;; unbuffered channel: acts as a rendez-vous point.
(def chan-1 (a/chan 3)) ;; channel with a buffer of size 3.
(def chan-2 (a/chan (a/dropping-buffer 3)) ;; channel with a *dropping* buffer of size 3
(def chan-3 (a/chan (a/sliding-buffer 3)) ;; channel with a *sliding* buffer of size 3
```

>!! >!! >!

>!! >!! :

```
(a/>!! my-channel :an-item)
```

nil (, ,, , ,) .

```
;; WON'T WORK
(a/>!! my-channel nil)
=> IllegalArgumentException Can't put nil on channel
```

>!! .

```
(let [ch (a/chan)] ;; unbuffered channel
  (a/>!! ch :item)
  ;; the above call blocks, until another process
  ;; takes the item from the channel.
  )
(let [ch (a/chan 3)] ;; channel with 3-size buffer
  (a/>!! ch :item-1) ;; => true
  (a/>!! ch :item-2) ;; => true
  (a/>!! ch :item-3) ;; => true
  (a/>!! ch :item-4)
  ;; now the buffer is full; blocks until :item-1 is taken from ch.
  )
```

(go ...) a/>! a/>! a/>!! a/>!! :

```
 (a/go (a/>! ch :item))
```

`a/>!!` `a/>!!` , OS goroutine .

`a/>!! (go ...)` anti-pattern.

```
;; NEVER DO THIS
(a/go
  (a/>!! ch :item))
```

<!! <!!

`<!!` `<!!` :

```
;; creating a channel
(def ch (a/chan 3))
;; putting some items in it
(do
  (a/>!! ch :item-1)
  (a/>!! ch :item-2)
  (a/>!! ch :item-3))
;; taking a value
(a/<!! ch) ;; => :item-1
(a/<!! ch) ;; => :item-2
```

`a/<!!` ( ), .

```
(def ch (a/chan))
(a/<!! ch) ;; blocks until another process puts something into ch or closes it
```

`(go ...)` `a/<!` `a/<!!` `a/<!!` :

```
 (a/go (let [x (a/<! ch)] ...))
```

`a/<!!` `a/<!!` , OS goroutine .

`a/<!! a/<!! (go ...)` anti-pattern.

```
;; NEVER DO THIS
(a/go
  (a/<!! ch))
```

`a/close!` *close* `a/close!` :

```
(a/close! ch)
```

`nil` .

```
(def ch (a/chan 5))
```

```
;; putting 2 values in the channel, then closing it
(a/>!! ch :item-1)
(a/>!! ch :item-2)
(a/close! ch)

;; taking from ch will return the items that were put in it, then nil
(a/<!! ch) ;; => :item-1
(a/<!! ch) ;; => :item-2
(a/<!! ch) ;; => nil
(a/<!! ch) ;; => nil
(a/<!! ch) ;; => nil

;; once the channel is closed, >!! will have no effect on the channel:
(a/>!! ch :item-3)
=> false ;; false means the put did not succeed
(a/<!! ch) ;; => nil
```

put!

`a/>!!` `a/>!!` ( ), `a/put!` `a/put!` .

```
(a/put! ch :item)
(a/put! ch :item (fn once-put [closed?] ...)) ;; callback function, will receive
```

ClojureScript `a/>!!` `put! (go)` .

take!

`a/<!!` ( ), `a/take!` `a/take!` .

```
(a/take! ch (fn [x] (do something with x)))
```

put . .

:

```
(def ch (a/chan (a/dropping-buffer 2)))

;; putting more items than buffer size
(a/>!! ch :item-1)
=> true ;; put succeeded
(a/>!! ch :item-2)
=> true
(a/>!! ch :item-3)
=> false ;; put failed

;; no we take from the channel
(a/<!! ch)
=> :item-1
(a/<!! ch)
=> :item-2
(a/<!! ch)
;; blocks! :item-3 is lost
```

:

```
(def ch (a/chan (a/sliding-buffer 2)))

;; putting more items than buffer size
(a/>!! ch :item-1)
=> true
(a/>!! ch :item-2)
=> true
(a/>!! ch :item-3)
=> true

;; no when we take from the channel:
(a/<!! ch)
=> :item-2
(a/<!! ch)
=> :item-3
;; :item-1 was lost
```

core.async : https://riptutorial.com/ko/clojure/topic/5496/core-async

8: core.match

core.match "" .

Examples

```
(let [x true
      y true
      z true]
  (match [x y z]
     [_ false true] 1
     [false true _ ] 2
     [_ _ false] 3
     [_ _ true] 4))

;=> 4
```

```
(let [v [1 2 3]]
  (match [v]
    [[1 1 1]] :a0
    [[1 _ 1]] :a1
    [[1 2 _]] :a2))  ;; _ is used for wildcard matching

;=> :a2
```

```
(let [x {:a 1 :b 1}]
   (match [x]
     [{:a _ :b 2}] :a0
     [{:a 1 :b _}] :a1
     [{:x 3 :y _ :z 4}] :a2))

;=> :a1
```

```
(match [['asymbol]]
  [['asymbol]] :success)

;=> :success
```

core.match : https://riptutorial.com/ko/clojure/topic/2569/core-match--

9: Java interop

- . .
- .- .
- .. .

Clojure Java . Clojure Java .

Examples

Java

. :

```
(.trim " hello ")
;;=> "hello"
```

.

```
(.substring "hello" 0 2)
;;=> "he"
```

Java

.- .

```
(def p (java.awt.Point. 0 1))
(.-x p)
;;=> 0
(.-y p)
;;=> 1
```

Java

.

```
(java.awt.Point. 0 1)
;;=> => #object[java.awt.Point 0x3776d535 "java.awt.Point[x=0,y=1]"]
```

```
(new java.awt.Point 0 1)
;;=> => #object[java.awt.Point 0x3776d535 "java.awt.Point[x=0,y=1]"]
```

.

```
(System/currentTimeMillis)
;;=> 1469493415265
```

.

```
(System/setProperty "foo" "42")
;;=> nil
(System/getProperty "foo")
;;=> "42"
```

Clojure

Java Clojure .

```
IFn times = Clojure.var("clojure.core", "*");
times.invoke(2, 2);
```

`clojure.core` `*` 2 & 2 .

Java interop : https://riptutorial.com/ko/clojure/topic/4036/java-interop

10:

Clojure . + .

. Clojure .

Examples

```
;; returns 3
(+ 1 2)

;; returns 300
(+ 50 210 40)

;; returns 2
(/ 8 4)
```

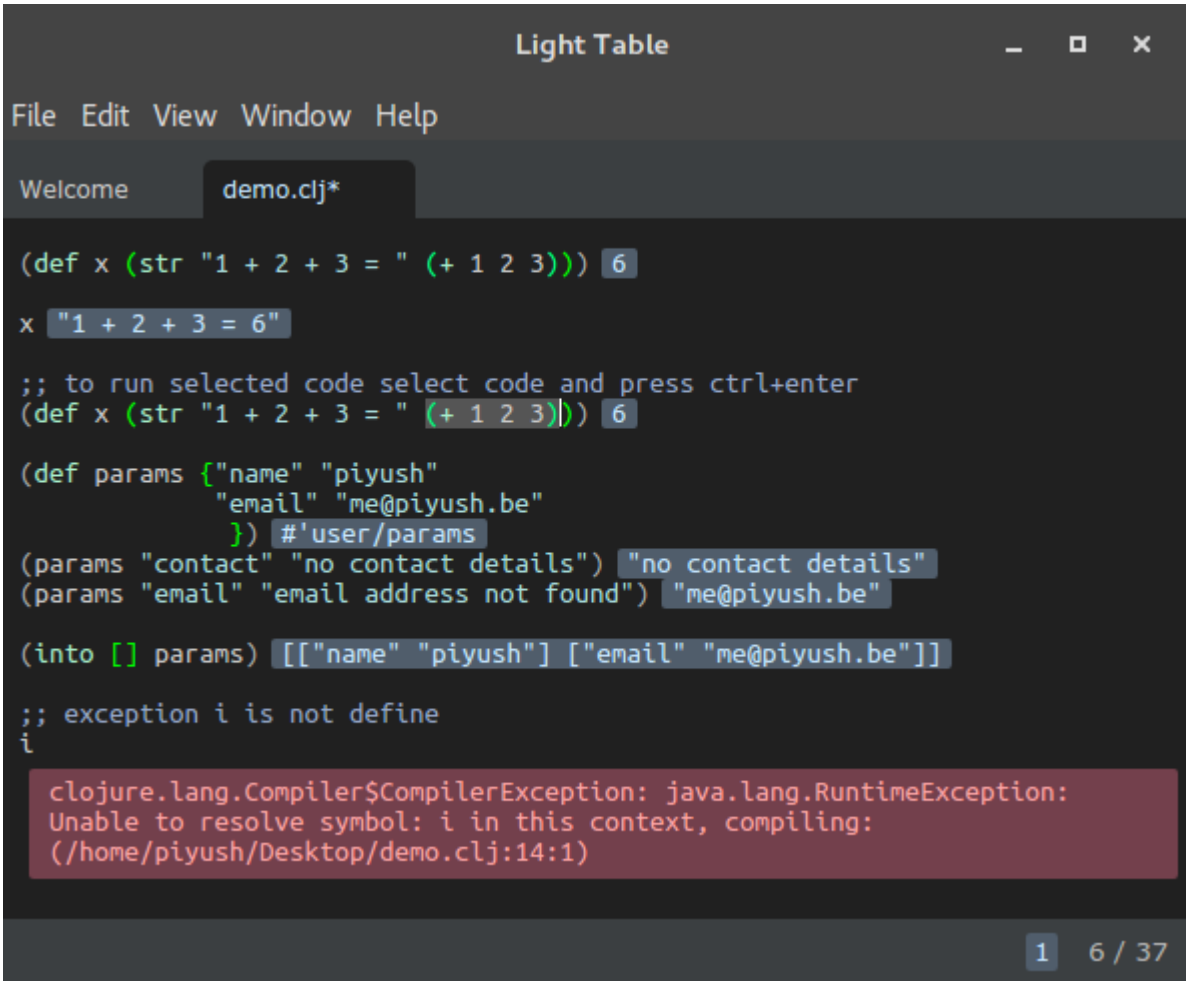
: https://riptutorial.com/ko/clojure/topic/8901/---

11:

Examples

Light Table Clojure .
`project.clj` lein / boot . .

, . `ctrl + enter` blocska . `ctrl + enter` . Clojure Light Table .



. .

Light Table Leiningen . Leiningen , .

: docs.lighttable.com

Clojure Emacs , melpa `clojure-mode` `cider` :

```
M-x package-install [RET] clojure-mode [RET]
M-x package-install [RET] cider [RET]
```

`.clj` `M-x cider-jack-in` REPL . `C-u M-x` (cider-jack-in) `lein boot` . `C-x C-e` .

Lisp paren-aware . Emacs .

- `paredit` Lisp .

  `Mx package-install [RET] paredit [RET]`

- `smartparens` `paredit` .

  `Mx package-install [RET] smartparens [RET]`

- `parinfer` paren Lisp .

  . `parinfer-mode` Github .

`paredit clojure-mode paredit ,`

```
(add-hook 'clojure-mode-hook #'paredit-mode)
```

`smartparens clojure-mode smartparens clojure-mode .`

```
(add-hook 'clojure-mode-hook #'smartparens-strict-mode)
```

Atom .

.

```
apm install parinfer
apm install language-clojure
apm install proto-repl
```

IntelliJ IDEA +

IDEA .

Cursive Plugin .

IDEA , . , , .

: `IntelliJ` , `Cursive` 30 . `IntelliJ` . , . 6 .

Spacemacs + CIDER

Spacemacs . vim . Spacemacs CIDER Clojure .

Clojure emacs . .

```
$ mv ~/.emacs.d ~/.emacs.d.backup
```

.

```
$ git clone https://github.com/syl20bnr/spacemacs ~/.emacs.d
```

, . . . spacemacs Clojure . `.spacemacs` `SPC fed` . `dotspacemacs-configuration-layers` `dotspacemacs-configuration-layers` . `clojure` .

```
(defun dotspacemacs/layers ()
  (setq-default
   ;; ...
   dotspacemacs-configuration-layers
   '(clojure
     ;; ...
    )
   ;; ...
))
```

`SPC fe R` `SPC fe R` ., `.clj` , `si` spacemacs Clojure REPL , , `ss` REPL .

spacemacs . Spacemacs ,

.

1. fireplace.vim : Clojure REPL
2. vim-sexp :
3. vim-sexp-regular-people :
4. vim-surround : "" , ,
5. salve.vim : Leiningen Boot Vim .
6. rainbow_parentheses.vim :

, , .

1. vim-clojure-static (vim 7.3.803)
2. Vim-Clojure-highlight

vim-sexp paredit.vim vim-parinfer .

: https://riptutorial.com/ko/clojure/topic/1387/--

12:

Examples

.

`defn` , .

```
(defn welcome ....)
```

Docstring.

```
(defn welcome
    "Return a welcome message to the world"
     ...)
```

.

```
(defn welcome
    "Return a welcome message"
    [name]
    ...)
```

.

```
(defn welcome
    "Return a welcome message"
    [name]
    (str "Hello, " name "!"))
```

:

```
=> (welcome "World")

"Hello, World!"
```

Clojure 0 .

```
(defn welcome
    "Without parameters"
    []
    "Hello!")

(defn square
    "Take one parameter"
    [x]
    (* x x))
```

```
(defn multiplier
    "Two parameters"
    [x y]
    (* x y))
```

. *arity* . , Clojure "" .

```
(defn sum-args
  ;; 3 arguments
  ([x y z]
     (+ x y z))
  ;; 2 arguments
  ([x y]
     (+ x y))
  ;; 1 argument
  ([x]
     (+ x 1)))
```

. .

```
(defn do-something
  ;; 2 arguments
  ([first second]
     (str first " " second))
  ;; 1 argument
  ([x]
     (* x x x)))
```

Clojure **&** . .

```
(defn sum [& args]
  (apply + args))

(defn sum-and-multiply [x & args]
  (* x (apply + args)))
```

:

```
=> (sum 1 11 23 42)
77

=> (sum-and-multiply 2 1 2 3)  ;; 2*(1+2+3)
12
```

: .

```
(fn [x y] (+ x y))
```

. `defn` (`&` ,) `fn` . `defn` `(def (fn ...))` .

```
#(+ %1 %2)
```

. . %1 , %2 , %3 . % .

. . .

```
(def f #(map #(+ %1 2) %1))
```

varargs . .

```
#(every? even? %&)
```

true .

```
(#(every? even? %&) 2 4 6 8)
;; true
(#(every? even? %&) 1 2 4 6)
;; false
```

. .

```
(fn addition [& addends] (apply + addends))
```

: https://riptutorial.com/ko/clojure/topic/3078/

13:

- macroexpand ' quote . (macroexpand (quote (infix 1 + 2))) .

, read-eval-print-loop eval .

.

.

- - , . . # .

Examples

Clojure . .

.

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

Clojure . , (: 1 + 2) .

```
(defmacro infix [first-operand operator second-operand]
    "Converts an infix expression into a prefix expression"
    (list operator first-operand second-operand))
```

.

- defmacro .
- infix .
- [first-operand operator second-operand] .
- (list operator first-operand second-operand) . infix list .

defmacro Clojure . . :

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

infix 1 + 2 (+ 1 2) . Clojure .

infix macroexpand .

```
(macroexpand '(infix 1 + 2))
;; => (+ 1 2)
```

macroexpand (1 + 2 (+ 1 2) infix Clojure .

(core.clj : 807) :

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

- ` syntax-quote (quote) : (let …) , (if …)
- ~ unquote . x (x)
- ~@ aka unquote-splicing unquote .
- # ID . , ID and# let and# if

: https://riptutorial.com/ko/clojure/topic/2322/

14:

- ( )
- (def symbol "docstring")
- (symbol_0 symbol_1 symbol_2 ...)

`(defn)` .

Examples

Clojure .

- :
  -
  - longs (`2^31 - 1`)
  - ()
-
- :
  -
  -
  -
-

: https://riptutorial.com/ko/clojure/topic/4449/

15:

.

Examples

(- >>)

.

```
(prn (str (+ 2 3)))
```

```
(->> 2
     (+ 3)
     (str)
     (prn))
```

(->)

.

```
 (rename-keys (assoc {:a 1} :b 1) {:b :new-b}))
```

? . ->

```
(-> {:a 1}
    (assoc :b 1)                ;;(assoc map key val)
    (rename-keys {:b :new-b}))  ;;(rename-keys map key-newkey-map)
```

(as->)

thread first thread last . .

```
(as-> [1 2] x
      (map #(+ 1 %) x)
      (if (> (count x) 2) "Large" "Small"))
```

: https://riptutorial.com/ko/clojure/topic/9582/-

16:

Clojure ( ). Clojure ( ) . Clojure .

Examples

def . atom :

```
(def counter (atom 0))
```

0 atom . .

```
(def foo (atom "Hello"))
(def bar (atom ["W" "o" "r" "l" "d"]))
```

@ .

```
@counter ; => 0
```

:

```
(def number (atom 3))
(println (inc @number))
;; This should output 4
```

swap! reset! . swap! . reset! .

```
(swap! counter inc) ; => 1
(reset! counter 0) ; => 0
```

10 2 .

```
(def count (atom 0))

(while (< @atom 10)
  (swap! atom inc)
  (println (Math/pow 2 @atom)))
```

: https://riptutorial.com/ko/clojure/topic/7519/

17:

Examples

http-kit

Ring Clojure HTTP API. Ruby 's Rack Python WSGI .

http-kit .

Leiningen :

```
lein new app myapp
```

`project.clj` http-kit .

```
  :dependencies [[org.clojure/clojure "1.8.0"]
                 [http-kit "2.1.18"]]
```

`:require core.clj` http-kit `core.clj` .

```
(ns test.core
  (:gen-class)
  (:require [org.httpkit.server :refer [run-server]]))
```

. .

```
(defn app [req]
  {:status  200
   :headers {"Content-Type" "text/html"}
   :body    "hello HTTP!"})
```

200 OK .

`-main` .

```
(defn -main
  [& args]
  (run-server app {:port 8080}))
```

`lein run` `http://localhost:8080/` `lein run` .

Luminus

Luminus Clojure . . Luminus . majour .

luminus . .

```
lein new luminus my-app
cd my-app
lein run
```

3000 .

`lein new luminus myapp` . .

- + aleph - Aleph .
- + -
- + http-kit - HTTP Kit .

- + h2 - db.core H2 db .
- + sqlite - db.core SQLite db
- + postgres - db.core PostreSQL .
- + mysql - db.core MySQL .
- + mongodb - db.core MongoDB
- + datomic - db.core Datomic .

- + auth - .
- + auth-jwe - JWE .
- + - CIDER nREPL CIDER .
- + cljs - [ClojureScript] [cljs] .
- + re-frame - [ClojureScript] [cljs]
- + - clj-webdriver
- + swagger - compojure-api Swagger-UI .
- + sassc - SassC SASS / SCSS .
- + service - HTML
- + war - Apache Tomcat WAR (WildFly Immutant)
- + site - (H2) ClojureScript .

. :

```
lein new luminus myapp +cljs
```

(:

```
lein new luminus myapp +cljs +swagger +postgres
```

: https://riptutorial.com/ko/clojure/topic/2323/--

18: CIDER

C lojure () ocks **R** **D** evelopment **E** nvironment nteractive . . CIDER . CIDER nREPL, REPL SLID CIDER .

Examples

CIDER `cider-eval-last-sexp` . `Cx Ce Cx Ce` .

CIDER `Cx Ce Cc Ce` .

/ .

:

```
(defn say-hello
  [username]
  (format "Hello, my name is %s" username))

(defn introducing-bob
  []
  (say-hello "Bob")) => "Hello, my name is Bob"
```

`say-hello` `Cx Ce Cc Ce` `Hello, my name is Bob` `Hello, my name is Bob` .

CIDER `cider-insert-last-sexp-in-repl` . `Cc Cp` .

CIDER `Cc Cp` .

.

```
(def databases {:database1  {:password "password"
                             :database "test"
                             :port "5432"
                             :host "localhost"
                             :user "username"}

                :database2 {:password "password"
                            :database "different_test_db"
                            :port "5432"
                            :host "localhost"
                            :user "vader"}})

(defn get-database-config
  []
  databases)

(get-database-config)
```

`get-database-config` `Cc Cp` .

```
{:database1
 {:password "password",
  :database "test",
  :port "5432",
  :host "localhost",
  :user "username"},
 :database2
 {:password "password",
  :database "different_test_db",
  :port "5432",
  :host "localhost",
  :user "vader"}}
```

CIDER : https://riptutorial.com/ko/clojure/topic/8847/-cider

19:

Examples

Clojure `nil false` .

:

```
(boolean nil)         ;=> false
(boolean false)       ;=> false
(boolean true)        ;=> true
(boolean :a)          ;=> true
(boolean "false")     ;=> true
(boolean 0)           ;=> true
(boolean "")          ;=> true
(boolean [])          ;=> true
(boolean '())         ;=> true

(filter identity [:a false :b true])   ;=> (:a :b true)
(remove identity [:a false :b true])   ;=> (false)
```

Clojure `false nil` `false` . `(boolean value)` . truthiness `(or) true` truthy , `(and) true` truthy .

```
=> (or false nil)
nil    ; none are truthy
=> (and '() [] {} #{} "" :x 0 1 true)
true    ; all are truthy
=> (boolean "false")
true    ; because naturally, all strings are truthy
```

: https://riptutorial.com/ko/clojure/topic/4116/

20:

- `'()` → `()`
- `'(1 2 3 4 5)` → `(1 2 3 4 5)`
- `'(1 foo 2 bar 3)` → `(1 'foo 2 'bar 3)`
- `(list 1 2 3 4 5)` → `(1 2 3 4 5)`
- `(list* [1 2 3 4 5])` → `(1 2 3 4 5)`
- `[]` → `[]`
- `[1 2 3 4 5]` → `[1 2 3 4 5]`
- `(vector 1 2 3 4 5)` → `[1 2 3 4 5]`
- `(vec '(1 2 3 4 5))` → `[1 2 3 4 5]`
- `{}` => `{}`
- `{:keyA 1 :keyB 2}` → `{:keyA 1 :keyB 2}`
- `{:keyA 1, :keyB 2}` → `{:keyA 1 :keyB 2}`
- `(hash-map :keyA 1 :keyB 2)` → `{:keyA 1 :keyB 2}`
- `(sorted-map 5 "five" 1 "one")` → `{1 "one" 5 "five"}` ( )
- `#{}` → `#{}`
- `#{1 2 3 4 5}` → `#{4 3 2 5 1}` ( )
- `(hash-set 1 2 3 4 5)` → `#{2 5 4 1 3}` ( )
- `(sorted-set 2 5 4 3 1)` → `#{1 2 3 4 5}`

Examples

Clojure , `conj` , `count` `seq` .

- `conj` "" "" . .
- `count` `count` .
- `seq` `nil` .

.

```
()
;;=> ()
```

Clojure . `conj` "conjoins". .

```
(conj () :foo)
;;=> (:foo)

(conj (conj () :bar) :foo)
;;=> (:foo :bar)
```

, , . `(:foo)` `:foo` REPL `(:foo)` nullary `IllegalArgumentException` .

```
(:foo)
```

```
;; java.lang.IllegalArgumentException: Wrong number of args passed to keyword: :foo
```

Clojure `quote` .

```
'(:foo)
;;=> (:foo)

'(:foo :bar)
;;=> (:foo :bar)
```

.

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

'(1 (+ 1 1) 3)
;;=> (1 (+ 1 1) 3)
```

`list` .

```
(list)
;;=> ()

(list :foo)
;;=> (:foo)

(list :foo :bar)
;;=> (:foo :bar)

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

`count` `count` .

```
(count ())
;;=> 0

(count (conj () :foo))
;;=> 1

(count '(:foo :bar))
;;=> 2
```

`list?` `list?` :

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

(list? '(:foo :bar))
;;=> true

(list? nil)
;;=> false

(list? 42)
```

```
;;=> false

(list? :foo)
;;=> false
```

`peek` .

```
(peek ())
;;=> nil

(peek '(:foo))
;;=> :foo

(peek '(:foo :bar))
;;=> :foo
```

`pop` .

```
(pop '(:foo))
;;=> ()

(pop '(:foo :bar))
;;=> (:bar)
```

`pop` (), `IllegalStateException` `IllegalStateException` .

```
(pop ())
;; java.lang.IllegalStateException: Can't pop empty list
```

., `seq` .

```
(seq ())
;;=> nil

(seq '(:foo))
;;=> (:foo)

(seq '(:foo :bar))
;;=> (:foo :bar)

(let [x '(:foo :bar)]
  (identical? x (seq x)))
;;=> true
```

., `first` `rest` . `cons` truct.

`seq?` `seq?` :

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

(seq? 42)
;;=> false
```

```
(seq? :foo)
;;=> false
```

.

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

(seq? '(:foo :bar))
;;=> true
```

`seq` `rseq` `keys` `vals` :

```
(seq? (seq ()))
;;=> false

(seq? (seq '(:foo :bar)))
;;=> true

(seq? (seq []))
;;=> false

(seq? (seq [:foo :bar]))
;;=> true

(seq? (rseq []))
;;=> false

(seq? (rseq [:foo :bar]))
;;=> true

(seq? (seq {}))
;;=> false

(seq? (seq {:foo :bar :baz :qux}))
;;=> true

(seq? (keys {}))
;;=> false

(seq? (keys {:foo :bar :baz :qux}))
;;=> true

(seq? (vals {}))
;;=> false

(seq? (vals {:foo :bar :baz :qux}))
;;=> true

(seq? (seq #{}))
;;=> false

(seq? (seq #{:foo :bar}))
;;=> true
```

. `peek` `pop` `count` , . Clojure `peek` `pop` `ClassCastException` .

```
(peek (seq [:foo :bar]))
;; java.lang.ClassCastException: clojure.lang.PersistentVector$ChunkedSeq cannot be cast to
clojure.lang.IPersistentStack

(pop (seq [:foo :bar]))
;; java.lang.ClassCastException: clojure.lang.PersistentVector$ChunkedSeq cannot be cast to
clojure.lang.IPersistentStack

(peek (seq #{:foo :bar}))
;; java.lang.ClassCastException: clojure.lang.APersistentMap$KeySeq cannot be cast to
clojure.lang.IPersistentStack

(pop (seq #{:foo :bar}))
;; java.lang.ClassCastException: clojure.lang.APersistentMap$KeySeq cannot be cast to
clojure.lang.IPersistentStack

(peek (seq {:foo :bar :baz :qux}))
;; java.lang.ClassCastException: clojure.lang.PersistentArrayMap$Seq cannot be cast to
clojure.lang.IPersistentStack

(pop (seq {:foo :bar :baz :qux}))
;; java.lang.ClassCastException: clojure.lang.PersistentArrayMap$Seq cannot be cast to
clojure.lang.IPersistentStack
```

`count` `count` . Clojure , ., `count` .`count` `count` `counted?` :

```
(counted? '(:foo :bar))
;;=> true

(counted? (seq '(:foo :bar)))
;;=> true

(counted? [:foo :bar])
;;=> true

(counted? (seq [:foo :bar]))
;;=> true

(counted? {:foo :bar :baz :qux})
;;=> true

(counted? (seq {:foo :bar :baz :qux}))
;;=> true

(counted? #{:foo :bar})
;;=> true

(counted? (seq #{:foo :bar}))
;;=> false
```

, `first` .`first` `seq` "seqable" .

```
(first nil)
;;=> nil

(first '(:foo))
;;=> :foo

(first '(:foo :bar))
```

```
;;=> :foo

(first [:foo])
;;=> :foo

(first [:foo :bar])
;;=> :foo

(first {:foo :bar})
;;=> [:foo :bar]

(first #{:foo})
;;=> :foo
```

, rest . first , seq . seq ! , rest nil back () .

```
(rest nil)
;;=> ()

(rest '(:foo))
;;=> ()

(rest '(:foo :bar))
;;=> (:bar)

(rest [:foo])
;;=> ()

(rest [:foo :bar])
;;=> (:bar)

(rest {:foo :bar})
;;=> ()

(rest #{:foo})
;;=> ()
```

nil rest next .

```
(next nil)
;;=> nil

(next '(:foo))
;;=> nil

(next [:foo])
;;=> nil
```

cons first first rest .

```
(cons :foo nil)
;;=> (:foo)

(cons :foo (cons :bar nil))
;;=> (:foo :bar)
```

Clojure . "seqable" . . reduce .

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

(reduce + [1 2 3])
;;=> 6

(reduce + #{1 2 3})
;;=> 6
```

first rest .

, lazy-seq , seq . first rest .

, .

```
(seq [:foo :bar])
;;=> (:foo :bar)

(lazy-seq [:foo :bar])
;;=> (:foo :bar)
```

.

```
(defn eager-fibonacci [a b]
  (cons a (eager-fibonacci b (+' a b))))

(defn lazy-fibonacci [a b]
  (lazy-seq (cons a (lazy-fibonacci b (+' a b)))))

(take 10 (eager-fibonacci 0 1))
;; java.lang.StackOverflowError:

(take 10 (lazy-fibonacci 0 1))
;;=> (0 1 1 2 3 5 8 13 21 34)
```

.

```
[]
;;=> []

[:foo]
;;=> [:foo]

[:foo :bar]
;;=> [:foo :bar]

[1 (+ 1 1) 3]
;;=> [1 2 3]
```

vector vector .

```
(vector)
;;=> []

(vector :foo)
```

```
;;=> [:foo]

(vector :foo :bar)
;;=> [:foo :bar]

(vector 1 (+ 1 1) 3)
;;=> [1 2 3]
```

vector? :

```
(vector? [])
;;=> true

(vector? [:foo :bar])
;;=> true

(vector? nil)
;;=> false

(vector? 42)
;;=> false

(vector? :foo)
;;=> false
```

conj .

```
(conj [] :foo)
;;=> [:foo]

(conj (conj [] :foo) :bar)
;;=> [:foo :bar]

(conj [] :foo :bar)
;;=> [:foo :bar]
```

count count .

```
(count [])
;;=> 0

(count (conj [] :foo))
;;=> 1

(count [:foo :bar])
;;=> 2
```

peek .

```
(peek [])
;;=> nil

(peek [:foo])
;;=> :foo

(peek [:foo :bar])
```

```
;;=> :bar
```

pop .

```
(pop [:foo])
;;=> []

(pop [:foo :bar])
;;=> [:foo]
```

IllegalStateException .

```
(pop [])
;; java.lang.IllegalStateException: Can't pop empty vector
```

. "" get :

```
(get [:foo :bar] 0)
;;=> :foo

(get [:foo :bar] 1)
;;=> :bar

(get [:foo :bar] -1)
;;=> nil

(get [:foo :bar] 2)
;;=> nil
```

.

```
([:foo :bar] 0)
;;=> :foo

([:foo :bar] 1)
;;=> :bar
```

nil IndexOutOfBoundsException .

```
([:foo :bar] -1)
;; java.lang.IndexOutOfBoundsException:

([:foo :bar] 2)
;; java.lang.IndexOutOfBoundsException:
```

assoc .

```
(assoc [:foo :bar] 0 42)
;;=> [42 :bar]

(assoc [:foo :bar] 1 42)
;;=> [:foo 42]
```

`count` Clojure `conj` . `count` `IndexOutOfBoundsException` .

```
(assoc [:foo :bar] 2 42)
;;=> [:foo :bar 42]

(assoc [:foo :bar] -1 42)
;; java.lang.IndexOutOfBoundsException:

(assoc [:foo :bar] 3 42)
;; java.lang.IndexOutOfBoundsException:
```

`seq` .

```
(seq [])
;;=> nil

(seq [:foo])
;;=> (:foo)

(seq [:foo :bar])
;;=> (:foo :bar)
```

`rseq` .

```
(rseq [])
;;=> nil

(rseq [:foo])
;;=> (:foo)

(rseq [:foo :bar])
;;=> (:bar :foo)
```

!

```
'(:foo :bar)
;;=> (:foo :bar)

(seq [:foo :bar])
;;=> (:foo :bar)

(list? '(:foo :bar))
;;=> true

(list? (seq [:foo :bar]))
;;=> false

(list? (rseq [:foo :bar]))
;;=> false
```

. , .

(octetorpe) .

```
#{}
```

```
;;=> #{}

#{:foo}
;;=> #{:foo}

#{:foo :bar}
;;=> #{:bar :foo}
```

, .

```
(= #{:foo :bar} #{:bar :foo})
;;=> true
```

`set?` :

```
(set? #{})
;;=> true

(set? #{:foo})
;;=> true

(set? #{:foo :bar})
;;=> true

(set? nil)
;;=> false

(set? 42)
;;=> false

(set? :foo)
;;=> false
```

contains "" `contains?` :

```
(contains? #{} :foo)
;;=> false

(contains? #{:foo} :foo)
;;=> true

(contains? #{:foo} :bar)
;;=> false

(contains? #{} nil)
;;=> false

(contains? #{nil} nil)
;;=> true
```

`nil` .

```
(#{} :foo)
;;=> nil

(#{:foo} :foo)
```

```
;;=> :foo

(#{:foo} :bar)
;;=> nil

(#{} nil)
;;=> nil

(#{nil} nil)
;;=> nil
```

conj .

```
(conj #{} :foo)
;;=> #{:foo}

(conj (conj #{} :foo) :bar)
;;=> #{:bar :foo}

(conj #{:foo} :foo)
;;=> #{:foo}
```

disj :

```
(disj #{} :foo)
;;=> #{}

(disj #{:foo} :foo)
;;=> #{}

(disj #{:foo} :bar)
;;=> #{:foo}

(disj #{:foo :bar} :foo)
;;=> #{:bar}

(disj #{:foo :bar} :bar)
;;=> #{:foo}
```

count .

```
(count #{})
;;=> 0

(count (conj #{} :foo))
;;=> 1

(count #{:foo :bar})
;;=> 2
```

seq .

```
(seq #{})
;;=> nil

(seq #{:foo})
```

```
;;=> (:foo)

(seq #{:foo :bar})
;;=> (:bar :foo)
```

. . "" .

.

```
{}
;;=> {}

{:foo :bar}
;;=> {:foo :bar}

{:foo :bar :baz :qux}
;;=> {:foo :bar, :baz :qux}
```

- . . :foo :bar . :foo :bar , :baz :qux . .

```
(= {:foo :bar :baz :qux}
   {:baz :qux :foo :bar})
;;=> true
```

`map?` :

```
(map? {})
;;=> true

(map? {:foo :bar})
;;=> true

(map? {:foo :bar :baz :qux})
;;=> true

(map? nil)
;;=> false

(map? 42)
;;=> false

(map? :foo)
;;=> false
```

contains "" `contains?` :

```
(contains? {:foo :bar :baz :qux} 42)
;;=> false

(contains? {:foo :bar :baz :qux} :foo)
;;=> true

(contains? {:foo :bar :baz :qux} :bar)
;;=> false
```

```
(contains? {:foo :bar :baz :qux} :baz)
;;=> true

(contains? {:foo :bar :baz :qux} :qux)
;;=> false

(contains? {:foo nil} :foo)
;;=> true

(contains? {:foo nil} :bar)
;;=> false
```

`get` :

```
(get {:foo :bar :baz :qux} 42)
;;=> nil

(get {:foo :bar :baz :qux} :foo)
;;=> :bar

(get {:foo :bar :baz :qux} :bar)
;;=> nil

(get {:foo :bar :baz :qux} :baz)
;;=> :qux

(get {:foo :bar :baz :qux} :qux)
;;=> nil

(get {:foo nil} :foo)
;;=> nil

(get {:foo nil} :bar)
;;=> nil
```

.

```
({:foo :bar :baz :qux} 42)
;;=> nil

({:foo :bar :baz :qux} :foo)
;;=> :bar

({:foo :bar :baz :qux} :bar)
;;=> nil

({:foo :bar :baz :qux} :baz)
;;=> :qux

({:foo :bar :baz :qux} :qux)
;;=> nil

({:foo nil} :foo)
;;=> nil

({:foo nil} :bar)
;;=> nil
```

`find` ( ) .

```
(find {:foo :bar :baz :qux} 42)
;;=> nil

(find {:foo :bar :baz :qux} :foo)
;;=> [:foo :bar]

(find {:foo :bar :baz :qux} :bar)
;;=> nil

(find {:foo :bar :baz :qux} :baz)
;;=> [:baz :qux]

(find {:foo :bar :baz :qux} :qux)
;;=> nil

(find {:foo nil} :foo)
;;=> [:foo nil]

(find {:foo nil} :bar)
;;=> nil
```

`key` `val` .

```
(key (find {:foo :bar} :foo))
;;=> :foo

(val (find {:foo :bar} :foo))
;;=> :bar
```

Clojure . `key` `val` `ClassCastException` .

```
(key [:foo :bar])
;; java.lang.ClassCastException:

(val [:foo :bar])
;; java.lang.ClassCastException:
```

`map-entry?` `map-entry?` :

```
(map-entry? (find {:foo :bar} :foo))
;;=> true

(map-entry? [:foo :bar])
;;=> false
```

`assoc` - .

```
(assoc {} :foo :bar)
;;=> {:foo :bar}

(assoc (assoc {} :foo :bar) :baz :qux)
;;=> {:foo :bar, :baz :qux}
```

```
(assoc {:baz :qux} :foo :bar)
;;=> {:baz :qux, :foo :bar}

(assoc {:foo :bar :baz :qux} :foo 42)
;;=> {:foo 42, :baz :qux}

(assoc {:foo :bar :baz :qux} :baz 42)
;;=> {:foo :bar, :baz 42}
```

dissoc dissoc - dissoc .

```
(dissoc {:foo :bar :baz :qux} 42)
;;=> {:foo :bar :baz :qux}

(dissoc {:foo :bar :baz :qux} :foo)
;;=> {:baz :qux}

(dissoc {:foo :bar :baz :qux} :bar)
;;=> {:foo :bar :baz :qux}

(dissoc {:foo :bar :baz :qux} :baz)
;;=> {:foo :bar}

(dissoc {:foo :bar :baz :qux} :qux)
;;=> {:foo :bar :baz :qux}

(dissoc {:foo nil} :foo)
;;=> {}
```

count count .

```
(count {})
;;=> 0

(count (assoc {} :foo :bar))
;;=> 1

(count {:foo :bar :baz :qux})
;;=> 2
```

seq .

```
(seq {})
;;=> nil

(seq {:foo :bar})
;;=> ([:foo :bar])

(seq {:foo :bar :baz :qux})
;;=> ([:foo :bar] [:baz :qux])
```

seq .

keys vals :

```
(keys {})
```

```
;;=> nil

(keys {:foo :bar})
;;=> (:foo)

(keys {:foo :bar :baz :qux})
;;=> (:foo :baz)

(vals {})
;;=> nil

(vals {:foo :bar})
;;=> (:bar)

(vals {:foo :bar :baz :qux})
;;=> (:bar :qux)
```

Clojure 1.9 . (" "). .

```
;; typical map syntax
(def p {:person/first"Darth" :person/last "Vader" :person/email "darth@death.star"})

;; namespace map literal syntax
(def p #:person{:first "Darth" :last "Vader" :email "darth@death.star"})
```

: https://riptutorial.com/ko/clojure/topic/1389/--

21:

Examples

&

```
(defrecord Logline [datetime action user id])
(def pattern #"(\d{8}-\d{2}:\d{2}:\d{2}.\d{3})\|.*\|(\w*),(\w*),(\d*)")
(defn parser [line]
  (if-let [[_ dt a u i] (re-find pattern line)]
          (->Logline dt a u i)))
```

.

```
(def sample "20170426-17:20:04.005|bip.com|1.0.0|alert|Update,john,12")
```

:

```
(parser sample)
```

:

```
#user.Logline{:datetime "20170426-17:20:04.005", :action "Update", :user "john", :id "12"}
```

: https://riptutorial.com/ko/clojure/topic/9822/--

22:

. , , .

Clojure 1.7 map, filter, take . comp .

. into , sequence sequence ., . .

```
(take 0 (sequence (map #(do (prn '-> %) %)) (range 5)))
;; -> 0
;; => ()
```

.

```
(take 0 (sequence (comp (map #(do (prn '-> %) %)) (remove number?)) (range 5)))
;; -> 0
;; -> 1
;; -> 2
;; -> 3
;; -> 4
;; => ()
```

.

```
(take 0 (map #(do (prn '-> %) %) (range 5)))
;; => ()
```

Examples

```
(let [xf (comp
           (map inc)
           (filter even?))]
  (transduce xf + [1 2 3 4 5 6 7 8 9 10]))
;; => 30
```

xf transduce . 1 .

transduce reduce + .

thread-last .

```
(->> [1 2 3 4 5 6 7 8 9 10]
     (map inc)
     (filter even?)
     (reduce +))
;; => 30
```

```
(def xf (filter keyword?))
```

.

```
(sequence xf [:a 1 2 :b :c]) ;; => (:a :b :c)
```

.

```
(transduce xf str [:a 1 2 :b :c]) ;; => ":a:b:c"
```

, conj :

```
(into [] xf [:a 1 2 :b :c]) ;; => [:a :b :c]
```

.

```
(require '[clojure.core.async :refer [chan >!! <!! poll!]])
(doseq [e [:a 1 2 :b :c]] (>!! ch e))
(<!! ch) ;; => :a
(<!! ch) ;; => :b
(<!! ch) ;; => :c
(poll! ch) ;;=> nil
```

/

Clojure ( ) . :

`(map inc) (filter odd?)`

: comp . , . 50 % .

:

```
(def composed-fn (comp (map inc) (filter odd?)))
```

:

```
;; So instead of doing this:
(->> [1 8 3 10 5]
     (map inc)
    (filter odd?))
;; Output [9 11]

;; We do this:
(into [] composed-fn [1 8 3 10 5])
;; Output: [9 11]
```

: https://riptutorial.com/ko/clojure/topic/10814/-

23:

Examples

( ) :

```
(slurp "./small_file.txt")
```

:

```
(spit "./file.txt" "Ocelots are Awesome!")    ; overwrite existing content
(spit "./log.txt" "2016-07-26 New entry." :append true)
```

.

```
(use 'clojure.java.io)
(with-open [rdr (reader "./file.txt")]
    (line-seq rdr)    ; returns lazy-seq
)    ; with-open macro calls (.close rdr)
```

.

```
(use 'clojure.java.io)
(with-open [wrtr (writer "./log.txt" :append true)]
    (.write wrtr "2016-07-26 New entry.")
)    ; with-open macro calls (.close wrtr)
```

:

```
(use 'clojure.java.io)
(with-open [wrtr (writer "./file.txt")]
    (.write wrtr "Everything in file.txt has been replaced with this text.")
)    ; with-open macro calls (.close wrtr)
```

:

- URL .
- `(slurp)` `(spit)` `clojure.java.io/reader` `/writer` .

: https://riptutorial.com/ko/clojure/topic/3922/-

S. No		Contributors
1	Clojure	adairdavid, Adeel Ansari, alejosocorro, Alex Miller, Arclite, avichalp, Blake Miller, Community, CP9, D-side, Geoff, Greg, KettuJKL, Kiran, Martin Janiczek, n2o, Nikita Prokopov, Sajjad, Sam Estep, Sean Allred, Zaz
2	clj-time	Rishu Saniya, Akanksha, Mrinal Saurabh, Vishakha Silky
3	Clojure destructuring	camdez, kaffein, kolen, leeor, Michał Marczyk, MuSaiXi, r00tt, RedBlueThing, ryo, tsleyson, Zaz
4	clojure.core	Akanksha, Mrinal Saurabh, Surbhi Garg
5	clojure.spec	Adam Lee, Alex Miller, kolen, leeor, nXqd
6	clojure.test	jisaw, kolen, leeor, porglezomp, Sam Estep
7	core.async	Valentin Waeselynck
8	core.match	Kiran
9	Java interop	leeor
10		Jim
11		Adeel Ansari, agent_orange, amalloy, g1eny0ung, Geoff, Kiran, kolen, MuSaiXi, Piyush, Qwerp-Derp, spinningarrow, stardiviner, superkonduktr, swlkr
12		alejosocorro, fokz, Qwerp-Derp, tar, tsleyson
13		Alex Miller, kolen, mathk, Sam Estep, snowcrshd
14		Aryaman Arora, Qwerp-Derp, Stephen Leppik, Zaz
15		Kusum Ijari, Mrinal Saurabh
16		Qwerp-Derp, systemfreund
17		Emin Tham, kolen, r00tt
18	CIDER	avichalp
19		Alan Thompson, Michiel Borkent, Zaz
20		Alex Miller, Kenogu Labz, nXqd, Sam Estep

21		user2611740
22		l0st3d, Mrinal Saurabh
23		Zaz