

Primitives

Numbers

Literals	Nil: nil Long: 1500 Double: 3.569 Boolean: true, false BigDecimal: 6.897M String: "abcde"
Arithmetic	+ - * / mod inc dec min max abs
Compare	== != <

Collections

Collections

Generic	count empty? not-empty? empty-to-nil into conj remove repeat range group-by
Tests	coll? list? vector? set? map? seq? hash-map? ordered-map? sorted-map? bytebuf?

Lists

Create	() list
Access	first second nth last peek rest
Modify	cons conj rest pop into concat flatten reduce reverse sort sort-by take take-while drop drop-while

Vectors

Use	count empty-to-nil str/index-of str/last-index-of str/replace-first str/replace-last str/replace-all str/lower-case str/upper-case str/join str/subs str/split str/truncate
Regex	match match-not
Trim	str/trim str/trim-to-nil
Test	string? empty? str/starts-with? str/ends-with? str/contains?
Other	
Keywords	keyword? keyword literals: <code>:a</code> <code>:xyz</code>
Symbols	symbol? symbol
Boolean	boolean? boolean true? false?

Functions	
Create	fn
Call	apply memoize
Test	fn?
Exception	throw
Misc	class eval
Other	

Create	<code>[]</code> vector
Access	first second nth last peek rest subvec
Modify	cons conj rest pop into concat flatten reduce reverse sort sort-by take take-while drop drop-while
Test	contains?

Sets	
Create	set
Test	contains?

Maps	
Create	<code>{}</code> hash-map ordered-map sorted-map
Access	find get keys vals key val
Modify	cons conj assoc into concat flatten reduce-kv
Test	contains?

Other Types	
-------------	--

[version](#) [uuid](#)
[time-ms](#) [time-ns](#)
[coalesce](#)

Meta [meta](#) [with-meta](#)
[vary-meta](#)

ByteBuffer

Misc [count](#) [empty?](#) [not-empty?](#) [bytebuf](#)
[bytebuf?](#) [subbytebuf](#)

Macros

Create [defmacro](#)

Branch [and](#) [or](#) [not](#) [when](#)
[when-not](#) [if-let](#)

Loop [list-comp](#) [dotimes](#)
[while](#)

Call [doto](#) [->](#) [->>](#)

Test [macro?](#) [cond](#)

Assert [assert](#)

Util [comment](#) [gensym](#)
[time](#)

Atoms

Create [atom](#)

Test [atom?](#)

Access [deref](#) [reset!](#) [swap!](#)
[compare-and-set!](#)

Special Forms

Forms [def](#) [if](#) [do](#) [let](#) [fn](#)
[loop](#) [defmacro](#)
[recur](#) [try](#)

IO

to [prn](#) [println](#)

to-str [pr-str](#)

from [readline](#) [read-string](#)

file-io [slurp](#) [spit](#) [io/file](#) [io](#)
[/file?](#) [io/exists-file?](#) [io](#)
[/exists-dir?](#) [io/list-files](#)
[io/delete-file](#) [io/copy-file](#)
[io/tmp-dir](#) [io/user-dir](#)

load [load-file](#) [load-string](#)

Java Interoperability

General [.](#)
Constructor: ([.](#) classname :
new args)
Method call: ([.](#) object method
args)

Embedding in Java

Eval

```
import org.venice.Venice;

public class Example {
    public static void main(String[] args) {
        Venice venice = new Venice();

        Long val = (Long)venice.eval("(+ 1 2)");
    }
}
```

Passing parameters

```
import org.venice.Venice;
import org.venice.Parameters;

public class Example {
    public static void main(String[] args) {
        Venice venice = new Venice();

        Long val = (Long)venice.eval(
            "(+ x y 3)",
            Parameters.of("x", 6, "y", 3L));
    }
}
```

Precompiled

```
import org.venice.Venice;
import org.venice.PreCompiled;

public class Example {
    public static void main(String[] args) {
        Venice venice = new Venice();

        PreCompiled precompiled = venice.precompile("(+ 1 x)");

        for(int ii=0; ii<100; ii++) {
            venice.eval(precompiled, Parameters.of("x", ii));
        }
    }
}
```

Java Interop

```
import java.time.ZonedDateTime;
import org.venice.Venice;

public class Example {
    public static void main(String[] args) {
        Venice venice = new Venice();

        Long val = (Long)venice.eval("( . :java.lang.Math :min 20 30)");

        ZonedDateTime ts = (ZonedDateTime)venice.eval(
            "(. ( . :java.time.ZonedDateTime :now) :plusDays 5)");
    }
}
```

Sandbox

```
import org.venice.Venice;
import import org.venice.javainterop.*;

public class Example {
```

```
public static void main(String[] args) {
    Venice venice = new Venice();

    JavaInterceptor interceptor =
        new JavaSandboxInterceptor(
            WhiteList.create(
                "java.lang.Math:min",
                "java.lang.Math:max",
                "java.time.ZonedDateTime:*",
                "java.util.ArrayList:new"));

    venice.eval("( . :java.lang.Math :min 20 30)"); // => OK
    venice.eval("( ( . :java.time.ZonedDateTime :now) :plusDays 5)"); // => OK
    venice.eval("( . :java.util.ArrayList :new)"); // => OK
    venice.eval("( . :java.lang.System :exit 0)"); // => Sandbox SecurityException
}
}
```

Function details

!=

(!= x y)

Returns true if both operands do not have the equivalent type

(*) (* x) (* x y) (* x y & more)

Returns the product of numbers. (*) returns 1

+

(+) (+ x) (+ x y) (+ x y & more)

Returns the sum of the numbers. (+) returns 0.

(+)
=> 0

(+ 1)
=> 1

(+ 1 2)
=> 3

(+ 1 2 3 4)
=> 10

-

(- x) (- x y) (- x y & more)

If one number is supplied, returns the negation, else subtracts the numbers from x and returns the result.

->

(-> x & forms)

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.

->>

(->> x & forms)

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.

▪

(. classname :new args) (. object method args) (. classname :class) (. object :class)

Java interop. Calls a constructor or an object method. The function is sandboxed

```
(. :java.lang.Math :PI)
=> 3.141592653589793
```

```
(. :java.lang.Long :new 10)
=> 10
```

```
(. (. :java.lang.Long :new 10) :toString)
=> 10
```

```
(. :java.lang.Math :min 10 20)
=> 10
```

```
(. :java.lang.Math :class)
=> class java.lang.Math
```

```
(. "java.lang.Math" :class)
=> class java.lang.Math
```

```
(. (. :java.io.File :new "/temp") :class)
=> class java.io.File
```

/

(/ x) (/ x y) (/ x y & more)

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

<

(< x y)

Returns true if x is smaller than y

<=

(<= x y)

Returns true if x is smaller or equal to y

==

(== x y)

Returns true if both operands have the equivalent type

>

(> x y)

Returns true if x is greater than y

>=

(>= x y)

Returns true if x is greater or equal to y

abs

(abs x)

Returns the absolute value of the number

and

(and & pred-forms)

Ands the predicate forms

apply

`(apply f args* coll)`

Applies f to all arguments composed of args and coll

assert

`(assert expr) (assert expr message)`

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

assoc

`(assoc coll key val) (assoc coll key val & kvs)`

When applied to a map, returns a new map of the same 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 \leq (count vector).

atom

`(atom x)`

Creates an atom with the initial value x

atom?

(atom? x)

Returns true if x is an atom, otherwise false

boolean

(boolean x)

Converts to boolean. Everything except 'false' and 'nil' is true in boolean context.

boolean?

(boolean? n)

Returns true if n is a boolean

bytebuf

(bytebuf x)

Converts to bytebuf. x can be a bytebuf, a list/vector of longs, or a string

bytebuf?

(bytebuf? x)

Returns true if x is a bytebuf

bytebuf?

(bytebuf? x)

Returns true if x is a bytebuf

class

(class x)

Returns the class of x

coalesce

(coalesce args*)

Returns the first non nil arg

coll?

(coll? obj)

Returns true if obj is a collection

comment

(comment & body)

Ignores body, yields nil

compare-and-set!

(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

concat

(concat coll) (concat coll & colls)

Returns a collection of the concatenation of the elements in the supplied colls.

concat

(concat coll) (concat coll & colls)

Returns a collection of the concatenation of the elements in the supplied colls.

concat

(concat coll) (concat coll & colls)

Returns a collection of the concatenation of the elements in the supplied colls.

cond

(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.

conj

(conj coll x) (conj coll x & xs)

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

conj

(conj coll x) (conj coll x & xs)

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

conj

(conj coll x) (conj coll x & xs)

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

conj

(conj coll x) (conj coll x & xs)

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

cons

(cons x coll)

Returns a new collection where x is the first element and coll is
the rest

cons

(cons x coll)

Returns a new collection where x is the first element and coll is
the rest

cons

(cons x coll)

Returns a new collection where x is the first element and coll is
the rest

contains?

(contains? coll key)

Returns true if key is present in the given collection, otherwise returns false.

contains?

(contains? coll key)

Returns true if key is present in the given collection, otherwise returns false.

contains?

(contains? coll key)

Returns true if key is present in the given collection, otherwise returns false.

count

(count coll)

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

count

(count coll)

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

count

(count coll)

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

dec

(dec x)

Decrements the number x

dec/add

(dec/add x y scale rounding-mode)

Adds two decimals and scales the result. rounding-mode is one of (:CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, :UP)

dec/div

(dec/div x y scale rounding-mode)

Divides x by y and scales the result. rounding-mode is one of (:CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, :UP)

dec/mul

(dec/mul x y scale rounding-mode)

Multiplies two decimals and scales the result. rounding-mode is one of (:CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, :UP)

dec/scale

(dec/scale x scale rounding-mode)

Scales a decimal. rounding-mode is one of (:CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, :UP)

dec/sub

(dec/sub x y scale rounding-mode)

Subtract y from x and scales the result. rounding-mode is one of (:CEILING, :DOWN, :FLOOR, :HALF_DOWN, :HALF_EVEN, :HALF_UP, :UNNECESSARY, :UP)

decimal?

(decimal? n)

Returns true if n is a decimal

def

(def name expr)

Creates a global variable.

(def val 5)

=> 5

defmacro

(macroexpand form)

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

```
(macroexpand '(-> c (+ 3) (* 2)))  
=> (quote (-> c (+ 3) (* 2)))
```

deref

```
(deref atom)
```

Dereferences an atom, returns its value

do

```
(do exprs)
```

Evaluates the expressions in order and returns the value of the last.

```
(do (println "Test...") (+ 1 1))  
=> 2
```

dotimes

```
(dotimes bindings & body)
```

Repeatedly executes body with name bound to integers from 0 through n-1.

doto

(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.

double?

(double? n)

Returns true if n is a double

drop

(drop n coll)

Returns a collection of all but the first n items in coll

drop

(drop n coll)

Returns a collection of all but the first n items in coll

drop-while

(drop-while predicate coll)

Returns a list of the items in coll starting from the first item for which (predicate item) returns logical false.

drop-while

(drop-while predicate coll)

Returns a list of the items in coll starting from the first item for which (predicate item) returns logical false.

empty-to-nil

(empty-to-nil x)

Returns nil if x is empty

empty-to-nil

(empty-to-nil x)

Returns nil if x is empty

empty?

(empty? x)

Returns true if x is empty

empty?

(empty? x)

Returns true if x is empty

empty?

(empty? x)

Returns true if x is empty

eval

(eval form)

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

```
(eval '(let [a 10] (+ 3 4 a)))
```

=> 17

```
(eval (list + 1 2 3))
```

=> 6

even?

```
(even? n)
```

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

false?

```
(false? x)
```

Returns true if x is false, false otherwise

find

```
(find map key)
```

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

```
(find {:a 1 :b 2} :b)
```

```
=> [:b 2]
```

```
(find {:a 1 :b 2} :z)
```

```
=>
```

first

```
(first coll)
```

Returns the first element of coll.

first

```
(first coll)
```

Returns the first element of coll.

flatten

```
(flatten coll)
```

Takes any nested combination of collections (lists, vectors, etc.) and returns their contents as a single, flat sequence. (flatten nil) returns an empty list.

flatten

(flatten coll)

Takes any nested combination of collections (lists, vectors, etc.) and returns their contents as a single, flat sequence. (flatten nil) returns an empty list.

flatten

(flatten coll)

Takes any nested combination of collections (lists, vectors, etc.) and returns their contents as a single, flat sequence. (flatten nil) returns an empty list.

fn

(fn [params*] exprs*)

Evaluates test.

```
(do (def sum (fn [x y] (+ x y))) (sum 2 3))  
=> 5
```

```
(map (fn [x] (* 2 x)) (range 1 5))  
=> (2 4 6 8)
```

fn?

`(fn? x)`

Returns true if x is a function

gensym

`(gensym)` `(gensym prefix)`

Generates a symbol.

get

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

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

group-by

`(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.

hash-map

(hash-map & keyvals)

Creates a new hash map containing the items.

hash-map?

(hash-map? obj)

Returns true if obj is a hash map

if

(if test true-expr false-expr)

Evaluates test.

```
(if (< 10 20) "yes" "no")  
=> yes
```

if-let

(if-let bindings then else)

bindings is a vector with 2 elements: binding-form test.

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

inc

(inc x)

Increments the number x

into

(into to-coll from-coll)

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

into

(into to-coll from-coll)

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

into

(into to-coll from-coll)

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

into

(into to-coll from-coll)

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

io/copy-file

(io/copy input output)

Copies input to output. Returns nil or throws IOException. Input and output must be a java.io.File.

io/delete-file

(io/delete-file x)

Deletes a file. x must be a java.io.File.

io/exists-dir?

(io/exists-dir? x)

Returns true if the file x exists and is a directory. x must be a java.io.File.

io/exists-file?

(io/exists-file? x)

Returns true if the file x exists. x must be a java.io.File.

io/file

(io/file path) (io/file parent child)

Returns a java.io.File. path, parent, and child can be a string or java.io.File

io/file?

(io/file? x)

Returns true if x is a java.io.File.

io/list-files

(io/list-files dir filterFn?)

Lists files in a directory. dir must be a java.io.File. filterFn is an optional filter that filters the files found

io/tmp-dir

(io/tmp-dir)

Returns the tmp dir as a java.io.File.

io/user-dir

(io/user-dir)

Returns the user dir (current working dir) as a java.io.File.

key

(key e)

Returns the key of the map entry.

keys

(keys map)

Returns a collection of the map's keys.

keyword

(keyword name)

Returns a keyword from the given name

keyword?

(keyword? x)

Returns true if x is a keyword

last

(last coll)

Returns the last element of coll.

last

```
(last coll)
```

Returns the last element of coll.

let

```
(let [bindings*] exprs*)
```

Evaluates the expressions and binds the values to symbols to new local context

```
(let [x 1] x))  
=> 1
```

list

```
(list & items)
```

Creates a new list containing the items.

list-comp

```
(list-comp 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 collection of evaluations of expr. Supported modifiers are: :when test.

```
(list-comp [x (range 10)] x)
```

```
=> (0 1 2 3 4 5 6 7 8 9)
```

```
(list-comp [x (range 5)] (* x 2))
```

```
=> (0 2 4 6 8)
```

```
(list-comp [x (range 10) :when (odd? x)] x)
```

```
=> (1 3 5 7 9)
```

```
(list-comp [x (range 10) :when (odd? x)] (* x 2))
```

```
=> (2 6 10 14 18)
```

```
(list-comp [x (list "abc") y [0 1 2]] [x y])
```

```
=> ([a 0] [a 1] [a 2] [b 0] [b 1] [b 2] [c 0] [c 1] [c 2])
```

list?

```
(list? obj)
```

Returns true if obj is a list

load-file

```
(load-file name)
```

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

load-string

`(load-string s)`

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

```
(load-string "(def x 1)")  
=> 1
```

long?

`(long? n)`

Returns true if n is a long

loop

`(loop [bindings*] exprs*)`

Evaluates the exprs and binds the bindings. Creates a recursion point with the bindings.

```
(loop [x 10]  
  (when (> x 1)  
    (println x)  
    (recur (- x 2))))  
=>
```

macro?

(macro? x)

Returns true if x is a macro

map?

(map? obj)

Returns true if obj is a map

match

(match s regex)

Returns true if the string s matches the regular expression regex

match-not

(match-not s regex)

Returns true if the string s does not match the regular expression regex

max

(max x) (max x y) (max x y & more)

Returns the greatest of the values

memoize

(memoize f)

Returns a memoized version of a referentially transparent function.

```
(do
  (def test (fn [a] (+ a 100)))
  (def test-memo (memoize test))
  (test-memo 1))
=> 101
```

meta

(meta obj)

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

min

(min x) (min x y) (min x y & more)

Returns the smallest of the values

mod

(mod n d)

Modulus of n and d.

neg?

(neg? x)

Returns true if x smaller than zero else false

nil?

(nil? x)

Returns true if x is nil, false otherwise

not

(not x)

Returns true if x is logical false, false otherwise.

not-empty?

(not-empty? x)

Returns true if x is not empty

not-empty?

(not-empty? x)

Returns true if x is not empty

nth

(nth coll idx)

Returns the nth element of coll.

nth

(nth coll idx)

Returns the nth element of coll.

number?

(number? n)

Returns true if n is a number (long, double, or decimal)

odd?

(odd? n)

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

or

(or & pred-forms)

Ors the predicate forms

ordered-map

(ordered-map & keyvals)

Creates a new ordered map containing the items.

ordered-map?

(ordered-map? obj)

Returns true if obj is an ordered map

peek

(peek coll)

For a list, same as first, for a vector, same as last

peek

(peek coll)

For a list, same as first, for a vector, same as last

pop

(pop coll)

For a list, returns a new list without the first item, for a vector, returns a new vector without the last item.

pop

(pop coll)

For a list, returns a new list without the first item, for a vector, returns a new vector without the last item.

pos?

(pos? x)

Returns true if x greater than zero else false

pr-str

(pr_str & xs)

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

println

(println & xs)

Prints to stdout with a trailing linefeed, with no args, prints the empty string. With one arg `x`, prints `x.toString()`. With more than one arg, prints the concatenation of the str values of the args with delimiter ' '.The function is sandboxed.

prn

(prn & xs)

Prints to stdout, with no args, prints the empty string. With one arg `x`, prints `x.toString()`. With more than one arg, prints the concatenation of the str values of the args with delimiter ' '.The function is sandboxed.

rand-double

(rand-double) (rand-double max)

Without argument returns a double long between 0.0 and 1.0. Without argument max returns a random long between 0.0 and max.

rand-long

(rand-long) (rand-long max)

Without argument returns a random long between 0 and MAX_LONG. Without argument max returns a random long between 0 and max exclusive.

range

(range end) (range start end) (range start end step)

Returns a collection of numbers from start (inclusive) to end (exclusive), by step, where start defaults to 0 and step defaults to 1. When start is equal to end, returns empty list.

read-string

(read-string x)

Reads from x

readline

(readline prompt)

Reads the next line from stdin. The function is sandboxed

recur

(recur expr*)

Evaluates the exprs and rebinds the bindings of the recursion point to the values of the exprs.

reduce

(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.

reduce

(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.

reduce-kv

`(reduce-kv f init coll)`

Reduces an associative collection. `f` should be a function of 3 arguments. Returns the result of applying `f` to `init`, the first key and the first value in `coll`, then applying `f` to that result and the 2nd key and value, etc. If `coll` contains no entries, returns `init` and `f` is not called. Note that `reduce-kv` is supported on vectors, where the keys will be the ordinals.

remove

`(remove predicate coll)`

Returns a collection of the items in `coll` for which `(predicate item)` returns logical false.

repeat

`(repeat n x)`

Returns a collection with the value `x` repeated `n` times

reset!

(reset! atom newval)

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

rest

(rest coll)

Returns a collection with second to list element

rest

(rest coll)

Returns a collection with second to list element

rest

(rest coll)

Returns a collection with second to list element

rest

(rest coll)

Returns a collection with second to list element

reverse

(reverse coll)

Returns a collection of the items in coll in reverse order

reverse

(reverse coll)

Returns a collection of the items in coll in reverse order

second

(second coll)

Returns the second element of coll.

second

(second coll)

Returns the second element of coll.

seq?

(seq? obj)

Returns true if obj is a sequential collection

set

(set & items)

Creates a new set containing the items.

set?

(set? obj)

Returns true if obj is a set

slurp

(slurp file & options)

Returns the file's content as text (string) or binary (bytebuf). Defaults to binary=false and encoding=UTF-8. Options: :encoding "UTF-8" :binary true/false.

some?

(some? x)

Returns true if x is not nil, false otherwise

sort

(sort coll) (sort compfn coll)

Returns a sorted list of the items in coll. If no compare function compfn is supplied, uses the natural compare. The compare function takes two arguments and returns -1, 0, or 1

sort

(sort coll) (sort compfn coll)

Returns a sorted list of the items in coll. If no compare function compfn is supplied, uses the natural compare. The compare function takes two arguments and returns -1, 0, or 1

sort-by

(sort-by keyfn coll) (sort-by keyfn compfn 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.

sort-by

(sort-by keyfn coll) (sort-by keyfn compfn 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.

sorted-map

(sorted-map & keyvals)

Creates a new sorted map containing the items.

sorted-map?

(sorted-map? obj)

Returns true if obj is a sorted map

spit

(spit f content & options)

Opens f, writes content, and then closes f. Defaults to append=true and encoding=UTF-8. Options: :append true/false, :encoding "UTF-8"

str

(str & xs)

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.

str/contains?

(str/contains? s substr)

True if s contains with substr.

str/ends-with?

(str/ends-with? s substr)

True if s ends with substr.

str/format

(str/format s format args*)

Returns a formatted string using the specified format string and arguments.

str/index-of

(str/index-of s value) (str/index-of s value from-index)

Return index of value (string or char) in s, optionally searching forward from from-index.
Return nil if value not found.

str/join

(str/join coll) (str/join separator coll)

Joins all elements in coll separated by an optional separator.

str/last-index-of

(str/last-index-of s value) (str/last-index-of s value from-index)

Return last index of value (string or char) in s, optionally searching backward from from-index. Return nil if value not found.

str/lower-case

(str/lower-case s)

Converts s to lowercase

str/replace-all

(str/replace-all s search replacement)

Replaces the all occurrences of search in s

str/replace-first

(str/replace-first s search replacement)

Replaces the first occurrence of search in s

str/replace-last

(str/replace-last s search replacement)

Replaces the last occurrence of search in s

str/split

(str/split s regex)

Splits string on a regular expression.

str/starts-with?

(str/starts-with? s substr)

True if s starts with substr.

str/subs

(str/subs s start) (str/subs s start end)

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

str/trim

(str/trim s substr)

Trims leading and trailing spaces from s.

str/trim-to-nil

(str/trim-to-nil s substr)

Trims leading and trailing spaces from s. Returns nil if the resulting string is empty

str/truncate

(str/truncate s maxlen marker)

Truncates a string to the max lenght maxlen and adds the marker to the end if the string needs to be truncated

```
(str/truncate "abcdefghij" 20 "...")  
=> abcdefghij
```

```
(str/truncate "abcdefghij" 9 "...")  
=> abcdef...
```

```
(str/truncate "abcdefghij" 4 "...")  
=> a...
```

str/upper-case

```
(str/upper-case s)
```

Converts s to uppercase

string?

```
(string? x)
```

Returns true if x is a string

subbytebuf

```
(subbytebuf x start) (subbytebuf x start end)
```

Returns a byte buffer of the items in buffer from start (inclusive) to end (exclusive). If end is not supplied, defaults to (count bytearray)

subvec

(subvec v start) (subvec v start end)

symbol?

(symbol? x)

Returns true if x is a symbol

take

(take n coll)

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

take

(take n coll)

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

take-while

(take-while predicate coll)

Returns a list of successive items from coll while (predicate item) returns logical true.

take-while

(take-while predicate coll)

Returns a list of successive items from coll while (predicate item) returns logical true.

throw

(throw) (throw x)

Throws exception with passed value x

time

(time [expr])

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

time-ms

(time-ms)

Returns the current time in milliseconds


```
(time-ms)  
=> 1531812377899
```

time-ns

```
(time-ns)
```

Returns the current value of the running Java Virtual Machine's high-resolution time source, in nanoseconds.

```
(time-ns)  
=> 403479792809369
```

true?

```
(true? x)
```

Returns true if x is true, false otherwise

try

```
(try (throw)) (try (throw expr)) (try (throw expr) (catch expr)) (try (throw expr) (catch expr)  
(finally expr))
```

Exception handling: try - catch -finally

```
(try (throw))  
=> VncException: nil
```

```
(try (throw "test message"))  
=> VncException: test message
```

```
(try (throw 100) (catch (do (+ 1 2) -1)))  
=> -1
```

```
(try (throw 100) (finally -2))  
=> -2
```

```
(try (throw 100) (catch (do (+ 1 2) -1)) (finally -2))  
=> -2
```

uuid

```
(uuid)
```

Generates a UUID.

val

```
(val e)
```

Returns the val of the map entry.

vals

```
(vals map)
```

Returns a collection of the map's values.

vary-meta

(vary-meta obj f & args)

Returns a copy of the object obj, with (apply f (meta obj) args) as its metadata.

vector

(vector & items)

Creates a new vector containing the items.

vector?

(vector? obj)

Returns true if obj is a vector

version

(version)

Returns the version.

when

(when test & body)

Evaluates test. If logical true, evaluates body in an implicit do.

when-not

(when-not test & body)

Evaluates test. If logical false, evaluates body in an implicit do.

while

(take-while pred) (take-while pred coll)

Repeatedly executes body while test expression is true. Presumes some side-effect will cause test to become false/nil. Returns nil

with-meta

(with-meta obj m)

Returns a copy of the object obj, with a map m as its metadata.

zero?

(zero? x)

Returns true if x zero else false