Clojure Core API Reference

Clojure v1.3 API

- Overview
- API Index

Namespaces

- clojure.core
- clojure.data
- clojure.inspector
- clojure.java.browse
- clojure.java.io
- clojure.java.javadoc
- clojure.java.shell
- clojure.main
- clojure.pprint
- clojure.reflect
- clojure.repl
- clojure.set
- clojure.stacktrace
- clojure.string
- clojure.template
- clojure.test
- clojure.walk
- clojure.xml
- clojure.zip

Other Versions

- v1.4 (in development)
- v1.2 (stable)
- v1.1 (stable)

Clojure Home

Table of Contents

agent-errors
aget
alength
alias
all-ns
alter
alter-meta!
alter-var-root
amap
ancestors
and
apply
areduce
array-map
aset
aset-boolean
aset-byte
aset-char
aset-double
aset-float
aset-int
aset-long
aset-short
assert
assoc
assoc!
assoc-in
associative?
atom
await
await-for
bases
bean
bigdec
bigint
biginteger
binding
bit-and
bit-and-not
bit-clear
bit-flip
bit-not
bit-or
bit-set
bit-shift-left
bit-shift-right
bit-test
bit-xor
boolean

boolean-array
booleans
bound-fn
bound-fn*
bound?
butlast
byte
byte-array
bytes
case
cast
char
char-array
char-escape-string
char-name-string
char?
chars
class
class?
clear-agent-errors
clojure-version
coll?
comment
commute
comp
comparator
compare
compare-and-set!
compile
complement
concat
cond
condp
conj
conj!
cons
constantly
construct-proxy
contains?
count
counted?
create-ns
create-struct
cycle
dec
dec'
decimal?
declare
definline

defmacro
defmethod
defmulti
defn
defn-
defonce
defprotocol
defrecord
defstruct
deftype
delay
delay?
deliver
denominator
deref
derive
descendants
disj
disj!
dissoc
dissoc!
distinct
distinct?
doall
dorun
doseq
dosync
dotimes
doto
double
double-array
doubles
drop
drop-last
drop-while
empty
empty?
ensure
enumeration-seq
error-handler
error-mode
eval
even?
every-pred
every?
extend
extend-protocol
extend-type
extenders

extends?
false?
ffirst
file-seq
filter
find
find-keyword
find-ns
find-var
first
flatten
float
float-array
float?
floats
flush
fn
fn?
fnext
fnil
for
force
format
frequencies
future
future-call
future-cancel
future-cancelled?
future-done?
future?
gen-class
gen-interface
gensym
get
get-in
get-method
get-proxy-class
get-thread-bindings
get-validator
group-by
hash
hash-map
hash-set
identical?
identity
if-let
if-not
ifn?
import

in-ns
inc
inc'
init-proxy
instance?
int
int-array
integer?
interleave
intern
interpose
into
into-array
ints
io!
isa?
iterate
iterator-seq
juxt
keep
keep-indexed
key
keys
keyword
keyword?
last
lazy-cat
lazy-seq
let
letfn
line-seq
list
list*
list?
load
load-file
load-reader
load-string
loaded-libs
locking
long
long-array
longs
loop
macroexpand
macroexpand-1
make-array
make-hierarchy
map

map-indexed
map?
mapcat
max
max-key
memfn
memoize
merge
merge-with
meta
methods
min
min-key
mod
name
namespace
namespace-munge
neg?
newline
next
nfirst
nil?
nnext
not
not-any?
not-empty
not-every?
not=
ns
ns-aliases
ns-imports
ns-interns
ns-map
ns-name
ns-publics
ns-refers
ns-resolve
ns-unalias
ns-unmap
nth
nthnext
nthrest
num
number?
numerator
object-array
odd?
or
parents

partial
partition
partition-all
partition-by
pcalls
peek
persistent!
pmap
pop
pop!
pop-thread-bindings
pos?
pr
pr-str
prefer-method
prefers
print
print-str
printf
println
println-str
prn
prn-str
promise
proxy
proxy-mappings
proxy-super
push-thread-bindings
pvalues
quot
rand
rand-int
rand-nth
range
ratio?
rational?
rationalize
re-find
re-groups
re-matcher
re-matches
re-pattern
re-seq
read
read-line
read-string
realized?
reduce
reductions

ref
ref-history-count
ref-max-history
ref-min-history
ref-set
refer
refer-clojure
reify
release-pending-sends
rem
remove
remove-all-methods
remove-method
remove-ns
remove-watch
repeat
repeatedly
replace
replicate
require
reset!
reset-meta!
resolve
rest
restart-agent
resultset-seq
reverse
reversible?
rseq
rsubseq
satisfies?
second
select-keys
send
send-off
seq
seq?
seque
sequence
sequential?
set
set-error-handler!
set-error-mode!
set-validator!
set?
short
short-array
shorts
shuffle

shutdown-agents
slurp
some
some-fn
sort
sort-by
sorted-map
sorted-map-by
sorted-set
sorted-set-by
sorted?
special-symbol?
spit
split-at
split-with
str
string?
struct
struct-map
subs
subseq
subvec
supers
swap!
symbol
symbol?
sync
take
take-last
take-nth
take-while
test
the-ns
thread-bound?
time
to-array
to-array-2d
trampoline
transient
tree-seq
true?
type
unchecked-add
unchecked-add-int
unchecked-byte
unchecked-char
unchecked-dec
unchecked-dec-int
unchecked-divide-int

unchecked-double
unchecked-float
unchecked-inc
unchecked-inc-int
unchecked-int
unchecked-long
unchecked-multiply
unchecked-multiply-int
unchecked-negate
unchecked-negate-int
unchecked-remainder-int
unchecked-short
unchecked-subtract
unchecked-subtract-int
underive
update-in
update-proxy
use
val
vals
var-get
var-set
var?
vary-meta
vec
vector
vector-of
vector?
when
when-first
when-let
when-not
while
with-bindings
with-bindings*
with-in-str
with-local-vars
with-meta
with-open
with-out-str
with-precision
with-redefs
with-redefs-fn
xml-seq
zero?
zipmap

clojure.core.protocols
InternalReduce

API for clojure.core - Clojure v1.3 (stable)

Full namespace name: clojure.core

Overview

```
Fundamental library of the Clojure language
```

Public Variables and Functions

---

*

function

```
Usage: (*)
       (* x)
       (* x y)
       (* x y & more)

Returns the product of nums. (*) returns 1. Does not auto-promote
longs, will throw on overflow. See also: *'
```

Added in Clojure version 1.2
Source

---

*'

function

```
Usage: (*')
       (*' x)
       (*' x y)
       (*' x y & more)

Returns the product of nums. (*) returns 1. Supports arbitrary precision.
See also: *
```

Added in Clojure version 1.0
Source

---

*1

var

```
bound in a repl thread to the most recent value printed
```

Added in Clojure version 1.0
Source

---

*2

var

```
bound in a repl thread to the second most recent value printed
```

Added in Clojure version 1.0
Source

---

*3

var

```
bound in a repl thread to the third most recent value printed
```

Added in Clojure version 1.0
Source

---

agent

var

```
The agent currently running an action on this thread, else nil
```

Added in Clojure version 1.0

---

clojure-version

var

```
The version info for Clojure core, as a map containing :major :minor
:incremental and :qualifier keys. Feature releases may increment
:minor and/or :major, bugfix releases will increment :incremental.
Possible values of :qualifier include "GA", "SNAPSHOT", "RC-x" "BETA-x"
```

Added in Clojure version 1.0
Source

---

command-line-args

var

```
A sequence of the supplied command line arguments, or nil if
none were supplied
```

Added in Clojure version 1.0

---

compile-files

var

```
Set to true when compiling files, false otherwise.
```

Added in Clojure version 1.0

---

compile-path

var

```
Specifies the directory where 'compile' will write out .class
files. This directory must be in the classpath for 'compile' to
work.

Defaults to "classes"
```

Added in Clojure version 1.0

---

*e

var

```
bound in a repl thread to the most recent exception caught by the repl
```

Added in Clojure version 1.0
Source

---

err

var

```
A java.io.Writer object representing standard error for print operations.

Defaults to System/err, wrapped in a PrintWriter
```

Added in Clojure version 1.0

---

file

var

```
The path of the file being evaluated, as a String.

Evaluates to nil when there is no file, eg. in the REPL.
```

Added in Clojure version 1.0

---

flush-on-newline

var

```
When set to true, output will be flushed whenever a newline is printed.

Defaults to true.
```

Added in Clojure version 1.0

---

in

var

```
A java.io.Reader object representing standard input for read operations.

Defaults to System/in, wrapped in a LineNumberingPushbackReader
```

Added in Clojure version 1.0

---

ns

var

```
A clojure.lang.Namespace object representing the current namespace.
```

Added in Clojure version 1.0

---

out

var

```
A java.io.Writer object representing standard output for print operations.

Defaults to System/out, wrapped in an OutputStreamWriter
```

Added in Clojure version 1.0

---

print-dup

var

```
When set to logical true, objects will be printed in a way that preserves
their type when read in later.

Defaults to false.
```

Added in Clojure version 1.0

---

print-length

var

```
*print-length* controls how many items of each collection the
printer will print. If it is bound to logical false, there is no
limit. Otherwise, it must be bound to an integer indicating the maximum
number of items of each collection to print. If a collection contains
more items, the printer will print items up to the limit followed by
'...' to represent the remaining items. The root binding is nil
indicating no limit.
```

Added in Clojure version 1.0
Source

---

print-level

var

```
*print-level* controls how many levels deep the printer will
print nested objects. If it is bound to logical false, there is no
limit. Otherwise, it must be bound to an integer indicating the maximum
level to print. Each argument to print is at level 0; if an argument is a
collection, its items are at level 1; and so on. If an object is a
collection and is at a level greater than or equal to the value bound to
*print-level*, the printer prints '#' to represent it. The root binding
is nil indicating no limit.
```

Added in Clojure version 1.0
Source

---

print-meta

var

```
If set to logical true, when printing an object, its metadata will also
be printed in a form that can be read back by the reader.

Defaults to false.
```

Added in Clojure version 1.0

---

print-readably

var

```
When set to logical false, strings and characters will be printed with
non-alphanumeric characters converted to the appropriate escape sequences.

Defaults to true
```

Added in Clojure version 1.0

---

read-eval

var

```
When set to logical false, the EvalReader (#=(...)) is disabled in the
read/load in the thread-local binding.
Example: (binding [*read-eval* false] (read-string "#=(eval (def x 3))"))

Defaults to true
```

Added in Clojure version 1.0

---

unchecked-math

var

```
While bound to true, compilations of +, -, *, inc, dec and the
coercions will be done without overflow checks. Default: false.
```

Added in Clojure version 1.3

---

warn-on-reflection

var

```
When set to true, the compiler will emit warnings when reflection is
needed to resolve Java method calls or field accesses.

Defaults to false.
```

Added in Clojure version 1.0

+

function

```
Usage: (+)
       (+ x)
       (+ x y)
       (+ x y & more)

Returns the sum of nums. (+) returns 0. Does not auto-promote
longs, will throw on overflow. See also: +'
```

Added in Clojure version 1.2
Source

+'

function

```
Usage: (+')
       (+' x)
       (+' x y)
       (+' x y & more)

Returns the sum of nums. (+) returns 0. Supports arbitrary precision.
See also: +
```

Added in Clojure version 1.0
Source

-

function

```
Usage: (- x)
       (- x y)
       (- x y & more)

If no ys are supplied, returns the negation of x, else subtracts
the ys from x and returns the result. Does not auto-promote
longs, will throw on overflow. See also: -'
```

Added in Clojure version 1.2
Source

-'

function

```
Usage: (-' x)
       (-' x y)
       (-' x y & more)

If no ys are supplied, returns the negation of x, else subtracts
the ys from x and returns the result. Supports arbitrary precision.
See also: -
```

Added in Clojure version 1.0
Source

->

macro

```
Usage: (-> x)
       (-> x form)
       (-> x form & more)

Threads the expr through the forms. Inserts x as the
second item in the first form, making a list of it if it is not a
list already. If there are more forms, inserts the first form as the
second item in second form, etc.
```

Added in Clojure version 1.0
Source

->>

macro

```
Usage: (->> x form)
       (->> x form & more)

Threads the expr through the forms. Inserts x as the
last item in the first form, making a list of it if it is not a
list already. If there are more forms, inserts the first form as the
last item in second form, etc.
```

Added in Clojure version 1.1

Source

..

macro

```
Usage: (.. x form)
       (.. x form & more)

form => fieldName-symbol or (instanceMethodName-symbol args*)

Expands into a member access (.) of the first member on the first
argument, followed by the next member on the result, etc. For
instance:

(.. System (getProperties) (get "os.name"))

expands to:

(. (. System (getProperties)) (get "os.name"))

but is easier to write, read, and understand.
```

Added in Clojure version 1.0
Source

/

function

```
Usage: (/ x)
       (/ x y)
       (/ x y & more)

If no denominators are supplied, returns 1/numerator,
else returns numerator divided by all of the denominators.
```

Added in Clojure version 1.0
Source

<

function

```
Usage: (< x)
```

```
       (< x y)
       (< x y & more)

Returns non-nil if nums are in monotonically increasing order,
otherwise false.
```

Added in Clojure version 1.0
Source

<=

function

```
Usage: (<= x)
       (<= x y)
       (<= x y & more)

Returns non-nil if nums are in monotonically non-decreasing order,
otherwise false.
```

Added in Clojure version 1.0
Source

=

function

```
Usage: (= x)
       (= x y)
       (= x y & more)

Equality. Returns true if x equals y, false if not. Same as
Java x.equals(y) except it also works for nil, and compares
numbers and collections in a type-independent manner.  Clojure's immutable data
structures define equals() (and thus =) as a value, not an identity,
comparison.
```

Added in Clojure version 1.0
Source

==

function

```
Usage: (== x)
       (== x y)
       (== x y & more)

Returns non-nil if nums all have the equivalent
value (type-independent), otherwise false
```

Added in Clojure version 1.0
Source

>

function

```
Usage: (> x)
       (> x y)
       (> x y & more)

Returns non-nil if nums are in monotonically decreasing order,
otherwise false.
```

Added in Clojure version 1.0
Source

>=

function

```
Usage: (>= x)
       (>= x y)
       (>= x y & more)

Returns non-nil if nums are in monotonically non-increasing order,
otherwise false.
```

Added in Clojure version 1.0
Source

---

accessor

function

```
Usage: (accessor s key)
```

```
Returns a fn that, given an instance of a structmap with the basis,
returns the value at the key.  The key must be in the basis. The
returned function should be (slightly) more efficient than using
get, but such use of accessors should be limited to known
performance-critical areas.
```

Added in Clojure version 1.0
Source

---

aclone

function

```
Usage: (aclone array)

Returns a clone of the Java array. Works on arrays of known
types.
```

Added in Clojure version 1.0
Source

---

add-classpath

function

```
Usage: (add-classpath url)

DEPRECATED

Adds the url (String or URL object) to the classpath per
URLClassLoader.addURL
```

Added in Clojure version 1.0
Deprecated since Clojure version 1.1
Source

---

add-watch

function

```
Usage: (add-watch reference key fn)

Alpha - subject to change.
Adds a watch function to an agent/atom/var/ref reference. The watch
```

```
fn must be a fn of 4 args: a key, the reference, its old-state, its
new-state. Whenever the reference's state might have been changed,
any registered watches will have their functions called. The watch fn
will be called synchronously, on the agent's thread if an agent,
before any pending sends if agent or ref. Note that an atom's or
ref's state may have changed again prior to the fn call, so use
old/new-state rather than derefing the reference. Note also that watch
fns may be called from multiple threads simultaneously. Var watchers
are triggered only by root binding changes, not thread-local
set!s. Keys must be unique per reference, and can be used to remove
the watch with remove-watch, but are otherwise considered opaque by
the watch mechanism.
```

Added in Clojure version 1.0
Source

---

agent

function

```
Usage: (agent state & options)

Creates and returns an agent with an initial value of state and
zero or more options (in any order):

:meta metadata-map

:validator validate-fn

:error-handler handler-fn

:error-mode mode-keyword

If metadata-map is supplied, it will be come the metadata on the
agent. validate-fn must be nil or a side-effect-free fn of one
argument, which will be passed the intended new state on any state
change. If the new state is unacceptable, the validate-fn should
return false or throw an exception.  handler-fn is called if an
action throws an exception or if validate-fn rejects a new state --
see set-error-handler! for details.  The mode-keyword may be either
:continue (the default if an error-handler is given) or :fail (the
default if no error-handler is given) -- see set-error-mode! for
details.
```

Added in Clojure version 1.0
Source

---

agent-error

function

```
Usage: (agent-error a)

Returns the exception thrown during an asynchronous action of the
agent if the agent is failed.  Returns nil if the agent is not
failed.
```

Added in Clojure version 1.2
Source

---

agent-errors

function

```
Usage: (agent-errors a)

DEPRECATED: Use 'agent-error' instead.
Returns a sequence of the exceptions thrown during asynchronous
actions of the agent.
```

Added in Clojure version 1.0
Deprecated since Clojure version 1.2
Source

---

aget

function

```
Usage: (aget array idx)
       (aget array idx & idxs)

Returns the value at the index/indices. Works on Java arrays of all
types.
```

Added in Clojure version 1.0
Source

---

alength

function

```
Usage: (alength array)

Returns the length of the Java array. Works on arrays of all
types.
```

Added in Clojure version 1.0
Source

---

alias

function

```
Usage: (alias alias namespace-sym)

Add an alias in the current namespace to another
namespace. Arguments are two symbols: the alias to be used, and
the symbolic name of the target namespace. Use :as in the ns macro in preference
to calling this directly.
```

Added in Clojure version 1.0
Source

---

all-ns

function

```
Usage: (all-ns)

Returns a sequence of all namespaces.
```

Added in Clojure version 1.0
Source

---

alter

function

```
Usage: (alter ref fun & args)

Must be called in a transaction. Sets the in-transaction-value of
ref to:

(apply fun in-transaction-value-of-ref args)
```

```
and returns the in-transaction-value of ref.
```

Added in Clojure version 1.0
Source

---

alter-meta!

function

```
Usage: (alter-meta! iref f & args)

Atomically sets the metadata for a namespace/var/ref/agent/atom to be:

(apply f its-current-meta args)

f must be free of side-effects
```

Added in Clojure version 1.0
Source

---

alter-var-root

function

```
Usage: (alter-var-root v f & args)

Atomically alters the root binding of var v by applying f to its
current value plus any args
```

Added in Clojure version 1.0
Source

---

amap

macro

```
Usage: (amap a idx ret expr)

Maps an expression across an array a, using an index named idx, and
return value named ret, initialized to a clone of a, then setting
each element of ret to the evaluation of expr, returning the new
array ret.
```

Added in Clojure version 1.0

Source

---

ancestors

function

```
Usage: (ancestors tag)
       (ancestors h tag)

Returns the immediate and indirect parents of tag, either via a Java type
inheritance relationship or a relationship established via derive. h
must be a hierarchy obtained from make-hierarchy, if not supplied
defaults to the global hierarchy
```

Added in Clojure version 1.0
Source

---

and

macro

```
Usage: (and)
       (and x)
       (and x & next)

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 in Clojure version 1.0
Source

---

apply

function

```
Usage: (apply f args)
       (apply f x args)
       (apply f x y args)
       (apply f x y z args)
       (apply f a b c d & args)

Applies fn f to the argument list formed by prepending intervening arguments to args.
```

Added in Clojure version 1.0
Source

---

areduce

macro

```
Usage: (areduce a idx ret init expr)

Reduces an expression across an array a, using an index named idx,
and return value named ret, initialized to init, setting ret to the
evaluation of expr at each step, returning ret.
```

Added in Clojure version 1.0
Source

---

array-map

function

```
Usage: (array-map)
       (array-map & keyvals)

Constructs an array-map.
```

Added in Clojure version 1.0
Source

---

aset

function

```
Usage: (aset array idx val)
       (aset array idx idx2 & idxv)

Sets the value at the index/indices. Works on Java arrays of
reference types. Returns val.
```

Added in Clojure version 1.0
Source

---

aset-boolean

function

```
Usage: (aset-boolean array idx val)
       (aset-boolean array idx idx2 & idxv)

Sets the value at the index/indices. Works on arrays of boolean. Returns val.
```

Added in Clojure version 1.0
Source

---

aset-byte

function

```
Usage: (aset-byte array idx val)
       (aset-byte array idx idx2 & idxv)

Sets the value at the index/indices. Works on arrays of byte. Returns val.
```

Added in Clojure version 1.0
Source

---

aset-char

function

```
Usage: (aset-char array idx val)
       (aset-char array idx idx2 & idxv)

Sets the value at the index/indices. Works on arrays of char. Returns val.
```

Added in Clojure version 1.0
Source

---

aset-double

function

```
Usage: (aset-double array idx val)
       (aset-double array idx idx2 & idxv)

Sets the value at the index/indices. Works on arrays of double. Returns val.
```

Added in Clojure version 1.0
Source

---

aset-float

function

```
Usage: (aset-float array idx val)
       (aset-float array idx idx2 & idxv)

Sets the value at the index/indices. Works on arrays of float. Returns val.
```

Added in Clojure version 1.0
Source

---

aset-int

function

```
Usage: (aset-int array idx val)
       (aset-int array idx idx2 & idxv)

Sets the value at the index/indices. Works on arrays of int. Returns val.
```

Added in Clojure version 1.0
Source

---

aset-long

function

```
Usage: (aset-long array idx val)
       (aset-long array idx idx2 & idxv)

Sets the value at the index/indices. Works on arrays of long. Returns val.
```

Added in Clojure version 1.0
Source

---

aset-short

function

```
Usage: (aset-short array idx val)
       (aset-short array idx idx2 & idxv)

Sets the value at the index/indices. Works on arrays of short. Returns val.
```

Added in Clojure version 1.0
Source

---

assert

macro

```
Usage: (assert x)
       (assert x message)

Evaluates expr and throws an exception if it does not evaluate to
logical true.
```

Added in Clojure version 1.0
Source

---

assoc

function

```
Usage: (assoc map key val)
       (assoc map key val & kvs)

assoc[iate]. When applied to a map, returns a new map of the
same (hashed/sorted) type, that contains the mapping of key(s) to
val(s). When applied to a vector, returns a new vector that
contains val at index. Note - index must be <= (count vector).
```

Added in Clojure version 1.0
Source

---

assoc!

function

```
Usage: (assoc! coll key val)
       (assoc! coll key val & kvs)

Alpha - subject to change.
When applied to a transient map, adds mapping of key(s) to
val(s). When applied to a transient vector, sets the val at index.
Note - index must be <= (count vector). Returns coll.
```

Added in Clojure version 1.1
Source

---

assoc-in

function

```
Usage: (assoc-in m [k & ks] v)

Associates a value in a nested associative structure, where ks is a
sequence of keys and v is the new value and returns a new nested structure.
If any levels do not exist, hash-maps will be created.
```

Added in Clojure version 1.0
Source

---

associative?

function

```
Usage: (associative? coll)

Returns true if coll implements Associative
```

Added in Clojure version 1.0
Source

---

atom

function

```
Usage: (atom x)
       (atom x & options)

Creates and returns an Atom with an initial value of x and zero or
more options (in any order):
```

```
:meta metadata-map

:validator validate-fn

If metadata-map is supplied, it will be come the metadata on the
atom. validate-fn must be nil or a side-effect-free fn of one
argument, which will be passed the intended new state on any state
change. If the new state is unacceptable, the validate-fn should
return false or throw an exception.
```

Added in Clojure version 1.0
Source

---

await

function

```
Usage: (await & agents)

Blocks the current thread (indefinitely!) until all actions
dispatched thus far, from this thread or agent, to the agent(s) have
occurred.  Will block on failed agents.  Will never return if
a failed agent is restarted with :clear-actions true.
```

Added in Clojure version 1.0
Source

---

await-for

function

```
Usage: (await-for timeout-ms & agents)

Blocks the current thread until all actions dispatched thus
far (from this thread or agent) to the agents have occurred, or the
timeout (in milliseconds) has elapsed. Returns nil if returning due
to timeout, non-nil otherwise.
```

Added in Clojure version 1.0
Source

---

bases

function

```
Usage: (bases c)

Returns the immediate superclass and direct interfaces of c, if any
```

Added in Clojure version 1.0
Source

---

bean

function

```
Usage: (bean x)

Takes a Java object and returns a read-only implementation of the
map abstraction based upon its JavaBean properties.
```

Added in Clojure version 1.0
Source

---

bigdec

function

```
Usage: (bigdec x)

Coerce to BigDecimal
```

Added in Clojure version 1.0
Source

---

bigint

function

```
Usage: (bigint x)

Coerce to BigInt
```

Added in Clojure version 1.3
Source

---

biginteger

function

```
Usage: (biginteger x)

Coerce to BigInteger
```

Added in Clojure version 1.0
Source

---

binding

macro

```
Usage: (binding bindings & body)

binding => var-symbol init-expr

Creates new bindings for the (already-existing) vars, with the
supplied initial values, executes the exprs in an implicit do, then
re-establishes the bindings that existed before.  The new bindings
are made in parallel (unlike let); all init-exprs are evaluated
before the vars are bound to their new values.
```

Added in Clojure version 1.0
Source

---

bit-and

function

```
Usage: (bit-and x y)
       (bit-and x y & more)

Bitwise and
```

Added in Clojure version 1.0
Source

---

bit-and-not

function

```
Usage: (bit-and-not x y)
       (bit-and-not x y & more)

Bitwise and with complement
```

Added in Clojure version 1.0
Source

---

bit-clear

function

```
Usage: (bit-clear x n)

Clear bit at index n
```

Added in Clojure version 1.0
Source

---

bit-flip

function

```
Usage: (bit-flip x n)

Flip bit at index n
```

Added in Clojure version 1.0
Source

---

bit-not

function

```
Usage: (bit-not x)

Bitwise complement
```

Added in Clojure version 1.0
Source

---

bit-or

function

```
Usage: (bit-or x y)
       (bit-or x y & more)

Bitwise or
```

Added in Clojure version 1.0
Source

---

bit-set

function

```
Usage: (bit-set x n)

Set bit at index n
```

Added in Clojure version 1.0
Source

---

bit-shift-left

function

```
Usage: (bit-shift-left x n)

Bitwise shift left
```

Added in Clojure version 1.0
Source

---

bit-shift-right

function

```
Usage: (bit-shift-right x n)
```

```
Bitwise shift right
```

Added in Clojure version 1.0
Source

---

bit-test

function

```
Usage: (bit-test x n)

Test bit at index n
```

Added in Clojure version 1.0
Source

---

bit-xor

function

```
Usage: (bit-xor x y)
       (bit-xor x y & more)

Bitwise exclusive or
```

Added in Clojure version 1.0
Source

---

boolean

function

```
Usage: (boolean x)

Coerce to boolean
```

Added in Clojure version 1.0
Source

---

boolean-array

function

```
Usage: (boolean-array size-or-seq)
       (boolean-array size init-val-or-seq)

Creates an array of booleans
```

Added in Clojure version 1.1
Source

---

booleans

function

```
Usage: (booleans xs)

Casts to boolean[]
```

Added in Clojure version 1.1
Source

---

bound-fn

macro

```
Usage: (bound-fn & fntail)

Returns a function defined by the given fntail, which will install the
same bindings in effect as in the thread at the time bound-fn was called.
This may be used to define a helper function which runs on a different
thread, but needs the same bindings in place.
```

Added in Clojure version 1.1
Source

---

bound-fn*

function

```
Usage: (bound-fn* f)

Returns a function, which will install the same bindings in effect as in
```

```
the thread at the time bound-fn* was called and then call f with any given
arguments. This may be used to define a helper function which runs on a
different thread, but needs the same bindings in place.
```

Added in Clojure version 1.1
Source

---

bound?

function

```
Usage: (bound? & vars)

Returns true if all of the vars provided as arguments have any bound value, root or
thread-local.
Implies that deref'ing the provided vars will succeed. Returns true if no vars are
provided.
```

Added in Clojure version 1.2
Source

---

butlast

function

```
Usage: (butlast coll)

Return a seq of all but the last item in coll, in linear time
```

Added in Clojure version 1.0
Source

---

byte

function

```
Usage: (byte x)

Coerce to byte
```

Added in Clojure version 1.0
Source

---

byte-array

function

```
Usage: (byte-array size-or-seq)
       (byte-array size init-val-or-seq)

Creates an array of bytes
```

Added in Clojure version 1.1
Source

---

bytes

function

```
Usage: (bytes xs)

Casts to bytes[]
```

Added in Clojure version 1.1
Source

---

case

macro

```
Usage: (case e & clauses)

Takes an expression, and a set of clauses.

Each clause can take the form of either:

test-constant result-expr

(test-constant1 ... test-constantN)  result-expr

The test-constants are not evaluated. They must be compile-time
literals, and need not be quoted.  If the expression is equal to a
test-constant, the corresponding result-expr is returned. A single
default expression can follow the clauses, and its value will be
returned if no clause matches. If no default expression is provided
and no clause matches, an IllegalArgumentException is thrown.

Unlike cond and condp, case does a constant-time dispatch, the
clauses are not considered sequentially.  All manner of constant
```

```
expressions are acceptable in case, including numbers, strings,
symbols, keywords, and (Clojure) composites thereof. Note that since
lists are used to group multiple constants that map to the same
expression, a vector can be used to match a list if needed. The
test-constants need not be all of the same type.
```

Added in Clojure version 1.2
Source

---

cast

function

```
Usage: (cast c x)

Throws a ClassCastException if x is not a c, else returns x.
```

Added in Clojure version 1.0
Source

---

char

function

```
Usage: (char x)

Coerce to char
```

Added in Clojure version 1.1
Source

---

char-array

function

```
Usage: (char-array size-or-seq)
       (char-array size init-val-or-seq)

Creates an array of chars
```

Added in Clojure version 1.1
Source

---

char-escape-string

var

```
Returns escape string for char or nil if none
```

Added in Clojure version 1.0
Source

---

char-name-string

var

```
Returns name string for char or nil if none
```

Added in Clojure version 1.0
Source

---

char?

function

```
Usage: (char? x)

Return true if x is a Character
```

Added in Clojure version 1.0
Source

---

chars

function

```
Usage: (chars xs)

Casts to chars[]
```

Added in Clojure version 1.1
Source

---

class

function

```
Usage: (class x)

Returns the Class of x
```

Added in Clojure version 1.0
Source

---

class?

function

```
Usage: (class? x)

Returns true if x is an instance of Class
```

Added in Clojure version 1.0
Source

---

clear-agent-errors

function

```
Usage: (clear-agent-errors a)

DEPRECATED: Use 'restart-agent' instead.
Clears any exceptions thrown during asynchronous actions of the
agent, allowing subsequent actions to occur.
```

Added in Clojure version 1.0
Deprecated since Clojure version 1.2
Source

---

clojure-version

function

```
Usage: (clojure-version)

Returns clojure version as a printable string.
```

Added in Clojure version 1.0
Source

---

coll?

function

```
Usage: (coll? x)

Returns true if x implements IPersistentCollection
```

Added in Clojure version 1.0
Source

---

comment

macro

```
Usage: (comment & body)

Ignores body, yields nil
```

Added in Clojure version 1.0
Source

---

commute

function

```
Usage: (commute ref fun & args)

Must be called in a transaction. Sets the in-transaction-value of
ref to:

(apply fun in-transaction-value-of-ref args)

and returns the in-transaction-value of ref.

At the commit point of the transaction, sets the value of ref to be:

(apply fun most-recently-committed-value-of-ref args)

Thus fun should be commutative, or, failing that, you must accept
last-one-in-wins behavior.  commute allows for more concurrency than
ref-set.
```

Added in Clojure version 1.0
Source

---

comp

function

```
Usage: (comp)
       (comp f)
       (comp f g)
       (comp f g h)
       (comp f1 f2 f3 & fs)

Takes a set of functions and returns a fn that is the composition
of those fns.  The returned fn takes a variable number of args,
applies the rightmost of fns to the args, the next
fn (right-to-left) to the result, etc.
```

Added in Clojure version 1.0
Source

---

comparator

function

```
Usage: (comparator pred)

Returns an implementation of java.util.Comparator based upon pred.
```

Added in Clojure version 1.0
Source

---

compare

function

```
Usage: (compare x y)

Comparator. Returns a negative number, zero, or a positive number
when x is logically 'less than', 'equal to', or 'greater than'
y. Same as Java x.compareTo(y) except it also works for nil, and
compares numbers and collections in a type-independent manner. x
must implement Comparable
```

Added in Clojure version 1.0
Source

---

compare-and-set!

function

```
Usage: (compare-and-set! atom oldval newval)

Atomically sets the value of atom to newval if and only if the
current value of the atom is identical to oldval. Returns true if
set happened, else false
```

Added in Clojure version 1.0
Source

---

compile

function

```
Usage: (compile lib)

Compiles the namespace named by the symbol lib into a set of
classfiles. The source for the lib must be in a proper
classpath-relative directory. The output files will go into the
directory specified by *compile-path*, and that directory too must
be in the classpath.
```

Added in Clojure version 1.0
Source

---

complement

function

```
Usage: (complement f)

Takes a fn f and returns a fn that takes the same arguments as f,
has the same effects, if any, and returns the opposite truth value.
```

Added in Clojure version 1.0
Source

---

concat

function

```
Usage: (concat)
       (concat x)
       (concat x y)
       (concat x y & zs)

Returns a lazy seq representing the concatenation of the elements in the supplied colls.
```

Added in Clojure version 1.0
Source

---

cond

macro

```
Usage: (cond & clauses)

Takes a set of test/expr pairs. It evaluates each test one at a
time.  If a test returns logical true, cond evaluates and returns
the value of the corresponding expr and doesn't evaluate any of the
other tests or exprs. (cond) returns nil.
```

Added in Clojure version 1.0
Source

---

condp

macro

```
Usage: (condp pred expr & clauses)

Takes a binary predicate, an expression, and a set of clauses.
Each clause can take the form of either:

test-expr result-expr

test-expr :>> result-fn

Note :>> is an ordinary keyword.

For each clause, (pred test-expr expr) is evaluated. If it returns
logical true, the clause is a match. If a binary clause matches, the
result-expr is returned, if a ternary clause matches, its result-fn,
which must be a unary function, is called with the result of the
predicate as its argument, the result of that call being the return
value of condp. A single default expression can follow the clauses,
```

```
and its value will be returned if no clause matches. If no default
expression is provided and no clause matches, an
IllegalArgumentException is thrown.
```

Added in Clojure version 1.0
Source

---

conj

function

```
Usage: (conj coll x)
       (conj coll x & xs)

conj[oin]. Returns a new collection with the xs
'added'. (conj nil item) returns (item).  The 'addition' may
happen at different 'places' depending on the concrete type.
```

Added in Clojure version 1.0
Source

---

conj!

function

```
Usage: (conj! coll x)

Alpha - subject to change.
Adds x to the transient collection, and return coll. The 'addition'
may happen at different 'places' depending on the concrete type.
```

Added in Clojure version 1.1
Source

---

cons

function

```
Usage: (cons x seq)

Returns a new seq where x is the first element and seq is
the rest.
```

Added in Clojure version 1.0

Source

---

constantly

function

```
Usage: (constantly x)

Returns a function that takes any number of arguments and returns x.
```

Added in Clojure version 1.0
Source

---

construct-proxy

function

```
Usage: (construct-proxy c & ctor-args)

Takes a proxy class and any arguments for its superclass ctor and
creates and returns an instance of the proxy.
```

Added in Clojure version 1.0
Source

---

contains?

function

```
Usage: (contains? coll key)

Returns true if key is present in the given collection, otherwise
returns false.  Note that for numerically indexed collections like
vectors and Java arrays, this tests if the numeric key is within the
range of indexes. 'contains?' operates constant or logarithmic time;
it will not perform a linear search for a value.  See also 'some'.
```

Added in Clojure version 1.0
Source

---

count

function

```
Usage: (count coll)

Returns the number of items in the collection. (count nil) returns
0.  Also works on strings, arrays, and Java Collections and Maps
```

Added in Clojure version 1.0
Source

---

counted?

function

```
Usage: (counted? coll)

Returns true if coll implements count in constant time
```

Added in Clojure version 1.0
Source

---

create-ns

function

```
Usage: (create-ns sym)

Create a new namespace named by the symbol if one doesn't already
exist, returns it or the already-existing namespace of the same
name.
```

Added in Clojure version 1.0
Source

---

create-struct

function

```
Usage: (create-struct & keys)

Returns a structure basis object.
```

Added in Clojure version 1.0
Source

---

cycle

function

```
Usage: (cycle coll)

Returns a lazy (infinite!) sequence of repetitions of the items in coll.
```

Added in Clojure version 1.0
Source

---

dec

function

```
Usage: (dec x)

Returns a number one less than num. Does not auto-promote
longs, will throw on overflow. See also: dec'
```

Added in Clojure version 1.2
Source

---

dec'

function

```
Usage: (dec' x)

Returns a number one less than num. Supports arbitrary precision.
See also: dec
```

Added in Clojure version 1.0
Source

---

decimal?

function

```
Usage: (decimal? n)

Returns true if n is a BigDecimal
```

Added in Clojure version 1.0
Source

---

declare

macro

```
Usage: (declare & names)

defs the supplied var names with no bindings, useful for making forward declarations.
```

Added in Clojure version 1.0
Source

---

definline

macro

```
Usage: (definline name & decl)

Experimental - like defmacro, except defines a named function whose
body is the expansion, calls to which may be expanded inline as if
it were a macro. Cannot be used with variadic (&) args.
```

Added in Clojure version 1.0
Source

---

defmacro

macro

```
Usage: (defmacro name doc-string? attr-map? [params*] body)
       (defmacro name doc-string? attr-map? ([params*] body) + attr-map?)

Like defn, but the resulting function name is declared as a
macro and will be used as a macro by the compiler when it is
```

```
called.
```

Added in Clojure version 1.0
Source

---

defmethod

macro

```
Usage: (defmethod multifn dispatch-val & fn-tail)

Creates and installs a new method of multimethod associated with dispatch-value.
```

Added in Clojure version 1.0
Source

---

defmulti

macro

```
Usage: (defmulti name docstring? attr-map? dispatch-fn & options)

Creates a new multimethod with the associated dispatch function.
The docstring and attribute-map are optional.

Options are key-value pairs and may be one of:
  :default    the default dispatch value, defaults to :default
  :hierarchy  the isa? hierarchy to use for dispatching
              defaults to the global hierarchy
```

Added in Clojure version 1.0
Source

---

defn

macro

```
Usage: (defn name doc-string? attr-map? [params*] body)
       (defn name doc-string? attr-map? ([params*] body) + attr-map?)

Same as (def name (fn [params* ] exprs*)) or (def
name (fn ([params* ] exprs*)+)) with any doc-string or attrs added
to the var metadata
```

Added in Clojure version 1.0
Source

---

defn-

macro

```
Usage: (defn- name & decls)

same as defn, yielding non-public def
```

Added in Clojure version 1.0
Source

---

defonce

macro

```
Usage: (defonce name expr)

defs name to have the root value of the expr iff the named var has no root value,
else expr is unevaluated
```

Added in Clojure version 1.0
Source

---

defprotocol

macro

```
Usage: (defprotocol name & opts+sigs)

A protocol is a named set of named methods and their signatures:
(defprotocol AProtocolName

  ;optional doc string
  "A doc string for AProtocol abstraction"

;method signatures
  (bar [this a b] "bar docs")
  (baz [this a] [this a b] [this a b c] "baz docs"))

No implementations are provided. Docs can be specified for the
protocol overall and for each method. The above yields a set of
polymorphic functions and a protocol object. All are
```

```
namespace-qualified by the ns enclosing the definition The resulting
functions dispatch on the type of their first argument, which is
required and corresponds to the implicit target object ('this' in
Java parlance). defprotocol is dynamic, has no special compile-time
effect, and defines no new types or classes. Implementations of
the protocol methods can be provided using extend.

defprotocol will automatically generate a corresponding interface,
with the same name as the protocol, i.e. given a protocol:
my.ns/Protocol, an interface: my.ns.Protocol. The interface will
have methods corresponding to the protocol functions, and the
protocol will automatically work with instances of the interface.

Note that you should not use this interface with deftype or
reify, as they support the protocol directly:

(defprotocol P
  (foo [this])
  (bar-me [this] [this y]))

(deftype Foo [a b c]
 P
  (foo [this] a)
  (bar-me [this] b)
  (bar-me [this y] (+ c y)))

(bar-me (Foo. 1 2 3) 42)
=> 45

(foo
  (let [x 42]
    (reify P
      (foo [this] 17)
      (bar-me [this] x)
      (bar-me [this y] x))))
=> 17
```

Added in Clojure version 1.2
Source

---

defrecord

macro

```
Usage: (defrecord name [& fields] & opts+specs)

Alpha - subject to change

(defrecord name [fields*]  options* specs*)

Currently there are no options.

Each spec consists of a protocol or interface name followed by zero
```

```
or more method bodies:

protocol-or-interface-or-Object
(methodName [args*] body)*

Dynamically generates compiled bytecode for class with the given
name, in a package with the same name as the current namespace, the
given fields, and, optionally, methods for protocols and/or
interfaces.

The class will have the (immutable) fields named by
fields, which can have type hints. Protocols/interfaces and methods
are optional. The only methods that can be supplied are those
declared in the protocols/interfaces.  Note that method bodies are
not closures, the local environment includes only the named fields,
and those fields can be accessed directy.

Method definitions take the form:

(methodname [args*] body)

The argument and return types can be hinted on the arg and
methodname symbols. If not supplied, they will be inferred, so type
hints should be reserved for disambiguation.

Methods should be supplied for all methods of the desired
protocol(s) and interface(s). You can also define overrides for
methods of Object. Note that a parameter must be supplied to
correspond to the target object ('this' in Java parlance). Thus
methods for interfaces will take one more argument than do the
interface declarations. Note also that recur calls to the method
head should *not* pass the target object, it will be supplied
automatically and can not be substituted.

In the method bodies, the (unqualified) name can be used to name the
class (for calls to new, instance? etc).

The class will have implementations of several (clojure.lang)
interfaces generated automatically: IObj (metadata support) and
IPersistentMap, and all of their superinterfaces.

In addition, defrecord will define type-and-value-based =,
and will defined Java .hashCode and .equals consistent with the
contract for java.util.Map.

When AOT compiling, generates compiled bytecode for a class with the
given name (a symbol), prepends the current ns as the package, and
writes the .class file to the *compile-path* directory.

Two constructors will be defined, one taking the designated fields
followed by a metadata map (nil for none) and an extension field
map (nil for none), and one taking only the fields (using nil for
meta and extension fields).
```

Added in Clojure version 1.2
Source

---

defstruct

macro

```
Usage: (defstruct name & keys)

Same as (def name (create-struct keys...))
```

Added in Clojure version 1.0
Source

---

deftype

macro

```
Usage: (deftype name [& fields] & opts+specs)

Alpha - subject to change

(deftype name [fields*]  options* specs*)

Currently there are no options.

Each spec consists of a protocol or interface name followed by zero
or more method bodies:

protocol-or-interface-or-Object
(methodName [args*] body)*

Dynamically generates compiled bytecode for class with the given
name, in a package with the same name as the current namespace, the
given fields, and, optionally, methods for protocols and/or
interfaces.

The class will have the (by default, immutable) fields named by
fields, which can have type hints. Protocols/interfaces and methods
are optional. The only methods that can be supplied are those
declared in the protocols/interfaces.  Note that method bodies are
not closures, the local environment includes only the named fields,
and those fields can be accessed directy. Fields can be qualified
with the metadata :volatile-mutable true or :unsynchronized-mutable
true, at which point (set! afield aval) will be supported in method
bodies. Note well that mutable fields are extremely difficult to use
correctly, and are present only to facilitate the building of higher
level constructs, such as Clojure's reference types, in Clojure
itself. They are for experts only - if the semantics and
implications of :volatile-mutable or :unsynchronized-mutable are not
immediately apparent to you, you should not be using them.
```

```
Method definitions take the form:

(methodname [args*] body)

The argument and return types can be hinted on the arg and
methodname symbols. If not supplied, they will be inferred, so type
hints should be reserved for disambiguation.

Methods should be supplied for all methods of the desired
protocol(s) and interface(s). You can also define overrides for
methods of Object. Note that a parameter must be supplied to
correspond to the target object ('this' in Java parlance). Thus
methods for interfaces will take one more argument than do the
interface declarations. Note also that recur calls to the method
head should *not* pass the target object, it will be supplied
automatically and can not be substituted.

In the method bodies, the (unqualified) name can be used to name the
class (for calls to new, instance? etc).

When AOT compiling, generates compiled bytecode for a class with the
given name (a symbol), prepends the current ns as the package, and
writes the .class file to the *compile-path* directory.

One constructors will be defined, taking the designated fields.
```

Added in Clojure version 1.2
Source

---

delay

macro

```
Usage: (delay & body)

Takes a body of expressions and yields a Delay object that will
invoke the body only the first time it is forced (with force or deref/@), and
will cache the result and return it on all subsequent force
calls. See also - realized?
```

Added in Clojure version 1.0
Source

---

delay?

function

```
Usage: (delay? x)
```

```
returns true if x is a Delay created with delay
```

Added in Clojure version 1.0
Source

---

deliver

function

```
Usage: (deliver promise val)

Alpha - subject to change.
Delivers the supplied value to the promise, releasing any pending
derefs. A subsequent call to deliver on a promise will throw an exception.
```

Added in Clojure version 1.1
Source

---

denominator

function

```
Usage: (denominator r)

Returns the denominator part of a Ratio.
```

Added in Clojure version 1.2
Source

---

deref

function

```
Usage: (deref ref)
       (deref ref timeout-ms timeout-val)

Also reader macro: @ref/@agent/@var/@atom/@delay/@future/@promise. Within a transaction,
returns the in-transaction-value of ref, else returns the
most-recently-committed value of ref. When applied to a var, agent
or atom, returns its current state. When applied to a delay, forces
it if not already forced. When applied to a future, will block if
computation not complete. When applied to a promise, will block
until a value is delivered.  The variant taking a timeout can be
used for blocking references (futures and promises), and will return
```

```
timeout-val if the timeout (in milliseconds) is reached before a
value is available. See also - realized?.
```

Added in Clojure version 1.0
Source

---

derive

function

```
Usage: (derive tag parent)
       (derive h tag parent)

Establishes a parent/child relationship between parent and
tag. Parent must be a namespace-qualified symbol or keyword and
child can be either a namespace-qualified symbol or keyword or a
class. h must be a hierarchy obtained from make-hierarchy, if not
supplied defaults to, and modifies, the global hierarchy.
```

Added in Clojure version 1.0
Source

---

descendants

function

```
Usage: (descendants tag)
       (descendants h tag)

Returns the immediate and indirect children of tag, through a
relationship established via derive. h must be a hierarchy obtained
from make-hierarchy, if not supplied defaults to the global
hierarchy. Note: does not work on Java type inheritance
relationships.
```

Added in Clojure version 1.0
Source

---

disj

function

```
Usage: (disj set)
       (disj set key)
       (disj set key & ks)
```

```
disj[oin]. Returns a new set of the same (hashed/sorted) type, that
does not contain key(s).
```

Added in Clojure version 1.0
Source

---

disj!

function

```
Usage: (disj! set)
       (disj! set key)
       (disj! set key & ks)

Alpha - subject to change.
disj[oin]. Returns a transient set of the same (hashed/sorted) type, that
does not contain key(s).
```

Added in Clojure version 1.1
Source

---

dissoc

function

```
Usage: (dissoc map)
       (dissoc map key)
       (dissoc map key & ks)

dissoc[iate]. Returns a new map of the same (hashed/sorted) type,
that does not contain a mapping for key(s).
```

Added in Clojure version 1.0
Source

---

dissoc!

function

```
Usage: (dissoc! map key)
       (dissoc! map key & ks)

Alpha - subject to change.
Returns a transient map that doesn't contain a mapping for key(s).
```

Added in Clojure version 1.1
Source

---

distinct

function

```
Usage: (distinct coll)

Returns a lazy sequence of the elements of coll with duplicates removed
```

Added in Clojure version 1.0
Source

---

distinct?

function

```
Usage: (distinct? x)
       (distinct? x y)
       (distinct? x y & more)

Returns true if no two of the arguments are =
```

Added in Clojure version 1.0
Source

---

doall

function

```
Usage: (doall coll)
       (doall n coll)

When lazy sequences are produced via functions that have side
effects, any effects other than those needed to produce the first
element in the seq do not occur until the seq is consumed. doall can
be used to force any effects. Walks through the successive nexts of
the seq, retains the head and returns it, thus causing the entire
seq to reside in memory at one time.
```

Added in Clojure version 1.0
Source

---

dorun

function

```
Usage: (dorun coll)
       (dorun n coll)

When lazy sequences are produced via functions that have side
effects, any effects other than those needed to produce the first
element in the seq do not occur until the seq is consumed. dorun can
be used to force any effects. Walks through the successive nexts of
the seq, does not retain the head and returns nil.
```

Added in Clojure version 1.0
Source

---

doseq

macro

```
Usage: (doseq seq-exprs & body)

Repeatedly executes body (presumably for side-effects) with
bindings and filtering as provided by "for".  Does not retain
the head of the sequence. Returns nil.
```

Added in Clojure version 1.0
Source

---

dosync

macro

```
Usage: (dosync & exprs)

Runs the exprs (in an implicit do) in a transaction that encompasses
exprs and any nested calls.  Starts a transaction if none is already
running on this thread. Any uncaught exception will abort the
transaction and flow out of dosync. The exprs may be run more than
once, but any effects on Refs will be atomic.
```

Added in Clojure version 1.0
Source

---

dotimes

macro

```
Usage: (dotimes bindings & body)

bindings => name n

Repeatedly executes body (presumably for side-effects) with name
bound to integers from 0 through n-1.
```

Added in Clojure version 1.0
Source

---

doto

macro

```
Usage: (doto x & forms)

Evaluates x then calls all of the methods and functions with the
value of x supplied at the front of the given arguments.  The forms
are evaluated in order.  Returns x.

(doto (new java.util.HashMap) (.put "a" 1) (.put "b" 2))
```

Added in Clojure version 1.0
Source

---

double

function

```
Usage: (double x)

Coerce to double
```

Added in Clojure version 1.0
Source

---

double-array

function

```
Usage: (double-array size-or-seq)
       (double-array size init-val-or-seq)

Creates an array of doubles
```

Added in Clojure version 1.0
Source

---

doubles

function

```
Usage: (doubles xs)

Casts to double[]
```

Added in Clojure version 1.0
Source

---

drop

function

```
Usage: (drop n coll)

Returns a lazy sequence of all but the first n items in coll.
```

Added in Clojure version 1.0
Source

---

drop-last

function

```
Usage: (drop-last s)
       (drop-last n s)

Return a lazy sequence of all but the last n (default 1) items in coll
```

Added in Clojure version 1.0

Source

---

drop-while

function

```
Usage: (drop-while pred coll)

Returns a lazy sequence of the items in coll starting from the first
item for which (pred item) returns nil.
```

Added in Clojure version 1.0
Source

---

empty

function

```
Usage: (empty coll)

Returns an empty collection of the same category as coll, or nil
```

Added in Clojure version 1.0
Source

---

empty?

function

```
Usage: (empty? coll)

Returns true if coll has no items - same as (not (seq coll)).
Please use the idiom (seq x) rather than (not (empty? x))
```

Added in Clojure version 1.0
Source

---

ensure

function

```
Usage: (ensure ref)

Must be called in a transaction. Protects the ref from modification
by other transactions.  Returns the in-transaction-value of
ref. Allows for more concurrency than (ref-set ref @ref)
```

Added in Clojure version 1.0
Source

---

enumeration-seq

function

```
Usage: (enumeration-seq e)

Returns a seq on a java.util.Enumeration
```

Added in Clojure version 1.0
Source

---

error-handler

function

```
Usage: (error-handler a)

Returns the error-handler of agent a, or nil if there is none.
See set-error-handler!
```

Added in Clojure version 1.2
Source

---

error-mode

function

```
Usage: (error-mode a)

Returns the error-mode of agent a.  See set-error-mode!
```

Added in Clojure version 1.2
Source

---

eval

function

```
Usage: (eval form)

Evaluates the form data structure (not text!) and returns the result.
```

Added in Clojure version 1.0
Source

---

even?

function

```
Usage: (even? n)

Returns true if n is even, throws an exception if n is not an integer
```

Added in Clojure version 1.0
Source

---

every-pred

function

```
Usage: (every-pred p)
       (every-pred p1 p2)
       (every-pred p1 p2 p3)
       (every-pred p1 p2 p3 & ps)

Takes a set of predicates and returns a function f that returns true if all of its
composing predicates return a logical true value against all of its arguments, else it
returns
false. Note that f is short-circuiting in that it will stop execution on the first
argument that triggers a logical false result against the original predicates.
```

Added in Clojure version 1.3
Source

---

every?

function

```
Usage: (every? pred coll)

Returns true if (pred x) is logical true for every x in coll, else
false.
```

Added in Clojure version 1.0
Source

---

extend

function

```
Usage: (extend atype & proto+mmaps)

Implementations of protocol methods can be provided using the extend construct:

 (extend AType
   AProtocol
    {:foo an-existing-fn
     :bar (fn [a b] ...)
     :baz (fn ([a]...) ([a b] ...)...)}
   BProtocol
     {...}
   ...)

 extend takes a type/class (or interface, see below), and one or more
 protocol + method map pairs. It will extend the polymorphism of the
 protocol's methods to call the supplied methods when an AType is
 provided as the first argument.

 Method maps are maps of the keyword-ized method names to ordinary
 fns. This facilitates easy reuse of existing fns and fn maps, for
 code reuse/mixins without derivation or composition. You can extend
 an interface to a protocol. This is primarily to facilitate interop
 with the host (e.g. Java) but opens the door to incidental multiple
 inheritance of implementation since a class can inherit from more
 than one interface, both of which extend the protocol. It is TBD how
 to specify which impl to use. You can extend a protocol on nil.

 If you are supplying the definitions explicitly (i.e. not reusing
 exsting functions or mixin maps), you may find it more convenient to
 use the extend-type or extend-protocol macros.

 Note that multiple independent extend clauses can exist for the same
 type, not all protocols need be defined in a single extend call.

 See also:
 extends?, satisfies?, extenders
```

Added in Clojure version 1.2
Source

---

extend-protocol

macro

```
Usage: (extend-protocol p & specs)

Useful when you want to provide several implementations of the same
protocol all at once. Takes a single protocol and the implementation
of that protocol for one or more types. Expands into calls to
extend-type:

(extend-protocol Protocol
  AType
    (foo [x] ...)
    (bar [x y] ...)
  BType
    (foo [x] ...)
    (bar [x y] ...)
  AClass
    (foo [x] ...)
    (bar [x y] ...)
  nil
    (foo [x] ...)
    (bar [x y] ...))

expands into:

(do
 (clojure.core/extend-type AType Protocol
   (foo [x] ...)
   (bar [x y] ...))
 (clojure.core/extend-type BType Protocol
   (foo [x] ...)
   (bar [x y] ...))
 (clojure.core/extend-type AClass Protocol
   (foo [x] ...)
   (bar [x y] ...))
 (clojure.core/extend-type nil Protocol
   (foo [x] ...)
   (bar [x y] ...)))
```

Added in Clojure version 1.2
Source

---

extend-type

macro

```
Usage: (extend-type t & specs)

A macro that expands into an extend call. Useful when you are
supplying the definitions explicitly inline, extend-type
automatically creates the maps required by extend.  Propagates the
class as a type hint on the first argument of all fns.

(extend-type MyType
  Countable
    (cnt [c] ...)
  Foo
    (bar [x y] ...)
    (baz ([x] ...) ([x y & zs] ...)))

expands into:

(extend MyType
 Countable
   {:cnt (fn [c] ...)}
 Foo
   {:baz (fn ([x] ...) ([x y & zs] ...))
    :bar (fn [x y] ...)})
```

Added in Clojure version 1.2
Source

---

extenders

function

```
Usage: (extenders protocol)

Returns a collection of the types explicitly extending protocol
```

Added in Clojure version 1.2
Source

---

extends?

function

```
Usage: (extends? protocol atype)

Returns true if atype extends protocol
```

Added in Clojure version 1.2
Source

---

false?

function

```
Usage: (false? x)

Returns true if x is the value false, false otherwise.
```

Added in Clojure version 1.0
Source

---

ffirst

function

```
Usage: (ffirst x)

Same as (first (first x))
```

Added in Clojure version 1.0
Source

---

file-seq

function

```
Usage: (file-seq dir)

A tree seq on java.io.Files
```

Added in Clojure version 1.0
Source

---

filter

function

```
Usage: (filter pred coll)

Returns a lazy sequence of the items in coll for which
(pred item) returns true. pred must be free of side-effects.
```

Added in Clojure version 1.0
Source

---

find

function

```
Usage: (find map key)

Returns the map entry for key, or nil if key not present.
```

Added in Clojure version 1.0
Source

---

find-keyword

function

```
Usage: (find-keyword name)
       (find-keyword ns name)

Returns a Keyword with the given namespace and name if one already
exists.  This function will not intern a new keyword. If the keyword
has not already been interned, it will return nil.  Do not use :
in the keyword strings, it will be added automatically.
```

Added in Clojure version 1.3
Source

---

find-ns

function

```
Usage: (find-ns sym)

Returns the namespace named by the symbol or nil if it doesn't exist.
```

Added in Clojure version 1.0
Source

---

find-var

function

```
Usage: (find-var sym)

Returns the global var named by the namespace-qualified symbol, or
nil if no var with that name.
```

Added in Clojure version 1.0
Source

---

first

function

```
Usage: (first coll)

Returns the first item in the collection. Calls seq on its
argument. If coll is nil, returns nil.
```

Added in Clojure version 1.0
Source

---

flatten

function

```
Usage: (flatten x)

Takes any nested combination of sequential things (lists, vectors,
etc.) and returns their contents as a single, flat sequence.
(flatten nil) returns nil.
```

Added in Clojure version 1.2
Source

---

float

function

```
Usage: (float x)

Coerce to float
```

Added in Clojure version 1.0
Source

---

float-array

function

```
Usage: (float-array size-or-seq)
       (float-array size init-val-or-seq)

Creates an array of floats
```

Added in Clojure version 1.0
Source

---

float?

function

```
Usage: (float? n)

Returns true if n is a floating point number
```

Added in Clojure version 1.0
Source

---

floats

function

```
Usage: (floats xs)

Casts to float[]
```

Added in Clojure version 1.0
Source

---

flush

function

```
Usage: (flush)

Flushes the output stream that is the current value of
*out*
```

Added in Clojure version 1.0
Source

---

fn

macro

```
Usage: (fn & sigs)

params => positional-params* , or positional-params* & next-param
positional-param => binding-form
next-param => binding-form
name => symbol

Defines a function
```

Added in Clojure version 1.0
Source

---

fn?

function

```
Usage: (fn? x)

Returns true if x implements Fn, i.e. is an object created via fn.
```

Added in Clojure version 1.0
Source

---

fnext

function

```
Usage: (fnext x)
```

```
Same as (first (next x))
```

Added in Clojure version 1.0
Source

---

fnil

function

```
Usage: (fnil f x)
       (fnil f x y)
       (fnil f x y z)

Takes a function f, and returns a function that calls f, replacing
a nil first argument to f with the supplied value x. Higher arity
versions can replace arguments in the second and third
positions (y, z). Note that the function f can take any number of
arguments, not just the one(s) being nil-patched.
```

Added in Clojure version 1.2
Source

---

for

macro

```
Usage: (for seq-exprs body-expr)

List comprehension. Takes a vector of one or more
 binding-form/collection-expr pairs, each followed by zero or more
 modifiers, and yields a lazy sequence of evaluations of expr.
 Collections are iterated in a nested fashion, rightmost fastest,
 and nested coll-exprs can refer to bindings created in prior
 binding-forms.  Supported modifiers are: :let [binding-form expr ...],
 :while test, :when test.

(take 100 (for [x (range 100000000) y (range 1000000) :while (< y x)] [x y]))
```

Added in Clojure version 1.0
Source

---

force

function

```
Usage: (force x)

If x is a Delay, returns the (possibly cached) value of its expression, else returns x
```

Added in Clojure version 1.0
Source

---

format

function

```
Usage: (format fmt & args)

Formats a string using java.lang.String.format, see java.util.Formatter for format
string syntax
```

Added in Clojure version 1.0
Source

---

frequencies

function

```
Usage: (frequencies coll)

Returns a map from distinct items in coll to the number of times
they appear.
```

Added in Clojure version 1.2
Source

---

future

macro

```
Usage: (future & body)

Takes a body of expressions and yields a future object that will
invoke the body in another thread, and will cache the result and
return it on all subsequent calls to deref/@. If the computation has
not yet finished, calls to deref/@ will block, unless the variant of
deref with timeout is used. See also - realized?.
```

Added in Clojure version 1.1

Source

---

future-call

function

```
Usage: (future-call f)

Takes a function of no args and yields a future object that will
invoke the function in another thread, and will cache the result and
return it on all subsequent calls to deref/@. If the computation has
not yet finished, calls to deref/@ will block, unless the variant
of deref with timeout is used. See also - realized?.
```

Added in Clojure version 1.1
Source

---

future-cancel

function

```
Usage: (future-cancel f)

Cancels the future, if possible.
```

Added in Clojure version 1.1
Source

---

future-cancelled?

function

```
Usage: (future-cancelled? f)

Returns true if future f is cancelled
```

Added in Clojure version 1.1
Source

---

future-done?

function

```
Usage: (future-done? f)

Returns true if future f is done
```

Added in Clojure version 1.1
Source

---

future?

function

```
Usage: (future? x)

Returns true if x is a future
```

Added in Clojure version 1.1
Source

---

gen-class

macro

```
Usage: (gen-class & options)

When compiling, generates compiled bytecode for a class with the
given package-qualified :name (which, as all names in these
parameters, can be a string or symbol), and writes the .class file
to the *compile-path* directory.  When not compiling, does
nothing. The gen-class construct contains no implementation, as the
implementation will be dynamically sought by the generated class in
functions in an implementing Clojure namespace. Given a generated
class org.mydomain.MyClass with a method named mymethod, gen-class
will generate an implementation that looks for a function named by
(str prefix mymethod) (default prefix: "-") in a
Clojure namespace specified by :impl-ns
(defaults to the current namespace). All inherited methods,
generated methods, and init and main functions (see :methods, :init,
and :main below) will be found similarly prefixed. By default, the
static initializer for the generated class will attempt to load the
Clojure support code for the class as a resource from the classpath,
e.g. in the example case, ``org/mydomain/MyClass__init.class``. This
behavior can be controlled by :load-impl-ns

Note that methods with a maximum of 18 parameters are supported.
```

In all subsequent sections taking types, the primitive types can be
referred to by their Java names (int, float etc), and classes in the
java.lang package can be used without a package qualifier. All other
classes must be fully qualified.

Options should be a set of key/value pairs, all except for :name are optional:

:name aname

The package-qualified name of the class to be generated

:extends aclass

Specifies the superclass, the non-private methods of which will be
overridden by the class. If not provided, defaults to Object.

:implements [interface ...]

One or more interfaces, the methods of which will be implemented by the class.

:init name

If supplied, names a function that will be called with the arguments
to the constructor. Must return [[superclass-constructor-args] state]
If not supplied, the constructor args are passed directly to
the superclass constructor and the state will be nil

:constructors {[param-types] [super-param-types], ...}

By default, constructors are created for the generated class which
match the signature(s) of the constructors for the superclass. This
parameter may be used to explicitly specify constructors, each entry
providing a mapping from a constructor signature to a superclass
constructor signature. When you supply this, you must supply an :init
specifier.

:post-init name

If supplied, names a function that will be called with the object as
the first argument, followed by the arguments to the constructor.
It will be called every time an object of this class is created,
immediately after all the inherited constructors have completed.
It's return value is ignored.

:methods [[name [param-types] return-type], ...]

The generated class automatically defines all of the non-private
methods of its superclasses/interfaces. This parameter can be used
to specify the signatures of additional methods of the generated
class. Static methods can be specified with ^{:static true} in the
signature's metadata. Do not repeat superclass/interface signatures
here.

:main boolean

If supplied and true, a static public main function will be generated. It will
pass each string of the String[] argument as a separate argument to
a function called (str prefix main).

```
:factory name

If supplied, a (set of) public static factory function(s) will be
created with the given name, and the same signature(s) as the
constructor(s).

:state name

If supplied, a public final instance field with the given name will be
created. You must supply an :init function in order to provide a
value for the state. Note that, though final, the state can be a ref
or agent, supporting the creation of Java objects with transactional
or asynchronous mutation semantics.

:exposes {protected-field-name {:get name :set name}, ...}

Since the implementations of the methods of the generated class
occur in Clojure functions, they have no access to the inherited
protected fields of the superclass. This parameter can be used to
generate public getter/setter methods exposing the protected field(s)
for use in the implementation.

:exposes-methods {super-method-name exposed-name, ...}

It is sometimes necessary to call the superclass' implementation of an
overridden method.  Those methods may be exposed and referred in
the new method implementation by a local name.

:prefix string

Default: "-" Methods called e.g. Foo will be looked up in vars called
prefixFoo in the implementing ns.

:impl-ns name

Default: the name of the current ns. Implementations of methods will be
looked up in this namespace.

:load-impl-ns boolean

Default: true. Causes the static initializer for the generated class
to reference the load code for the implementing namespace. Should be
true when implementing-ns is the default, false if you intend to
load the code via some other method.
```

Added in Clojure version 1.0
Source

---

gen-interface

macro

```
Usage: (gen-interface & options)

When compiling, generates compiled bytecode for an interface with
 the given package-qualified :name (which, as all names in these
 parameters, can be a string or symbol), and writes the .class file
 to the *compile-path* directory.  When not compiling, does nothing.

 In all subsequent sections taking types, the primitive types can be
 referred to by their Java names (int, float etc), and classes in the
 java.lang package can be used without a package qualifier. All other
 classes must be fully qualified.

 Options should be a set of key/value pairs, all except for :name are
 optional:

 :name aname

 The package-qualified name of the class to be generated

 :extends [interface ...]

 One or more interfaces, which will be extended by this interface.

 :methods [ [name [param-types] return-type], ...]

 This parameter is used to specify the signatures of the methods of
 the generated interface.  Do not repeat superinterface signatures
 here.
```

Added in Clojure version 1.0
Source

---

gensym

function

```
Usage: (gensym)
       (gensym prefix-string)

Returns a new symbol with a unique name. If a prefix string is
supplied, the name is prefix# where # is some unique number. If
prefix is not supplied, the prefix is 'G__'.
```

Added in Clojure version 1.0
Source

---

get

function

```
Usage: (get map key)
       (get map key not-found)

Returns the value mapped to key, not-found or nil if key not present.
```

Added in Clojure version 1.0
Source

---

get-in

function

```
Usage: (get-in m ks)
       (get-in m ks not-found)

Returns the value in a nested associative structure,
where ks is a sequence of ke(ys. Returns nil if the key is not present,
or the not-found value if supplied.
```

Added in Clojure version 1.2
Source

---

get-method

function

```
Usage: (get-method multifn dispatch-val)

Given a multimethod and a dispatch value, returns the dispatch fn
that would apply to that value, or nil if none apply and no default
```

Added in Clojure version 1.0
Source

---

get-proxy-class

function

```
Usage: (get-proxy-class & bases)
```

```
Takes an optional single class followed by zero or more
interfaces. If not supplied class defaults to Object.  Creates an
returns an instance of a proxy class derived from the supplied
classes. The resulting value is cached and used for any subsequent
requests for the same class set. Returns a Class object.
```

Added in Clojure version 1.0
Source

---

get-thread-bindings

function

```
Usage: (get-thread-bindings)

Get a map with the Var/value pairs which is currently in effect for the
current thread.
```

Added in Clojure version 1.1
Source

---

get-validator

function

```
Usage: (get-validator iref)

Gets the validator-fn for a var/ref/agent/atom.
```

Added in Clojure version 1.0
Source

---

group-by

function

```
Usage: (group-by f coll)

Returns a map of the elements of coll keyed by the result of
f on each element. The value at each key will be a vector of the
corresponding elements, in the order they appeared in coll.
```

Added in Clojure version 1.2
Source

---

hash

function

```
Usage: (hash x)

Returns the hash code of its argument
```

Added in Clojure version 1.0
Source

---

hash-map

function

```
Usage: (hash-map)
       (hash-map & keyvals)

keyval => key val
Returns a new hash map with supplied mappings.
```

Added in Clojure version 1.0
Source

---

hash-set

function

```
Usage: (hash-set)
       (hash-set & keys)

Returns a new hash set with supplied keys.
```

Added in Clojure version 1.0
Source

---

identical?

function

```
Usage: (identical? x y)

Tests if 2 arguments are the same object
```

Added in Clojure version 1.0
Source

---

identity

function

```
Usage: (identity x)

Returns its argument.
```

Added in Clojure version 1.0
Source

---

if-let

macro

```
Usage: (if-let bindings then)
       (if-let bindings then else & oldform)

bindings => binding-form test

If test is true, evaluates then with binding-form bound to the value of
test, if not, yields else
```

Added in Clojure version 1.0
Source

---

if-not

macro

```
Usage: (if-not test then)
       (if-not test then else)

Evaluates test. If logical false, evaluates and returns then expr,
otherwise else expr, if supplied, else nil.
```

Added in Clojure version 1.0

Source

---

ifn?

function

```
Usage: (ifn? x)

Returns true if x implements IFn. Note that many data structures
(e.g. sets and maps) implement IFn
```

Added in Clojure version 1.0
Source

---

import

macro

```
Usage: (import & import-symbols-or-lists)

import-list => (package-symbol class-name-symbols*)

For each name in class-name-symbols, adds a mapping from name to the
class named by package.name to the current namespace. Use :import in the ns
macro in preference to calling this directly.
```

Added in Clojure version 1.0
Source

---

in-ns

function

```
Usage: (in-ns name)

Sets *ns* to the namespace named by the symbol, creating it if needed.
```

Added in Clojure version 1.0

---

inc

function

```
Usage: (inc x)

Returns a number one greater than num. Does not auto-promote
longs, will throw on overflow. See also: inc'
```

Added in Clojure version 1.2
Source

---

inc'

function

```
Usage: (inc' x)

Returns a number one greater than num. Supports arbitrary precision.
See also: inc
```

Added in Clojure version 1.0
Source

---

init-proxy

function

```
Usage: (init-proxy proxy mappings)

Takes a proxy instance and a map of strings (which must
correspond to methods of the proxy superclass/superinterfaces) to
fns (which must take arguments matching the corresponding method,
plus an additional (explicit) first arg corresponding to this, and
sets the proxy's fn map.  Returns the proxy.
```

Added in Clojure version 1.0
Source

---

instance?

function

```
Usage: (instance? c x)
```

```
Evaluates x and tests if it is an instance of the class
c. Returns true or false
```

Added in Clojure version 1.0
Source

---

int

function

```
Usage: (int x)

Coerce to int
```

Added in Clojure version 1.0
Source

---

int-array

function

```
Usage: (int-array size-or-seq)
       (int-array size init-val-or-seq)

Creates an array of ints
```

Added in Clojure version 1.0
Source

---

integer?

function

```
Usage: (integer? n)

Returns true if n is an integer
```

Added in Clojure version 1.0
Source

---

interleave

function

```
Usage: (interleave c1 c2)
       (interleave c1 c2 & colls)

Returns a lazy seq of the first item in each coll, then the second etc.
```

Added in Clojure version 1.0
Source

---

intern

function

```
Usage: (intern ns name)
       (intern ns name val)

Finds or creates a var named by the symbol name in the namespace
ns (which can be a symbol or a namespace), setting its root binding
to val if supplied. The namespace must exist. The var will adopt any
metadata from the name symbol.  Returns the var.
```

Added in Clojure version 1.0
Source

---

interpose

function

```
Usage: (interpose sep coll)

Returns a lazy seq of the elements of coll separated by sep
```

Added in Clojure version 1.0
Source

---

into

function

```
Usage: (into to from)
```

```
Returns a new coll consisting of to-coll with all of the items of
from-coll conjoined.
```

Added in Clojure version 1.0
Source

---

into-array

function

```
Usage: (into-array aseq)
       (into-array type aseq)

Returns an array with components set to the values in aseq. The array's
component type is type if provided, or the type of the first value in
aseq if present, or Object. All values in aseq must be compatible with
the component type. Class objects for the primitive types can be obtained
using, e.g., Integer/TYPE.
```

Added in Clojure version 1.0
Source

---

ints

function

```
Usage: (ints xs)

Casts to int[]
```

Added in Clojure version 1.0
Source

---

io!

macro

```
Usage: (io! & body)

If an io! block occurs in a transaction, throws an
IllegalStateException, else runs body in an implicit do. If the
first expression in body is a literal string, will use that as the
exception message.
```

Added in Clojure version 1.0
Source

---

isa?

function

```
Usage: (isa? child parent)
       (isa? h child parent)

Returns true if (= child parent), or child is directly or indirectly derived from
parent, either via a Java type inheritance relationship or a
relationship established via derive. h must be a hierarchy obtained
from make-hierarchy, if not supplied defaults to the global
hierarchy
```

Added in Clojure version 1.0
Source

---

iterate

function

```
Usage: (iterate f x)

Returns a lazy sequence of x, (f x), (f (f x)) etc. f must be free of side-effects
```

Added in Clojure version 1.0
Source

---

iterator-seq

function

```
Usage: (iterator-seq iter)

Returns a seq on a java.util.Iterator. Note that most collections
providing iterators implement Iterable and thus support seq directly.
```

Added in Clojure version 1.0
Source

---

juxt

function

```
Usage: (juxt f)
       (juxt f g)
       (juxt f g h)
       (juxt f g h & fs)

Takes a set of functions and returns a fn that is the juxtaposition
of those fns.  The returned fn takes a variable number of args, and
returns a vector containing the result of applying each fn to the
args (left-to-right).
((juxt a b c) x) => [(a x) (b x) (c x)]
```

Added in Clojure version 1.1
Source

---

keep

function

```
Usage: (keep f coll)

Returns a lazy sequence of the non-nil results of (f item). Note,
this means false return values will be included.  f must be free of
side-effects.
```

Added in Clojure version 1.2
Source

---

keep-indexed

function

```
Usage: (keep-indexed f coll)

Returns a lazy sequence of the non-nil results of (f index item). Note,
this means false return values will be included.  f must be free of
side-effects.
```

Added in Clojure version 1.2
Source

---

key

function

```
Usage: (key e)

Returns the key of the map entry.
```

Added in Clojure version 1.0
Source

---

keys

function

```
Usage: (keys map)

Returns a sequence of the map's keys.
```

Added in Clojure version 1.0
Source

---

keyword

function

```
Usage: (keyword name)
       (keyword ns name)

Returns a Keyword with the given namespace and name.  Do not use :
in the keyword strings, it will be added automatically.
```

Added in Clojure version 1.0
Source

---

keyword?

function

```
Usage: (keyword? x)

Return true if x is a Keyword
```

Added in Clojure version 1.0

Source

---

last

function

```
Usage: (last coll)

Return the last item in coll, in linear time
```

Added in Clojure version 1.0
Source

---

lazy-cat

macro

```
Usage: (lazy-cat & colls)

Expands to code which yields a lazy sequence of the concatenation
of the supplied colls.  Each coll expr is not evaluated until it is
needed.

(lazy-cat xs ys zs) === (concat (lazy-seq xs) (lazy-seq ys) (lazy-seq zs))
```

Added in Clojure version 1.0
Source

---

lazy-seq

macro

```
Usage: (lazy-seq & body)

Takes a body of expressions that returns an ISeq or nil, and yields
a Seqable object that will invoke the body only the first time seq
is called, and will cache the result and return it on all subsequent
seq calls. See also - realized?
```

Added in Clojure version 1.0
Source

---

let

macro

```
Usage: (let bindings & body)

binding => binding-form init-expr

Evaluates the exprs in a lexical context in which the symbols in
the binding-forms are bound to their respective init-exprs or parts
therein.
```

Added in Clojure version 1.0
Source

---

letfn

macro

```
Usage: (letfn fnspecs & body)

fnspec ==> (fname [params*] exprs) or (fname ([params*] exprs)+)

Takes a vector of function specs and a body, and generates a set of
bindings of functions to their names. All of the names are available
in all of the definitions of the functions, as well as the body.
```

Added in Clojure version 1.0
Source

---

line-seq

function

```
Usage: (line-seq rdr)

Returns the lines of text from rdr as a lazy sequence of strings.
rdr must implement java.io.BufferedReader.
```

Added in Clojure version 1.0
Source

---

list

function

```
Usage: (list & items)

Creates a new list containing the items.
```

Added in Clojure version 1.0
Source

---

list*

function

```
Usage: (list* args)
       (list* a args)
       (list* a b args)
       (list* a b c args)
       (list* a b c d & more)

Creates a new list containing the items prepended to the rest, the
last of which will be treated as a sequence.
```

Added in Clojure version 1.0
Source

---

list?

function

```
Usage: (list? x)

Returns true if x implements IPersistentList
```

Added in Clojure version 1.0
Source

---

load

function

```
Usage: (load & paths)
```

```
Loads Clojure code from resources in classpath. A path is interpreted as
classpath-relative if it begins with a slash or relative to the root
directory for the current namespace otherwise.
```

Added in Clojure version 1.0
Source

---

load-file

function

```
Usage: (load-file name)

Sequentially read and evaluate the set of forms contained in the file.
```

Added in Clojure version 1.0

---

load-reader

function

```
Usage: (load-reader rdr)

Sequentially read and evaluate the set of forms contained in the
stream/file
```

Added in Clojure version 1.0
Source

---

load-string

function

```
Usage: (load-string s)

Sequentially read and evaluate the set of forms contained in the
string
```

Added in Clojure version 1.0
Source

---

loaded-libs

function

```
Usage: (loaded-libs)

Returns a sorted set of symbols naming the currently loaded libs
```

Added in Clojure version 1.0
Source

---

locking

macro

```
Usage: (locking x & body)

Executes exprs in an implicit do, while holding the monitor of x.
Will release the monitor of x in all circumstances.
```

Added in Clojure version 1.0
Source

---

long

function

```
Usage: (long x)

Coerce to long
```

Added in Clojure version 1.0
Source

---

long-array

function

```
Usage: (long-array size-or-seq)
       (long-array size init-val-or-seq)

Creates an array of longs
```

Added in Clojure version 1.0

Source

---

longs

function

```
Usage: (longs xs)

Casts to long[]
```

Added in Clojure version 1.0
Source

---

loop

macro

```
Usage: (loop bindings & body)

Evaluates the exprs in a lexical context in which the symbols in
the binding-forms are bound to their respective init-exprs or parts
therein. Acts as a recur target.
```

Added in Clojure version 1.0
Source

---

macroexpand

function

```
Usage: (macroexpand form)

Repeatedly calls macroexpand-1 on form until it no longer
represents a macro form, then returns it.  Note neither
macroexpand-1 nor macroexpand expand macros in subforms.
```

Added in Clojure version 1.0
Source

---

macroexpand-1

function

```
Usage: (macroexpand-1 form)

If form represents a macro form, returns its expansion,
else returns form.
```

Added in Clojure version 1.0
Source

---

make-array

function

```
Usage: (make-array type len)
       (make-array type dim & more-dims)

Creates and returns an array of instances of the specified class of
the specified dimension(s).  Note that a class object is required.
Class objects can be obtained by using their imported or
fully-qualified name.  Class objects for the primitive types can be
obtained using, e.g., Integer/TYPE.
```

Added in Clojure version 1.0
Source

---

make-hierarchy

function

```
Usage: (make-hierarchy)

Creates a hierarchy object for use with derive, isa? etc.
```

Added in Clojure version 1.0
Source

---

map

function

```
Usage: (map f coll)
```

```
       (map f c1 c2)
       (map f c1 c2 c3)
       (map f c1 c2 c3 & colls)

Returns a lazy sequence consisting of the result of applying f to the
set of first items of each coll, followed by applying f to the set
of second items in each coll, until any one of the colls is
exhausted.  Any remaining items in other colls are ignored. Function
f should accept number-of-colls arguments.
```

Added in Clojure version 1.0
Source

---

map-indexed

function

```
Usage: (map-indexed f coll)

Returns a lazy sequence consisting of the result of applying f to 0
and the first item of coll, followed by applying f to 1 and the second
item in coll, etc, until coll is exhausted. Thus function f should
accept 2 arguments, index and item.
```

Added in Clojure version 1.2
Source

---

map?

function

```
Usage: (map? x)

Return true if x implements IPersistentMap
```

Added in Clojure version 1.0
Source

---

mapcat

function

```
Usage: (mapcat f & colls)
```

```
Returns the result of applying concat to the result of applying map
to f and colls.  Thus function f should return a collection.
```

Added in Clojure version 1.0
Source

---

max

function

```
Usage: (max x)
       (max x y)
       (max x y & more)

Returns the greatest of the nums.
```

Added in Clojure version 1.0
Source

---

max-key

function

```
Usage: (max-key k x)
       (max-key k x y)
       (max-key k x y & more)

Returns the x for which (k x), a number, is greatest.
```

Added in Clojure version 1.0
Source

---

memfn

macro

```
Usage: (memfn name & args)

Expands into code that creates a fn that expects to be passed an
object and any args and calls the named instance method on the
object passing the args. Use when you want to treat a Java method as
a first-class fn.
```

Added in Clojure version 1.0

Source

---

memoize

function

```
Usage: (memoize f)

Returns a memoized version of a referentially transparent function. The
memoized version of the function keeps a cache of the mapping from arguments
to results and, when calls with the same arguments are repeated often, has
higher performance at the expense of higher memory use.
```

Added in Clojure version 1.0
Source

---

merge

function

```
Usage: (merge & maps)

Returns a map that consists of the rest of the maps conj-ed onto
the first.  If a key occurs in more than one map, the mapping from
the latter (left-to-right) will be the mapping in the result.
```

Added in Clojure version 1.0
Source

---

merge-with

function

```
Usage: (merge-with f & maps)

Returns a map that consists of the rest of the maps conj-ed onto
the first.  If a key occurs in more than one map, the mapping(s)
from the latter (left-to-right) will be combined with the mapping in
the result by calling (f val-in-result val-in-latter).
```

Added in Clojure version 1.0
Source

---

meta

function

```
Usage: (meta obj)

Returns the metadata of obj, returns nil if there is no metadata.
```

Added in Clojure version 1.0
Source

---

methods

function

```
Usage: (methods multifn)

Given a multimethod, returns a map of dispatch values -> dispatch fns
```

Added in Clojure version 1.0
Source

---

min

function

```
Usage: (min x)
       (min x y)
       (min x y & more)

Returns the least of the nums.
```

Added in Clojure version 1.0
Source

---

min-key

function

```
Usage: (min-key k x)
       (min-key k x y)
       (min-key k x y & more)

Returns the x for which (k x), a number, is least.
```

Added in Clojure version 1.0
Source

---

mod

function

```
Usage: (mod num div)

Modulus of num and div. Truncates toward negative infinity.
```

Added in Clojure version 1.0
Source

---

name

function

```
Usage: (name x)

Returns the name String of a string, symbol or keyword.
```

Added in Clojure version 1.0
Source

---

namespace

function

```
Usage: (namespace x)

Returns the namespace String of a symbol or keyword, or nil if not present.
```

Added in Clojure version 1.0
Source

---

namespace-munge

function

```
Usage: (namespace-munge ns)

Convert a Clojure namespace name to a legal Java package name.
```

Added in Clojure version 1.2
Source

---

neg?

function

```
Usage: (neg? x)

Returns true if num is less than zero, else false
```

Added in Clojure version 1.0
Source

---

newline

function

```
Usage: (newline)

Writes a platform-specific newline to *out*
```

Added in Clojure version 1.0
Source

---

next

function

```
Usage: (next coll)

Returns a seq of the items after the first. Calls seq on its
argument.  If there are no more items, returns nil.
```

Added in Clojure version 1.0
Source

---

nfirst

function

```
Usage: (nfirst x)

Same as (next (first x))
```

Added in Clojure version 1.0
Source

---

nil?

function

```
Usage: (nil? x)

Returns true if x is nil, false otherwise.
```

Added in Clojure version 1.0
Source

---

nnext

function

```
Usage: (nnext x)

Same as (next (next x))
```

Added in Clojure version 1.0
Source

---

not

function

```
Usage: (not x)

Returns true if x is logical false, false otherwise.
```

Added in Clojure version 1.0
Source

---

not-any?

function

```
Usage: (not-any? pred coll)

Returns false if (pred x) is logical true for any x in coll,
else true.
```

Added in Clojure version 1.0
Source

---

not-empty

function

```
Usage: (not-empty coll)

If coll is empty, returns nil, else coll
```

Added in Clojure version 1.0
Source

---

not-every?

function

```
Usage: (not-every? pred coll)

Returns false if (pred x) is logical true for every x in
coll, else true.
```

Added in Clojure version 1.0
Source

---

not=

function

```
Usage: (not= x)
```

```
       (not= x y)
       (not= x y & more)

Same as (not (= obj1 obj2))
```

Added in Clojure version 1.0
Source

---

ns

macro

```
Usage: (ns name docstring? attr-map? references*)

Sets *ns* to the namespace named by name (unevaluated), creating it
if needed.  references can be zero or more of: (:refer-clojure ...)
(:require ...) (:use ...) (:import ...) (:load ...) (:gen-class)
with the syntax of refer-clojure/require/use/import/load/gen-class
respectively, except the arguments are unevaluated and need not be
quoted. (:gen-class ...), when supplied, defaults to :name
corresponding to the ns name, :main true, :impl-ns same as ns, and
:init-impl-ns true. All options of gen-class are
supported. The :gen-class directive is ignored when not
compiling. If :gen-class is not supplied, when compiled only an
nsname__init.class will be generated. If :refer-clojure is not used, a
default (refer 'clojure) is used.  Use of ns is preferred to
individual calls to in-ns/require/use/import:

(ns foo.bar
  (:refer-clojure :exclude [ancestors printf])
  (:require (clojure.contrib sql sql.tests))
  (:use (my.lib this that))
  (:import (java.util Date Timer Random)
           (java.sql Connection Statement)))
```

Added in Clojure version 1.0
Source

---

ns-aliases

function

```
Usage: (ns-aliases ns)

Returns a map of the aliases for the namespace.
```

Added in Clojure version 1.0
Source

---

ns-imports

function

```
Usage: (ns-imports ns)

Returns a map of the import mappings for the namespace.
```

Added in Clojure version 1.0
Source

---

ns-interns

function

```
Usage: (ns-interns ns)

Returns a map of the intern mappings for the namespace.
```

Added in Clojure version 1.0
Source

---

ns-map

function

```
Usage: (ns-map ns)

Returns a map of all the mappings for the namespace.
```

Added in Clojure version 1.0
Source

---

ns-name

function

```
Usage: (ns-name ns)

Returns the name of the namespace, a symbol.
```

Added in Clojure version 1.0
Source

---

ns-publics

function

```
Usage: (ns-publics ns)

Returns a map of the public intern mappings for the namespace.
```

Added in Clojure version 1.0
Source

---

ns-refers

function

```
Usage: (ns-refers ns)

Returns a map of the refer mappings for the namespace.
```

Added in Clojure version 1.0
Source

---

ns-resolve

function

```
Usage: (ns-resolve ns sym)
       (ns-resolve ns env sym)

Returns the var or Class to which a symbol will be resolved in the
namespace (unless found in the environement), else nil.  Note that
if the symbol is fully qualified, the var/Class to which it resolves
need not be present in the namespace.
```

Added in Clojure version 1.0
Source

---

ns-unalias

function

```
Usage: (ns-unalias ns sym)

Removes the alias for the symbol from the namespace.
```

Added in Clojure version 1.0
Source

---

ns-unmap

function

```
Usage: (ns-unmap ns sym)

Removes the mappings for the symbol from the namespace.
```

Added in Clojure version 1.0
Source

---

nth

function

```
Usage: (nth coll index)
       (nth coll index not-found)

Returns the value at the index. get returns nil if index out of
bounds, nth throws an exception unless not-found is supplied.  nth
also works for strings, Java arrays, regex Matchers and Lists, and,
in O(n) time, for sequences.
```

Added in Clojure version 1.0
Source

---

nthnext

function

```
Usage: (nthnext coll n)

Returns the nth next of coll, (seq coll) when n is 0.
```

Added in Clojure version 1.0
Source

---

nthrest

function

```
Usage: (nthrest coll n)

Returns the nth rest of coll, coll when n is 0.
```

Added in Clojure version 1.3
Source

---

num

function

```
Usage: (num x)

Coerce to Number
```

Added in Clojure version 1.0
Source

---

number?

function

```
Usage: (number? x)

Returns true if x is a Number
```

Added in Clojure version 1.0
Source

---

numerator

function

```
Usage: (numerator r)

Returns the numerator part of a Ratio.
```

Added in Clojure version 1.2
Source

---

object-array

function

```
Usage: (object-array size-or-seq)

Creates an array of objects
```

Added in Clojure version 1.2
Source

---

odd?

function

```
Usage: (odd? n)

Returns true if n is odd, throws an exception if n is not an integer
```

Added in Clojure version 1.0
Source

---

or

macro

```
Usage: (or)
       (or x)
       (or x & next)

Evaluates exprs one at a time, from left to right. If a form
returns a logical true value, or returns that value and doesn't
evaluate any of the other expressions, otherwise it returns the
value of the last expression. (or) returns nil.
```

Added in Clojure version 1.0
Source

---

parents

function

```
Usage: (parents tag)
       (parents h tag)

Returns the immediate parents of tag, either via a Java type
inheritance relationship or a relationship established via derive. h
must be a hierarchy obtained from make-hierarchy, if not supplied
defaults to the global hierarchy
```

Added in Clojure version 1.0
Source

---

partial

function

```
Usage: (partial f arg1)
       (partial f arg1 arg2)
       (partial f arg1 arg2 arg3)
       (partial f arg1 arg2 arg3 & more)

Takes a function f and fewer than the normal arguments to f, and
returns a fn that takes a variable number of additional args. When
called, the returned function calls f with args + additional args.
```

Added in Clojure version 1.0
Source

---

partition

function

```
Usage: (partition n coll)
       (partition n step coll)
       (partition n step pad coll)

Returns a lazy sequence of lists of n items each, at offsets step
apart. If step is not supplied, defaults to n, i.e. the partitions
do not overlap. If a pad collection is supplied, use its elements as
necessary to complete last partition upto n items. In case there are
not enough padding elements, return a partition with less than n items.
```

Added in Clojure version 1.0
Source

---

partition-all

function

```
Usage: (partition-all n coll)
       (partition-all n step coll)

Returns a lazy sequence of lists like partition, but may include
partitions with fewer than n items at the end.
```

Added in Clojure version 1.2
Source

---

partition-by

function

```
Usage: (partition-by f coll)

Applies f to each value in coll, splitting it each time f returns
a new value.  Returns a lazy seq of partitions.
```

Added in Clojure version 1.2
Source

---

pcalls

function

```
Usage: (pcalls & fns)

Executes the no-arg fns in parallel, returning a lazy sequence of
their values
```

Added in Clojure version 1.0
Source

---

peek

function

```
Usage: (peek coll)

For a list or queue, same as first, for a vector, same as, but much
more efficient than, last. If the collection is empty, returns nil.
```

Added in Clojure version 1.0
Source

---

persistent!

function

```
Usage: (persistent! coll)

Alpha - subject to change.
Returns a new, persistent version of the transient collection, in
constant time. The transient collection cannot be used after this
call, any such use will throw an exception.
```

Added in Clojure version 1.1
Source

---

pmap

function

```
Usage: (pmap f coll)
       (pmap f coll & colls)

Like map, except f is applied in parallel. Semi-lazy in that the
parallel computation stays ahead of the consumption, but doesn't
realize the entire result unless required. Only useful for
computationally intensive functions where the time of f dominates
the coordination overhead.
```

Added in Clojure version 1.0
Source

---

pop

function

```
Usage: (pop coll)

For a list or queue, returns a new list/queue without the first
item, for a vector, returns a new vector without the last item. If
the collection is empty, throws an exception.  Note - not the same
as next/butlast.
```

Added in Clojure version 1.0
Source

---

pop!

function

```
Usage: (pop! coll)

Alpha - subject to change.
Removes the last item from a transient vector. If
the collection is empty, throws an exception. Returns coll
```

Added in Clojure version 1.1
Source

---

pop-thread-bindings

function

```
Usage: (pop-thread-bindings)

Pop one set of bindings pushed with push-binding before. It is an error to
pop bindings without pushing before.
```

Added in Clojure version 1.1
Source

---

pos?

function

```
Usage: (pos? x)
```

```
Returns true if num is greater than zero, else false
```

Added in Clojure version 1.0
Source

---

pr

function

```
Usage: (pr)
       (pr x)
       (pr x & more)

Prints the object(s) to the output stream that is the current value
of *out*.  Prints the object(s), separated by spaces if there is
more than one.  By default, pr and prn print in a way that objects
can be read by the reader
```

Added in Clojure version 1.0
Source

---

pr-str

function

```
Usage: (pr-str & xs)

pr to a string, returning it
```

Added in Clojure version 1.0
Source

---

prefer-method

function

```
Usage: (prefer-method multifn dispatch-val-x dispatch-val-y)

Causes the multimethod to prefer matches of dispatch-val-x over dispatch-val-y
when there is a conflict
```

Added in Clojure version 1.0
Source

---

prefers

function

```
Usage: (prefers multifn)

Given a multimethod, returns a map of preferred value -> set of other values
```

Added in Clojure version 1.0
Source

---

print

function

```
Usage: (print & more)

Prints the object(s) to the output stream that is the current value
of *out*.  print and println produce output for human consumption.
```

Added in Clojure version 1.0
Source

---

print-str

function

```
Usage: (print-str & xs)

print to a string, returning it
```

Added in Clojure version 1.0
Source

---

printf

function

```
Usage: (printf fmt & args)
```

```
Prints formatted output, as per format
```

Added in Clojure version 1.0
Source

---

println

function

```
Usage: (println & more)

Same as print followed by (newline)
```

Added in Clojure version 1.0
Source

---

println-str

function

```
Usage: (println-str & xs)

println to a string, returning it
```

Added in Clojure version 1.0
Source

---

prn

function

```
Usage: (prn & more)

Same as pr followed by (newline). Observes *flush-on-newline*
```

Added in Clojure version 1.0
Source

---

prn-str

function

```
Usage: (prn-str & xs)

prn to a string, returning it
```

Added in Clojure version 1.0
Source

---

promise

function

```
Usage: (promise)

Alpha - subject to change.
Returns a promise object that can be read with deref/@, and set,
once only, with deliver. Calls to deref/@ prior to delivery will
block, unless the variant of deref with timeout is used. All
subsequent derefs will return the same delivered value without
blocking. See also - realized?.
```

Added in Clojure version 1.1
Source

---

proxy

macro

```
Usage: (proxy class-and-interfaces args & fs)

class-and-interfaces - a vector of class names

args - a (possibly empty) vector of arguments to the superclass
constructor.

f => (name [params*] body) or
(name ([params*] body) ([params+] body) ...)

Expands to code which creates a instance of a proxy class that
implements the named class/interface(s) by calling the supplied
fns. A single class, if provided, must be first. If not provided it
defaults to Object.

The interfaces names must be valid interface types. If a method fn
is not provided for a class method, the superclass methd will be
called. If a method fn is not provided for an interface method, an
```

```
UnsupportedOperationException will be thrown should it be
called. Method fns are closures and can capture the environment in
which proxy is called. Each method fn takes an additional implicit
first arg, which is bound to 'this. Note that while method fns can
be provided to override protected methods, they have no other access
to protected members, nor to super, as these capabilities cannot be
proxied.
```

Added in Clojure version 1.0
Source

---

proxy-mappings

function

```
Usage: (proxy-mappings proxy)

Takes a proxy instance and returns the proxy's fn map.
```

Added in Clojure version 1.0
Source

---

proxy-super

macro

```
Usage: (proxy-super meth & args)

Use to call a superclass method in the body of a proxy method.
Note, expansion captures 'this
```

Added in Clojure version 1.0
Source

---

push-thread-bindings

function

```
Usage: (push-thread-bindings bindings)

WARNING: This is a low-level function. Prefer high-level macros like
binding where ever possible.

Takes a map of Var/value pairs. Binds each Var to the associated value for
```

```
the current thread. Each call *MUST* be accompanied by a matching call to
pop-thread-bindings wrapped in a try-finally!

    (push-thread-bindings bindings)
    (try
      ...
      (finally
        (pop-thread-bindings)))
```

Added in Clojure version 1.1
Source

---

pvalues

macro

```
Usage: (pvalues & exprs)

Returns a lazy sequence of the values of the exprs, which are
evaluated in parallel
```

Added in Clojure version 1.0
Source

---

quot

function

```
Usage: (quot num div)

quot[ient] of dividing numerator by denominator.
```

Added in Clojure version 1.0
Source

---

rand

function

```
Usage: (rand)
       (rand n)

Returns a random floating point number between 0 (inclusive) and
n (default 1) (exclusive).
```

Added in Clojure version 1.0
Source

---

rand-int

function

```
Usage: (rand-int n)

Returns a random integer between 0 (inclusive) and n (exclusive).
```

Added in Clojure version 1.0
Source

---

rand-nth

function

```
Usage: (rand-nth coll)

Return a random element of the (sequential) collection. Will have
the same performance characteristics as nth for the given
collection.
```

Added in Clojure version 1.2
Source

---

range

function

```
Usage: (range)
       (range end)
       (range start end)
       (range start end step)

Returns a lazy seq of nums from start (inclusive) to end
(exclusive), by step, where start defaults to 0, step to 1, and end
to infinity.
```

Added in Clojure version 1.0
Source

---

ratio?

function

```
Usage: (ratio? n)

Returns true if n is a Ratio
```

Added in Clojure version 1.0
Source

---

rational?

function

```
Usage: (rational? n)

Returns true if n is a rational number
```

Added in Clojure version 1.0
Source

---

rationalize

function

```
Usage: (rationalize num)

returns the rational value of num
```

Added in Clojure version 1.0
Source

---

re-find

function

```
Usage: (re-find m)
       (re-find re s)

Returns the next regex match, if any, of string to pattern, using
java.util.regex.Matcher.find().  Uses re-groups to return the
groups.
```

Added in Clojure version 1.0
Source

---

re-groups

function

```
Usage: (re-groups m)

Returns the groups from the most recent match/find. If there are no
nested groups, returns a string of the entire match. If there are
nested groups, returns a vector of the groups, the first element
being the entire match.
```

Added in Clojure version 1.0
Source

---

re-matcher

function

```
Usage: (re-matcher re s)

Returns an instance of java.util.regex.Matcher, for use, e.g. in
re-find.
```

Added in Clojure version 1.0
Source

---

re-matches

function

```
Usage: (re-matches re s)

Returns the match, if any, of string to pattern, using
java.util.regex.Matcher.matches().  Uses re-groups to return the
groups.
```

Added in Clojure version 1.0
Source

---

re-pattern

function

```
Usage: (re-pattern s)

Returns an instance of java.util.regex.Pattern, for use, e.g. in
re-matcher.
```

Added in Clojure version 1.0
Source

---

re-seq

function

```
Usage: (re-seq re s)

Returns a lazy sequence of successive matches of pattern in string,
using java.util.regex.Matcher.find(), each such match processed with
re-groups.
```

Added in Clojure version 1.0
Source

---

read

function

```
Usage: (read)
       (read stream)
       (read stream eof-error? eof-value)
       (read stream eof-error? eof-value recursive?)

Reads the next object from stream, which must be an instance of
java.io.PushbackReader or some derivee.  stream defaults to the
current value of *in* .
```

Added in Clojure version 1.0
Source

---

read-line

function

```
Usage: (read-line)

Reads the next line from stream that is the current value of *in* .
```

Added in Clojure version 1.0
Source

---

read-string

function

```
Usage: (read-string s)

Reads one object from the string s
```

Added in Clojure version 1.0
Source

---

realized?

function

```
Usage: (realized? x)

Returns true if a value has been produced for a promise, delay, future or lazy sequence.
```

Added in Clojure version 1.3
Source

---

reduce

function

```
Usage: (reduce f coll)
       (reduce f val coll)

f should be a function of 2 arguments. If val is not supplied,
returns the result of applying f to the first 2 items in coll, then
applying f to that result and the 3rd item, etc. If coll contains no
items, f must accept no arguments as well, and reduce returns the
result of calling f with no arguments.  If coll has only 1 item, it
is returned and f is not called.  If val is supplied, returns the
result of applying f to val and the first item in coll, then
applying f to that result and the 2nd item, etc. If coll contains no
```

```
items, returns val and f is not called.
```

Added in Clojure version 1.0
Source

---

reductions

function

```
Usage: (reductions f coll)
       (reductions f init coll)

Returns a lazy seq of the intermediate values of the reduction (as
per reduce) of coll by f, starting with init.
```

Added in Clojure version 1.2
Source

---

ref

function

```
Usage: (ref x)
       (ref x & options)

Creates and returns a Ref with an initial value of x and zero or
more options (in any order):

:meta metadata-map

:validator validate-fn

:min-history (default 0)
:max-history (default 10)

If metadata-map is supplied, it will be come the metadata on the
ref. validate-fn must be nil or a side-effect-free fn of one
argument, which will be passed the intended new state on any state
change. If the new state is unacceptable, the validate-fn should
return false or throw an exception. validate-fn will be called on
transaction commit, when all refs have their final values.

Normally refs accumulate history dynamically as needed to deal with
read demands. If you know in advance you will need history you can
set :min-history to ensure it will be available when first needed (instead
of after a read fault). History is limited, and the limit can be set
with :max-history.
```

Added in Clojure version 1.0
Source

---

ref-history-count

function

```
Usage: (ref-history-count ref)

Returns the history count of a ref
```

Added in Clojure version 1.1
Source

---

ref-max-history

function

```
Usage: (ref-max-history ref)
       (ref-max-history ref n)

Gets the max-history of a ref, or sets it and returns the ref
```

Added in Clojure version 1.1
Source

---

ref-min-history

function

```
Usage: (ref-min-history ref)
       (ref-min-history ref n)

Gets the min-history of a ref, or sets it and returns the ref
```

Added in Clojure version 1.1
Source

---

ref-set

function

```
Usage: (ref-set ref val)

Must be called in a transaction. Sets the value of ref.
Returns val.
```

Added in Clojure version 1.0
Source

---

refer

function

```
Usage: (refer ns-sym & filters)

refers to all public vars of ns, subject to filters.
filters can include at most one each of:

:exclude list-of-symbols
:only list-of-symbols
:rename map-of-fromsymbol-tosymbol

For each public interned var in the namespace named by the symbol,
adds a mapping from the name of the var to the var to the current
namespace.  Throws an exception if name is already mapped to
something else in the current namespace. Filters can be used to
select a subset, via inclusion or exclusion, or to provide a mapping
to a symbol different from the var's name, in order to prevent
clashes. Use :use in the ns macro in preference to calling this directly.
```

Added in Clojure version 1.0
Source

---

refer-clojure

macro

```
Usage: (refer-clojure & filters)

Same as (refer 'clojure.core <filters>)
```

Added in Clojure version 1.0
Source

---

reify

macro

```
Usage: (reify & opts+specs)

reify is a macro with the following structure:

(reify options* specs*)

 Currently there are no options.

 Each spec consists of the protocol or interface name followed by zero
 or more method bodies:

 protocol-or-interface-or-Object
 (methodName [args+] body)*

 Methods should be supplied for all methods of the desired
 protocol(s) and interface(s). You can also define overrides for
 methods of Object. Note that the first parameter must be supplied to
 correspond to the target object ('this' in Java parlance). Thus
 methods for interfaces will take one more argument than do the
 interface declarations.  Note also that recur calls to the method
 head should *not* pass the target object, it will be supplied
 automatically and can not be substituted.

 The return type can be indicated by a type hint on the method name,
 and arg types can be indicated by a type hint on arg names. If you
 leave out all hints, reify will try to match on same name/arity
 method in the protocol(s)/interface(s) - this is preferred. If you
 supply any hints at all, no inference is done, so all hints (or
 default of Object) must be correct, for both arguments and return
 type. If a method is overloaded in a protocol/interface, multiple
 independent method definitions must be supplied.  If overloaded with
 same arity in an interface you must specify complete hints to
 disambiguate - a missing hint implies Object.

 recur works to method heads The method bodies of reify are lexical
 closures, and can refer to the surrounding local scope:

 (str (let [f "foo"]
      (reify Object
        (toString [this] f))))
 == "foo"

 (seq (let [f "foo"]
      (reify clojure.lang.Seqable
        (seq [this] (seq f)))))
 == (\f \o \o))
```

Added in Clojure version 1.2
Source

---

release-pending-sends

function

```
Usage: (release-pending-sends)

Normally, actions sent directly or indirectly during another action
are held until the action completes (changes the agent's
state). This function can be used to dispatch any pending sent
actions immediately. This has no impact on actions sent during a
transaction, which are still held until commit. If no action is
occurring, does nothing. Returns the number of actions dispatched.
```

Added in Clojure version 1.0
Source

---

rem

function

```
Usage: (rem num div)

remainder of dividing numerator by denominator.
```

Added in Clojure version 1.0
Source

---

remove

function

```
Usage: (remove pred coll)

Returns a lazy sequence of the items in coll for which
(pred item) returns false. pred must be free of side-effects.
```

Added in Clojure version 1.0
Source

---

remove-all-methods

function

```
Usage: (remove-all-methods multifn)
```

```
Removes all of the methods of multimethod.
```

Added in Clojure version 1.2
Source

---

remove-method

function

```
Usage: (remove-method multifn dispatch-val)

Removes the method of multimethod associated with dispatch-value.
```

Added in Clojure version 1.0
Source

---

remove-ns

function

```
Usage: (remove-ns sym)

Removes the namespace named by the symbol. Use with caution.
Cannot be used to remove the clojure namespace.
```

Added in Clojure version 1.0
Source

---

remove-watch

function

```
Usage: (remove-watch reference key)

Alpha - subject to change.
Removes a watch (set by add-watch) from a reference
```

Added in Clojure version 1.0
Source

---

repeat

function

```
Usage: (repeat x)
       (repeat n x)

Returns a lazy (infinite!, or length n if supplied) sequence of xs.
```

Added in Clojure version 1.0
Source

---

repeatedly

function

```
Usage: (repeatedly f)
       (repeatedly n f)

Takes a function of no args, presumably with side effects, and
returns an infinite (or length n if supplied) lazy sequence of calls
to it
```

Added in Clojure version 1.0
Source

---

replace

function

```
Usage: (replace smap coll)

Given a map of replacement pairs and a vector/collection, returns a
vector/seq with any elements = a key in smap replaced with the
corresponding val in smap
```

Added in Clojure version 1.0
Source

---

replicate

function

```
Usage: (replicate n x)
```

```
DEPRECATED: Use 'repeat' instead.
Returns a lazy seq of n xs.
```

Added in Clojure version 1.0
Deprecated since Clojure version 1.3
Source

---

require

function

```
Usage: (require & args)

Loads libs, skipping any that are already loaded. Each argument is
either a libspec that identifies a lib, a prefix list that identifies
multiple libs whose names share a common prefix, or a flag that modifies
how all the identified libs are loaded. Use :require in the ns macro
in preference to calling this directly.

Libs

A 'lib' is a named set of resources in classpath whose contents define a
library of Clojure code. Lib names are symbols and each lib is associated
with a Clojure namespace and a Java package that share its name. A lib's
name also locates its root directory within classpath using Java's
package name to classpath-relative path mapping. All resources in a lib
should be contained in the directory structure under its root directory.
All definitions a lib makes should be in its associated namespace.

'require loads a lib by loading its root resource. The root resource path
is derived from the lib name in the following manner:
Consider a lib named by the symbol 'x.y.z; it has the root directory
<classpath>/x/y/, and its root resource is <classpath>/x/y/z.clj. The root
resource should contain code to create the lib's namespace (usually by using
the ns macro) and load any additional lib resources.

Libspecs

A libspec is a lib name or a vector containing a lib name followed by
options expressed as sequential keywords and arguments.

Recognized options: :as
:as takes a symbol as its argument and makes that symbol an alias to the
  lib's namespace in the current namespace.

Prefix Lists

It's common for Clojure code to depend on several libs whose names have
the same prefix. When specifying libs, prefix lists can be used to reduce
repetition. A prefix list contains the shared prefix followed by libspecs
with the shared prefix removed from the lib names. After removing the
prefix, the names that remain must not contain any periods.
```

```
Flags

A flag is a keyword.
Recognized flags: :reload, :reload-all, :verbose
:reload forces loading of all the identified libs even if they are
  already loaded
:reload-all implies :reload and also forces loading of all libs that the
  identified libs directly or indirectly load via require or use
:verbose triggers printing information about each load, alias, and refer

Example:

The following would load the libraries clojure.zip and clojure.set
abbreviated as 's'.

(require '(clojure zip [set :as s]))
```

Added in Clojure version 1.0
Source

---

reset!

function

```
Usage: (reset! atom newval)

Sets the value of atom to newval without regard for the
current value. Returns newval.
```

Added in Clojure version 1.0
Source

---

reset-meta!

function

```
Usage: (reset-meta! iref metadata-map)

Atomically resets the metadata for a namespace/var/ref/agent/atom
```

Added in Clojure version 1.0
Source

---

resolve

function

```
Usage: (resolve sym)
       (resolve env sym)

same as (ns-resolve *ns* symbol) or (ns-resolve *ns* &env symbol)
```

Added in Clojure version 1.0
Source

---

rest

function

```
Usage: (rest coll)

Returns a possibly empty seq of the items after the first. Calls seq on its
argument.
```

Added in Clojure version 1.0
Source

---

restart-agent

function

```
Usage: (restart-agent a new-state & options)

When an agent is failed, changes the agent state to new-state and
then un-fails the agent so that sends are allowed again.  If
a :clear-actions true option is given, any actions queued on the
agent that were being held while it was failed will be discarded,
otherwise those held actions will proceed.  The new-state must pass
the validator if any, or restart will throw an exception and the
agent will remain failed with its old state and error.  Watchers, if
any, will NOT be notified of the new state.  Throws an exception if
the agent is not failed.
```

Added in Clojure version 1.2
Source

---

resultset-seq

function

```
Usage: (resultset-seq rs)

Creates and returns a lazy sequence of structmaps corresponding to
the rows in the java.sql.ResultSet rs
```

Added in Clojure version 1.0
Source

---

reverse

function

```
Usage: (reverse coll)

Returns a seq of the items in coll in reverse order. Not lazy.
```

Added in Clojure version 1.0
Source

---

reversible?

function

```
Usage: (reversible? coll)

Returns true if coll implements Reversible
```

Added in Clojure version 1.0
Source

---

rseq

function

```
Usage: (rseq rev)

Returns, in constant time, a seq of the items in rev (which
can be a vector or sorted-map), in reverse order. If rev is empty returns nil
```

Added in Clojure version 1.0

Source

---

rsubseq

function

```
Usage: (rsubseq sc test key)
       (rsubseq sc start-test start-key end-test end-key)

sc must be a sorted collection, test(s) one of <, <=, > or
>=. Returns a reverse seq of those entries with keys ek for
which (test (.. sc comparator (compare ek key)) 0) is true
```

Added in Clojure version 1.0
Source

---

satisfies?

function

```
Usage: (satisfies? protocol x)

Returns true if x satisfies the protocol
```

Added in Clojure version 1.2
Source

---

second

function

```
Usage: (second x)

Same as (first (next x))
```

Added in Clojure version 1.0
Source

---

select-keys

function

```
Usage: (select-keys map keyseq)

Returns a map containing only those entries in map whose key is in keys
```

Added in Clojure version 1.0
Source

---

send

function

```
Usage: (send a f & args)

Dispatch an action to an agent. Returns the agent immediately.
Subsequently, in a thread from a thread pool, the state of the agent
will be set to the value of:

(apply action-fn state-of-agent args)
```

Added in Clojure version 1.0
Source

---

send-off

function

```
Usage: (send-off a f & args)

Dispatch a potentially blocking action to an agent. Returns the
agent immediately. Subsequently, in a separate thread, the state of
the agent will be set to the value of:

(apply action-fn state-of-agent args)
```

Added in Clojure version 1.0
Source

---

seq

function

```
Usage: (seq coll)

Returns a seq on the collection. If the collection is
empty, returns nil.  (seq nil) returns nil. seq also works on
Strings, native Java arrays (of reference types) and any objects
that implement Iterable.
```

Added in Clojure version 1.0
Source

---

seq?

function

```
Usage: (seq? x)

Return true if x implements ISeq
```

Added in Clojure version 1.0
Source

---

seque

function

```
Usage: (seque s)
       (seque n-or-q s)

Creates a queued seq on another (presumably lazy) seq s. The queued
seq will produce a concrete seq in the background, and can get up to
n items ahead of the consumer. n-or-q can be an integer n buffer
size, or an instance of java.util.concurrent BlockingQueue. Note
that reading from a seque can block if the reader gets ahead of the
producer.
```

Added in Clojure version 1.0
Source

---

sequence

function

```
Usage: (sequence coll)
```

```
Coerces coll to a (possibly empty) sequence, if it is not already
one. Will not force a lazy seq. (sequence nil) yields ()
```

Added in Clojure version 1.0
Source

---

sequential?

function

```
Usage: (sequential? coll)

Returns true if coll implements Sequential
```

Added in Clojure version 1.0
Source

---

set

function

```
Usage: (set coll)

Returns a set of the distinct elements of coll.
```

Added in Clojure version 1.0
Source

---

set-error-handler!

function

```
Usage: (set-error-handler! a handler-fn)

Sets the error-handler of agent a to handler-fn.  If an action
being run by the agent throws an exception or doesn't pass the
validator fn, handler-fn will be called with two arguments: the
agent and the exception.
```

Added in Clojure version 1.2
Source

---

set-error-mode!

function

```
Usage: (set-error-mode! a mode-keyword)

Sets the error-mode of agent a to mode-keyword, which must be
either :fail or :continue.  If an action being run by the agent
throws an exception or doesn't pass the validator fn, an
error-handler may be called (see set-error-handler!), after which,
if the mode is :continue, the agent will continue as if neither the
action that caused the error nor the error itself ever happened.

If the mode is :fail, the agent will become failed and will stop
accepting new 'send' and 'send-off' actions, and any previously
queued actions will be held until a 'restart-agent'.  Deref will
still work, returning the state of the agent before the error.
```

Added in Clojure version 1.2
Source

---

set-validator!

function

```
Usage: (set-validator! iref validator-fn)

Sets the validator-fn for a var/ref/agent/atom. validator-fn must be nil or a
side-effect-free fn of one argument, which will be passed the intended
new state on any state change. If the new state is unacceptable, the
validator-fn should return false or throw an exception. If the current state (root
value if var) is not acceptable to the new validator, an exception
will be thrown and the validator will not be changed.
```

Added in Clojure version 1.0
Source

---

set?

function

```
Usage: (set? x)

Returns true if x implements IPersistentSet
```

Added in Clojure version 1.0
Source

---

short

function

```
Usage: (short x)

Coerce to short
```

Added in Clojure version 1.0
Source

---

short-array

function

```
Usage: (short-array size-or-seq)
       (short-array size init-val-or-seq)

Creates an array of shorts
```

Added in Clojure version 1.1
Source

---

shorts

function

```
Usage: (shorts xs)

Casts to shorts[]
```

Added in Clojure version 1.1
Source

---

shuffle

function

```
Usage: (shuffle coll)
```

```
Return a random permutation of coll
```

Added in Clojure version 1.2
Source

---

shutdown-agents

function

```
Usage: (shutdown-agents)

Initiates a shutdown of the thread pools that back the agent
system. Running actions will complete, but no new actions will be
accepted
```

Added in Clojure version 1.0
Source

---

slurp

function

```
Usage: (slurp f & opts)

Opens a reader on f and reads all its contents, returning a string.
See clojure.java.io/reader for a complete list of supported arguments.
```

Added in Clojure version 1.0
Source

---

some

function

```
Usage: (some pred coll)

Returns the first logical true value of (pred x) for any x in coll,
else nil.  One common idiom is to use a set as pred, for example
this will return :fred if :fred is in the sequence, otherwise nil:
(some #{:fred} coll)
```

Added in Clojure version 1.0
Source

---

some-fn

function

```
Usage: (some-fn p)
       (some-fn p1 p2)
       (some-fn p1 p2 p3)
       (some-fn p1 p2 p3 & ps)

Takes a set of predicates and returns a function f that returns the first logical true
value
returned by one of its composing predicates against any of its arguments, else it returns
logical false. Note that f is short-circuiting in that it will stop execution on the first
argument that triggers a logical true result against the original predicates.
```

Added in Clojure version 1.3
Source

---

sort

function

```
Usage: (sort coll)
       (sort comp coll)

Returns a sorted sequence of the items in coll. If no comparator is
supplied, uses compare. comparator must
implement java.util.Comparator.
```

Added in Clojure version 1.0
Source

---

sort-by

function

```
Usage: (sort-by keyfn coll)
       (sort-by keyfn comp coll)

Returns a sorted sequence of the items in coll, where the sort
order is determined by comparing (keyfn item).  If no comparator is
supplied, uses compare. comparator must
implement java.util.Comparator.
```

Added in Clojure version 1.0
Source

---

sorted-map

function

```
Usage: (sorted-map & keyvals)

keyval => key val
Returns a new sorted map with supplied mappings.
```

Added in Clojure version 1.0
Source

---

sorted-map-by

function

```
Usage: (sorted-map-by comparator & keyvals)

keyval => key val
Returns a new sorted map with supplied mappings, using the supplied comparator.
```

Added in Clojure version 1.0
Source

---

sorted-set

function

```
Usage: (sorted-set & keys)

Returns a new sorted set with supplied keys.
```

Added in Clojure version 1.0
Source

---

sorted-set-by

function

```
Usage: (sorted-set-by comparator & keys)

Returns a new sorted set with supplied keys, using the supplied comparator.
```

Added in Clojure version 1.1
Source

---

sorted?

function

```
Usage: (sorted? coll)

Returns true if coll implements Sorted
```

Added in Clojure version 1.0
Source

---

special-symbol?

function

```
Usage: (special-symbol? s)

Returns true if s names a special form
```

Added in Clojure version 1.0
Source

---

spit

function

```
Usage: (spit f content & options)

Opposite of slurp.  Opens f with writer, writes content, then
closes f. Options passed to clojure.java.io/writer.
```

Added in Clojure version 1.2
Source

---

split-at

function

```
Usage: (split-at n coll)

Returns a vector of [(take n coll) (drop n coll)]
```

Added in Clojure version 1.0
Source

---

split-with

function

```
Usage: (split-with pred coll)

Returns a vector of [(take-while pred coll) (drop-while pred coll)]
```

Added in Clojure version 1.0
Source

---

str

function

```
Usage: (str)
       (str x)
       (str x & ys)

With no args, returns the empty string. With one arg x, returns
x.toString().  (str nil) returns the empty string. With more than
one arg, returns the concatenation of the str values of the args.
```

Added in Clojure version 1.0
Source

---

string?

function

```
Usage: (string? x)

Return true if x is a String
```

Added in Clojure version 1.0
Source

---

struct

function

```
Usage: (struct s & vals)

Returns a new structmap instance with the keys of the
structure-basis. vals must be supplied for basis keys in order -
where values are not supplied they will default to nil.
```

Added in Clojure version 1.0
Source

---

struct-map

function

```
Usage: (struct-map s & inits)

Returns a new structmap instance with the keys of the
structure-basis. keyvals may contain all, some or none of the basis
keys - where values are not supplied they will default to nil.
keyvals can also contain keys not in the basis.
```

Added in Clojure version 1.0
Source

---

subs

function

```
Usage: (subs s start)
       (subs s start end)

Returns the substring of s beginning at start inclusive, and ending
at end (defaults to length of string), exclusive.
```

Added in Clojure version 1.0
Source

---

subseq

function

```
Usage: (subseq sc test key)
       (subseq sc start-test start-key end-test end-key)

sc must be a sorted collection, test(s) one of <, <=, > or
>=. Returns a seq of those entries with keys ek for
which (test (.. sc comparator (compare ek key)) 0) is true
```

Added in Clojure version 1.0
Source

---

subvec

function

```
Usage: (subvec v start)
       (subvec v start end)

Returns a persistent vector of the items in vector from
start (inclusive) to end (exclusive).  If end is not supplied,
defaults to (count vector). This operation is O(1) and very fast, as
the resulting vector shares structure with the original and no
trimming is done.
```

Added in Clojure version 1.0
Source

---

supers

function

```
Usage: (supers class)

Returns the immediate and indirect superclasses and interfaces of c, if any
```

Added in Clojure version 1.0
Source

---

swap!

function

```
Usage: (swap! atom f)
       (swap! atom f x)
       (swap! atom f x y)
       (swap! atom f x y & args)

Atomically swaps the value of atom to be:
(apply f current-value-of-atom args). Note that f may be called
multiple times, and thus should be free of side effects.  Returns
the value that was swapped in.
```

Added in Clojure version 1.0
Source

---

symbol

function

```
Usage: (symbol name)
       (symbol ns name)

Returns a Symbol with the given namespace and name.
```

Added in Clojure version 1.0
Source

---

symbol?

function

```
Usage: (symbol? x)

Return true if x is a Symbol
```

Added in Clojure version 1.0
Source

---

sync

macro

```
Usage: (sync flags-ignored-for-now & body)

transaction-flags => TBD, pass nil for now

Runs the exprs (in an implicit do) in a transaction that encompasses
exprs and any nested calls.  Starts a transaction if none is already
running on this thread. Any uncaught exception will abort the
transaction and flow out of sync. The exprs may be run more than
once, but any effects on Refs will be atomic.
```

Added in Clojure version 1.0
Source

---

take

function

```
Usage: (take n coll)

Returns a lazy sequence of the first n items in coll, or all items if
there are fewer than n.
```

Added in Clojure version 1.0
Source

---

take-last

function

```
Usage: (take-last n coll)

Returns a seq of the last n items in coll.  Depending on the type
of coll may be no better than linear time.  For vectors, see also subvec.
```

Added in Clojure version 1.1
Source

---

take-nth

function

```
Usage: (take-nth n coll)

Returns a lazy seq of every nth item in coll.
```

Added in Clojure version 1.0
Source

---

take-while

function

```
Usage: (take-while pred coll)

Returns a lazy sequence of successive items from coll while
(pred item) returns true. pred must be free of side-effects.
```

Added in Clojure version 1.0
Source

---

test

function

```
Usage: (test v)

test [v] finds fn at key :test in var metadata and calls it,
presuming failure will throw exception
```

Added in Clojure version 1.0
Source

---

the-ns

function

```
Usage: (the-ns x)

If passed a namespace, returns it. Else, when passed a symbol,
returns the namespace named by it, throwing an exception if not
found.
```

Added in Clojure version 1.0
Source

---

thread-bound?

function

```
Usage: (thread-bound? & vars)

Returns true if all of the vars provided as arguments have thread-local bindings.
Implies that set!'ing the provided vars will succeed.  Returns true if no vars are
provided.
```

Added in Clojure version 1.2
Source

---

time

macro

```
Usage: (time expr)

Evaluates expr and prints the time it took.  Returns the value of
expr.
```

Added in Clojure version 1.0
Source

---

to-array

function

```
Usage: (to-array coll)

Returns an array of Objects containing the contents of coll, which
can be any Collection.  Maps to java.util.Collection.toArray().
```

Added in Clojure version 1.0
Source

---

to-array-2d

function

```
Usage: (to-array-2d coll)

Returns a (potentially-ragged) 2-dimensional array of Objects
containing the contents of coll, which can be any Collection of any
Collection.
```

Added in Clojure version 1.0
Source

---

trampoline

function

```
Usage: (trampoline f)
       (trampoline f & args)

trampoline can be used to convert algorithms requiring mutual
recursion without stack consumption. Calls f with supplied args, if
any. If f returns a fn, calls that fn with no arguments, and
continues to repeat, until the return value is not a fn, then
returns that non-fn value. Note that if you want to return a fn as a
final value, you must wrap it in some data structure and unpack it
after trampoline returns.
```

Added in Clojure version 1.0
Source

---

transient

function

```
Usage: (transient coll)

Alpha - subject to change.
Returns a new, transient version of the collection, in constant time.
```

Added in Clojure version 1.1
Source

---

tree-seq

function

```
Usage: (tree-seq branch? children root)
```

```
Returns a lazy sequence of the nodes in a tree, via a depth-first walk.
 branch? must be a fn of one arg that returns true if passed a node
 that can have children (but may not).  children must be a fn of one
 arg that returns a sequence of the children. Will only be called on
 nodes for which branch? returns true. Root is the root node of the
tree.
```

Added in Clojure version 1.0
Source

---

true?

function

```
Usage: (true? x)

Returns true if x is the value true, false otherwise.
```

Added in Clojure version 1.0
Source

---

type

function

```
Usage: (type x)

Returns the :type metadata of x, or its Class if none
```

Added in Clojure version 1.0
Source

---

unchecked-add

function

```
Usage: (unchecked-add x y)

Returns the sum of x and y, both long.
Note - uses a primitive operator subject to overflow.
```

Added in Clojure version 1.0
Source

---

unchecked-add-int

function

```
Usage: (unchecked-add-int x y)

Returns the sum of x and y, both int.
Note - uses a primitive operator subject to overflow.
```

Added in Clojure version 1.0
Source

---

unchecked-byte

function

```
Usage: (unchecked-byte x)

Coerce to byte. Subject to rounding or truncation.
```

Added in Clojure version 1.3
Source

---

unchecked-char

function

```
Usage: (unchecked-char x)

Coerce to char. Subject to rounding or truncation.
```

Added in Clojure version 1.3
Source

---

unchecked-dec

function

```
Usage: (unchecked-dec x)
```

```
Returns a number one less than x, a long.
Note - uses a primitive operator subject to overflow.
```

Added in Clojure version 1.0
Source

---

unchecked-dec-int

function

```
Usage: (unchecked-dec-int x)

Returns a number one less than x, an int.
Note - uses a primitive operator subject to overflow.
```

Added in Clojure version 1.0
Source

---

unchecked-divide-int

function

```
Usage: (unchecked-divide-int x y)

Returns the division of x by y, both int.
Note - uses a primitive operator subject to truncation.
```

Added in Clojure version 1.0
Source

---

unchecked-double

function

```
Usage: (unchecked-double x)

Coerce to double. Subject to rounding.
```

Added in Clojure version 1.3
Source

---

unchecked-float

function

```
Usage: (unchecked-float x)

Coerce to float. Subject to rounding.
```

Added in Clojure version 1.3
Source

---

unchecked-inc

function

```
Usage: (unchecked-inc x)

Returns a number one greater than x, a long.
Note - uses a primitive operator subject to overflow.
```

Added in Clojure version 1.0
Source

---

unchecked-inc-int

function

```
Usage: (unchecked-inc-int x)

Returns a number one greater than x, an int.
Note - uses a primitive operator subject to overflow.
```

Added in Clojure version 1.0
Source

---

unchecked-int

function

```
Usage: (unchecked-int x)

Coerce to int. Subject to rounding or truncation.
```

Added in Clojure version 1.3

Source

---

unchecked-long

function

```
Usage: (unchecked-long x)

Coerce to long. Subject to rounding or truncation.
```

Added in Clojure version 1.3
Source

---

unchecked-multiply

function

```
Usage: (unchecked-multiply x y)

Returns the product of x and y, both long.
Note - uses a primitive operator subject to overflow.
```

Added in Clojure version 1.0
Source

---

unchecked-multiply-int

function

```
Usage: (unchecked-multiply-int x y)

Returns the product of x and y, both int.
Note - uses a primitive operator subject to overflow.
```

Added in Clojure version 1.0
Source

---

unchecked-negate

function

```
Usage: (unchecked-negate x)

Returns the negation of x, a long.
Note - uses a primitive operator subject to overflow.
```

Added in Clojure version 1.0
Source

---

unchecked-negate-int

function

```
Usage: (unchecked-negate-int x)

Returns the negation of x, an int.
Note - uses a primitive operator subject to overflow.
```

Added in Clojure version 1.0
Source

---

unchecked-remainder-int

function

```
Usage: (unchecked-remainder-int x y)

Returns the remainder of division of x by y, both int.
Note - uses a primitive operator subject to truncation.
```

Added in Clojure version 1.0
Source

---

unchecked-short

function

```
Usage: (unchecked-short x)

Coerce to short. Subject to rounding or truncation.
```

Added in Clojure version 1.3
Source

---

unchecked-subtract

function

```
Usage: (unchecked-subtract x y)

Returns the difference of x and y, both long.
Note - uses a primitive operator subject to overflow.
```

Added in Clojure version 1.0
Source

---

unchecked-subtract-int

function

```
Usage: (unchecked-subtract-int x y)

Returns the difference of x and y, both int.
Note - uses a primitive operator subject to overflow.
```

Added in Clojure version 1.0
Source

---

underive

function

```
Usage: (underive tag parent)
       (underive h tag parent)

Removes a parent/child relationship between parent and
tag. h must be a hierarchy obtained from make-hierarchy, if not
supplied defaults to, and modifies, the global hierarchy.
```

Added in Clojure version 1.0
Source

---

update-in

function

```
Usage: (update-in m [k & ks] f & args)

'Updates' a value in a nested associative structure, where ks is a
sequence of keys and f is a function that will take the old value
and any supplied args and return the new value, and returns a new
nested structure.  If any levels do not exist, hash-maps will be
created.
```

Added in Clojure version 1.0
Source

---

update-proxy

function

```
Usage: (update-proxy proxy mappings)

Takes a proxy instance and a map of strings (which must
correspond to methods of the proxy superclass/superinterfaces) to
fns (which must take arguments matching the corresponding method,
plus an additional (explicit) first arg corresponding to this, and
updates (via assoc) the proxy's fn map. nil can be passed instead of
a fn, in which case the corresponding method will revert to the
default behavior. Note that this function can be used to update the
behavior of an existing instance without changing its identity.
Returns the proxy.
```

Added in Clojure version 1.0
Source

---

use

function

```
Usage: (use & args)

Like 'require, but also refers to each lib's namespace using
clojure.core/refer. Use :use in the ns macro in preference to calling
this directly.

'use accepts additional options in libspecs: :exclude, :only, :rename.
The arguments and semantics for :exclude, :only, and :rename are the same
as those documented for clojure.core/refer.
```

Added in Clojure version 1.0
Source

---

val

function

```
Usage: (val e)

Returns the value in the map entry.
```

Added in Clojure version 1.0
Source

---

vals

function

```
Usage: (vals map)

Returns a sequence of the map's values.
```

Added in Clojure version 1.0
Source

---

var-get

function

```
Usage: (var-get x)

Gets the value in the var object
```

Added in Clojure version 1.0
Source

---

var-set

function

```
Usage: (var-set x val)

Sets the value in the var object to val. The var must be
thread-locally bound.
```

Added in Clojure version 1.0
Source

---

var?

function

```
Usage: (var? v)

Returns true if v is of type clojure.lang.Var
```

Added in Clojure version 1.0
Source

---

vary-meta

function

```
Usage: (vary-meta obj f & args)

Returns an object of the same type and value as obj, with
(apply f (meta obj) args) as its metadata.
```

Added in Clojure version 1.0
Source

---

vec

function

```
Usage: (vec coll)

Creates a new vector containing the contents of coll.
```

Added in Clojure version 1.0
Source

---

vector

function

```
Usage: (vector)
       (vector a)
```

```
       (vector a b)
       (vector a b c)
       (vector a b c d)
       (vector a b c d & args)

Creates a new vector containing the args.
```

Added in Clojure version 1.0
Source

---

vector-of

function

```
Usage: (vector-of t)
       (vector-of t & elements)

Creates a new vector of a single primitive type t, where t is one
of :int :long :float :double :byte :short :char or :boolean. The
resulting vector complies with the interface of vectors in general,
but stores the values unboxed internally.

Optionally takes one or more elements to populate the vector.
```

Added in Clojure version 1.2
Source

---

vector?

function

```
Usage: (vector? x)

Return true if x implements IPersistentVector
```

Added in Clojure version 1.0
Source

---

when

macro

```
Usage: (when test & body)
```

```
Evaluates test. If logical true, evaluates body in an implicit do.
```

Added in Clojure version 1.0
Source

---

when-first

macro

```
Usage: (when-first bindings & body)

bindings => x xs

Same as (when (seq xs) (let [x (first xs)] body))
```

Added in Clojure version 1.0
Source

---

when-let

macro

```
Usage: (when-let bindings & body)

bindings => binding-form test

When test is true, evaluates body with binding-form bound to the value of test
```

Added in Clojure version 1.0
Source

---

when-not

macro

```
Usage: (when-not test & body)

Evaluates test. If logical false, evaluates body in an implicit do.
```

Added in Clojure version 1.0
Source

---

while

macro

```
Usage: (while test & body)

Repeatedly executes body while test expression is true. Presumes
some side-effect will cause test to become false/nil. Returns nil
```

Added in Clojure version 1.0
Source

---

with-bindings

macro

```
Usage: (with-bindings binding-map & body)

Takes a map of Var/value pairs. Installs for the given Vars the associated
values as thread-local bindings. The executes body. Pops the installed
bindings after body was evaluated. Returns the value of body.
```

Added in Clojure version 1.1
Source

---

with-bindings*

function

```
Usage: (with-bindings* binding-map f & args)

Takes a map of Var/value pairs. Installs for the given Vars the associated
values as thread-local bindings. Then calls f with the supplied arguments.
Pops the installed bindings after f returned. Returns whatever f returns.
```

Added in Clojure version 1.1
Source

---

with-in-str

macro

```
Usage: (with-in-str s & body)
```

```
Evaluates body in a context in which *in* is bound to a fresh
StringReader initialized with the string s.
```

Added in Clojure version 1.0
Source

---

with-local-vars

macro

```
Usage: (with-local-vars name-vals-vec & body)

varbinding=> symbol init-expr

Executes the exprs in a context in which the symbols are bound to
vars with per-thread bindings to the init-exprs.  The symbols refer
to the var objects themselves, and must be accessed with var-get and
var-set
```

Added in Clojure version 1.0
Source

---

with-meta

function

```
Usage: (with-meta obj m)

Returns an object of the same type and value as obj, with
map m as its metadata.
```

Added in Clojure version 1.0
Source

---

with-open

macro

```
Usage: (with-open bindings & body)

bindings => [name init ...]

Evaluates body in a try expression with names bound to the values
of the inits, and a finally clause that calls (.close name) on each
```

```
name in reverse order.
```

Added in Clojure version 1.0
Source

---

with-out-str

macro

```
Usage: (with-out-str & body)

Evaluates exprs in a context in which *out* is bound to a fresh
StringWriter.  Returns the string created by any nested printing
calls.
```

Added in Clojure version 1.0
Source

---

with-precision

macro

```
Usage: (with-precision precision & exprs)

Sets the precision and rounding mode to be used for BigDecimal operations.

Usage: (with-precision 10 (/ 1M 3))
or:    (with-precision 10 :rounding HALF_DOWN (/ 1M 3))

The rounding mode is one of CEILING, FLOOR, HALF_UP, HALF_DOWN,
HALF_EVEN, UP, DOWN and UNNECESSARY; it defaults to HALF_UP.
```

Added in Clojure version 1.0
Source

---

with-redefs

macro

```
Usage: (with-redefs bindings & body)

binding => var-symbol temp-value-expr

Temporarily redefines Vars while executing the body.  The
```

```
temp-value-exprs will be evaluated and each resulting value will
replace in parallel the root value of its Var.  After the body is
executed, the root values of all the Vars will be set back to their
old values.  These temporary changes will be visible in all threads.
Useful for mocking out functions during testing.
```

Added in Clojure version 1.3
Source

---

with-redefs-fn

function

```
Usage: (with-redefs-fn binding-map func)

Temporarily redefines Vars during a call to func.  Each val of
binding-map will replace the root value of its key which must be
a Var.  After func is called with no args, the root values of all
the Vars will be set back to their old values.  These temporary
changes will be visible in all threads.  Useful for mocking out
functions during testing.
```

Added in Clojure version 1.3
Source

---

xml-seq

function

```
Usage: (xml-seq root)

A tree seq on the xml elements as per xml/parse
```

Added in Clojure version 1.0
Source

---

zero?

function

```
Usage: (zero? x)

Returns true if num is zero, else false
```

Added in Clojure version 1.0
Source

---

zipmap

function

```
Usage: (zipmap keys vals)

Returns a map with the keys mapped to the corresponding vals.
```

Added in Clojure version 1.0
Source

clojure.core.protocols

InternalReduce

var

```
Protocol for concrete seq types that can reduce themselves
faster than first/next recursion. Called by clojure.core/reduce.
```

Source