Venice V 1.8.3   Cheat Sheet

Primitives

Literals

Literals	Nil: nil
	Boolean: true, false
	Integer: 150I
	Long: 1500
	Double: 3.569
	BigDecimal: 6.897M
	String: "abcd", "ab\"cd", "PI: \u03C0"
	String: """{ "age": 42 }"""

Numbers

Arithmetic	+ - * / mod inc dec min max abs negate floor ceil sqrt square pow log log10
Convert	int long double decimal
Compare	== = < > <= >= compare
Test	zero? pos? neg? even? odd? number? int? long? double? decimal?
Random	rand-long rand-double rand-gaussian
Trigonometry	to-radians to-degrees sin cos tan
Statistics	mean median quartiles quantile standard-deviation
BigDecimal	dec/add dec/sub dec/mul dec/div dec/scale

Strings

Create	str str/format str/quote str/double-quote str/double-unquote
Use	count compare empty-to-nil first last nth nfirst nlast seq reverse shuffle str/index-of str/last-index-of str/subs str/chars str/repeat str/reverse str/truncate str/expand str/lorem-ipsum
Split/Join	str/split str/split-lines str/join

Collections

Collections

Generic	count compare empty-to-nil empty into cons conj remove repeat repeatedly replace range group-by frequencies get-in seq reverse shuffle
Tests	empty? not-empty? coll? list? vector? set? sorted-set? mutable-set? map? sequential? hash-map? ordered-map? sorted-map? mutable-map? bytebuf?
Process	map map-indexed filter reduce keep docoll

Lists

Create	() list list* mutable-list
Access	first second third fourth nth last peek rest butlast nfirst nlast some
Modify	cons conj rest pop into concat interpose interleave mapcat flatten sort sort-by take take-while drop drop-while split-at split-with
Test	list? mutable-list? every? not-every? any? not-any?

Vectors

Create	[] vector mapv
Access	first second third nth last peek butlast rest nfirst nlast subvec some
Modify	cons conj rest pop into concat distinct dedupe partition interpose interleave mapcat flatten sort sort-by take take-while drop drop-while update update! split-with
Nested	get-in assoc-in update-in dissoc-in
Test	

Replace	str/replace-first str/replace-last str/replace-all
Strip	str/strip-start str/strip-end str/strip-indent str/strip-margin
Conversion	str/lower-case str/upper-case str/cr-lf
Regex	match? not-match?
Trim	str/trim str/trim-to-nil
Hex	str/hex-to-bytebuf str/bytebuf-to-hex str/format-bytebuf
Encode/Decode	str/encode-base64 str/decode-base64 str/encode-url str/decode-url str/escape-html str/escape-xml
Validation	str/valid-email-addr?
Test	string? empty? str/blank? str/starts-with? str/ends-with? str/contains? str/equals-ignore-case? str/quoted? str/double-quoted?
Test char	str/char? str/digit? str/letter? str/whitespace? str/linefeed? str/lower-case? str/upper-case?

Chars

Use	char char?

Other

Keywords	:a :blue keyword? keyword
Symbols	'a 'blue symbol? symbol
Just	just just?
Boolean	boolean not boolean? true? false?

	vector? contains? not-contains? every? not-every? any? not-any?

Sets

Create	#{} set sorted-set mutable-set
Modify	cons cons! conj conj! disj difference union intersection
Test	set? sorted-set? mutable-set? contains? not-contains? every? not-every? any? not-any?

Maps

Create	{} hash-map ordered-map sorted-map mutable-map zipmap
Access	find get keys vals key val entries
Modify	cons conj assoc assoc! update update! dissoc dissoc! into concat flatten filter-k filter-kv reduce-kv merge merge-with map-invert
Nested	get-in assoc-in update-in dissoc-in
Test	map? sequential? hash-map? ordered-map? sorted-map? mutable-map? map-entry? contains? not-contains?

Stack

Create	stack
Access	peek pop! push! count
Test	empty? stack?

Queue

Create	queue
Access	peek poll! offer! count
Test	empty? queue?

Byte Buffer

Create	bytebuf bytebuf-allocate bytebuf-from-string
Test	empty? not-empty? bytebuf?
Use	count bytebuf-capacity bytebuf-limit bytebuf-to-string bytebuf-sub bytebuf-pos bytebuf-pos! bytebuf-put-byte! bytebuf-put-long! bytebuf-put-int! bytebuf-put-buf!

Arrays

Create	make-array object-array string-array int-array long-array float-array double-array
Use	aget aset alength asub acopy amap

Regex

Base64	str/encode-base64 str/decode-base64
Hex	str/hex-to-bytebuf str/bytebuf-to-hex str/format-bytebuf

Time

Date	time/date time/date?
Local Date	time/local-date time/local-date? time/local-date-parse
Local Date Time	time/local-date-time time /local-date-time? time /local-date-time-parse
Zoned Date Time	time/zoned-date-time time /zoned-date-time? time /zoned-date-time-parse
Fields	time/year time/month time /day-of-week time/day-of-month time/day-of-year time/hour time/minute time/second
Fields etc	time/length-of-year time /length-of-month time /first-day-of-month time /last-day-of-month
Zone	time/zone time/zone-offset
Format	time/formatter time/format
Test	time/after? time/not-after? time/before? time/not-before? time/within? time/leap-year?
Miscellaneous	time/with-time time/plus time /minus time/period time /earliest time/latest
Util	time/zone-ids time/to-millis

Transducers

Use	transduce
Functions	map map-indexed filter drop drop-while take take-while keep remove dedupe distinct sorted reverse flatten halt-when
Reductions	rf-first rf-last rf-every? rf-any?
Early	reduced reduced? deref deref?

Functions

Create

General

regex/pattern   regex/matcher   regex /find   regex/find-all   regex/find-group regex/find-all-groups   regex/reset regex/find?   regex/matches   regex /matches?   regex/group   regex /groupcount

Concurrency

Atoms	atom atom? deref deref? reset! swap! compare-and-set! add-watch remove-watch
Futures	future future? future-done? future-cancel future-cancelled? futures-fork futures-wait deref deref? realized?
Promises	promise promise? deliver realized?
Delay	delay delay? deref deref? force realized?
Agents	agent send send-off restart-agent set-error-handler! agent-error await await-for shutdown-agents shutdown-agents? await-termination-agents await-termination-agents?
Scheduler	schedule-delay schedule-at-fixed-rate
Locking	locking
Volatiles	volatile volatile? deref deref? reset! swap!
ThreadLocal	thread-local thread-local? thread-local-clear assoc dissoc get
Threads	thread-id thread-name thread-interrupted? thread-interrupted

System

Venice	version sandboxed?
System	system-prop system-env system-exit-code charset-default-encoding
Java	java-version java-version-info
OS	os-type os-type? os-arch os-name os-version
Time	current-time-millis nano-time format-nano-time

	fn defn defn- identity comp partial memoize juxt fnil trampoline complement constantly every-pred any-pred
Call	apply -> ->>
Test	fn?
Exception	throw
Misc	nil? some? doc eval name callstack coalesce load-resource
Environment	var-get bound? resolve
Tree Walker	prewalk postwalk
Meta	meta with-meta vary-meta

Macros

Create	defn defn- defmacro macroexpand macroexpand-all
Branch	and or when when-not if-not if-let when-let
Loop	list-comp dotimes while
Call	doto -> ->> -<> as->
Loading	load-module load-file load-classpath-file load-string
Test	macro? cond condp case
Assert	assert
Util	comment gensym time with-out-str with-err-str
Profiling	time perf

Special Forms

Forms	def defonce def-dynamic defmulti defmethod if do let binding fn loop recur set! try try-with
Profiling	dobench dorun prof

Types

Test	type supertype instance? deftype?
Define	deftype deftype-of deftype-or
Create	.:

Other	uuid sleep host-name host-address gc pid shutdown-hook
Shell	sh with-sh-dir with-sh-env with-sh-throw

IO

to	print println printf flush newline
to-str	pr-str with-out-str
from	read-line read-string
file	io/file io/file-parent io/file-name io/file-path io/file-absolute-path io/file-canonical-path io/file-ext? io/file-size
file dir	io/mkdir io/mkdirs
file i/o	io/slurp io/slurp-lines io/spit io/copy-file io/move-file io/delete-file io/delete-file-on-exit io/delete-file-tree
file list	io/list-files io/list-files-glob io/list-file-tree
file test	io/file? io/exists-file? io/exists-dir? io/file-can-read? io/file-can-write? io/file-can-execute? io/file-hidden?
file other	io/temp-file io/tmp-dir io/user-dir io/user-home-dir
classpath	io/load-classpath-resource io/classpath-resource?
stream	io/copy-stream io/slurp-stream io/spit-stream io/uri-stream io/wrap-os-with-buffered-writer io/wrap-os-with-print-writer io/wrap-is-with-buffered-reader
http	io/download io/internet-avail?
zip	io/zip io/zip-file io/zip-list io/zip-list-entry-names io/zip-append io/zip-remove io/zip? io/unzip io/unzip-first io/unzip-nth io/unzip-all io/unzip-to-dir
gzip	io/gzip io/gzip-to-stream io/gzip? io/ungzip io/ungzip-to-stream
other	with-out-str io/mime-type io/default-charset

Miscellaneous

JSON

Namespace

Current	*ns*
Remove	ns-unmap ns-remove

Java Interoperability

Java	. proxify import java-iterator-to-list java-enumeration-to-list cast class
Test	java-obj? exists-class?
Support	imports supers bases formal-type

Application

Management	app/build app/manifest

	json/write-str json/read-str json/spit json/slurp json /pretty-print
XML	xml/parse-str xml/parse xml /path-> xml/children xml/text
PDF	pdf/render pdf/watermark pdf /text-to-pdf pdf/available? pdf /check-required-libs
PDF Tools	pdf/merge pdf/copy pdf/pages
Cryptography	crypt/md5-hash crypt/sha512-hash crypt/pbkdf2-hash crypt/encrypt crypt/decrypt
CSV	csv/read csv/write csv/write-str
CIDR	cidr/parse cidr/in-range? cidr /inet-addr
Other	*version* *newline* *loaded-modules* *loaded-files* *ns* *run-mode* *ansi-term*

Embedding in Java

Eval

```java
import com.github.jlangch.venice.Venice;

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

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

Passing parameters

```java
import com.github.jlangch.venice.Venice;
import com.github.jlangch.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

```java
import com.github.jlangch.venice.Venice;
import com.github.jlangch.venice.PreCompiled;

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

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

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

Java Interop

```java
import java.time.ZonedDateTime;
import com.github.jlangch.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

```java
import com.github.jlangch.venice.Venice;
import com.github.jlangch.venice.javainterop.*;

public class Example {
    public static void main(String[] args) {
        final IInterceptor interceptor =
            new SandboxInterceptor(
                new SandboxRules()
                        .rejectAllVeniceIoFunctions()
                        .allowAccessToStandardSystemProperties()
                        .withClasses(
                          "java.lang.Math:min",
                          "java.time.ZonedDateTime:*",
                          "java.util.ArrayList:new",
                          "java.util.ArrayList:add"));

        final Venice venice = new Venice(interceptor);

        // => OK (static method)
        venice.eval("(. :java.lang.Math :min 20 30)");

        // => OK (constructor & instance method)
        venice.eval("(. (. :java.time.ZonedDateTime :now) :plusDays 5))");

        // => OK (constructor & instance method)
        venice.eval(
            "(doto (. :java.util.ArrayList :new)  " +
            "      (. :add 1)                     " +
            "      (. :add 2))                    ");

        // => FAIL (invoking non whitelisted static method)
        venice.eval("(. :java.lang.System :exit 0)");

        // => FAIL (invoking rejected Venice I/O function)
        venice.eval("(io/slurp \"/tmp/file\")");

        // => FAIL (accessing non whitelisted system property)
        venice.eval("(system-prop \"db.password\")");
    }
}
```

Function details

#{}

Creates a set.

```
#{10 20 30}
=> #{10 20 30}
```

()

Creates a list.

```
'(10 20 30)
=> (10 20 30)
```

*

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

Returns the product of numbers. (*) returns 1

```
(*)
=> 1

(* 4)
=> 4

(* 4 3)
=> 12

(* 4 3 2)
=> 24

(* 4I 3I)
=> 12I

(* 6.0 2)
=> 12.0

(* 6 1.5M)
=> 9.0M
```

ansi-term

True if Venice runs in an ANSI terminal, otherwise false

```
*ansi-term*
=> false
```

loaded-files

The loaded files

```
*loaded-files*
=> #{}
```

loaded-modules

The loaded modules

```
*loaded-modules*
=> #{:pdf :str :csv :io :time :json :cidr :regex :core}
```

newline

The system newline

```
*newline*
=> "\n"
```

ns

The current namespace

```
*ns*
=> user

(do
  (ns test)
  *ns*)
=> test
```

run-mode

The current run-mode one of (:repl, :script, :app)

```
*run-mode*
=> :script
```

version

The Venice version

```
*version*
=> "0.0.0"
```

+

```
(+)
(+ 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

(+ 1I 2I)
=> 3I

(+ 1 2.5)
=> 3.5

(+ 1 2.5M)
=> 3.5M
```

-

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

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

```
(- 4)
=> -4

(- 8 3 -2 -1)
=> 8

(- 5I 2I)
=> 3I

(- 8 2.5)
=> 5.5

(- 8 1.5M)
=> 6.5M
```

-<>

```
(-<> x & forms)
```

Threads the x through the forms. Inserts x at position of the <>
symbol of the first form, making a list of it if is not a list
already. If there are more forms, inserts the first form at position
of the <> symbol in second form, etc.

```
(-<> 5
    (+ <> 3)
    (/ 2 <>)
    (- <> 1))
=> -1
```

->

```
(-> x & forms)
```

Threads the x 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.

```
(-> 5 (+ 3) (/ 2) (- 1))
=> 3

(do
  (def person
       {:name "Peter Meier"
        :address {:street "Lindenstrasse 45"
                  :city "Bern"
                  :zip 3000}})

  (-> person :address :street))
=> "Lindenstrasse 45"
```

->>

```
(->> x & forms)
```

Threads the x 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.

```
(->> 5 (+ 3) (/ 32) (- 1))
=> -3

(->> [ {:a 1 :b 2} {:a 3 :b 4} {:a 5 :b 6} {:a 7 :b 8} ]
     (map (fn [x] (get x :b)))
     (filter (fn [x] (> x 4)))
     (map inc))))
=> (7 9)
```

.

```
(. classname :new args)
(. classname method-name args)
(. classname field-name)
(. classname :class)
(. object method-name args)
(. object field-name)
(. object :class)
```

Java interop. Calls a constructor or an class/object method or accesses a class/instance field. The function is
sandboxed.

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

;; invoke static method
(. :java.time.ZonedDateTime :now)
=> 2020-06-03T10:26:01.956+02:00[Europe/Zurich]

;; invoke static method
(. :java.lang.Math :min 10 20)
=> 10

;; access static field
(. :java.lang.Math :PI)
=> 3.141592653589793

;; invoke method
(. (. :java.lang.Long :new 10) :toString)
=> "10"

;; get class name
(. :java.lang.Math :class)
=> class java.lang.Math
```

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

.:

```
(.: type-name args*)
```

Instantiates a custom type.

```
(do
  (ns foo)
  (deftype :complex [real :long, imaginary :long])
  (def x (.: :complex 100 200))
  [(:real x) (:imaginary x)])
=> [100 200]
```

/

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

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

```
(/ 2.0)
=> 0.5

(/ 12 2 3)
=> 2

(/ 12 3)
=> 4

(/ 12I 3I)
=> 4I

(/ 6.0 2)
=> 3.0

(/ 6 1.5M)
=> 4.0000000000000000M
```

<

```
(< x y)
```

Returns true if x is smaller than y

```
(< 2 3)
=> true

(< 2 3.0)
=> true

(< 2 3.0M)
=> true
```

<=

```
(<= x y)
```

Returns true if x is smaller or equal to y

```
(<= 2 3)
=> true

(<= 3 3)
=> true

(<= 2 3.0)
=> true

(<= 2 3.0M)
=> true
```

=

```
(= x y)
```

Returns true if both operands have equivalent type and value

```
(= 0 0)
=> true

(= 0 1)
=> false

(= 0 0.0)
=> false

(= 0 0.0M)
=> false
```

==

```
(== x y)
```

Returns true if both operands have equivalent value

```
(== 0 0)
=> true

(== 0 1)
=> false

(== 0 0.0)
=> true

(== 0 0.0M)
=> true
```

>

```
(> x y)
```

Returns true if x is greater than y

```
(> 3 2)
=> true

(> 3 3)
=> false

(> 3.0 2)
=> true

(> 3.0M 2)
=> true
```

>=

```
(>= x y)
```

Returns true if x is greater or equal to y

```
(>= 3 2)
=> true

(>= 3 3)
=> true

(>= 3.0 2)
=> true

(>= 3.0M 2)
=> true
```

[]

Creates a vector.

```
[10 20 30]
=> [10 20 30]
```

abs

```
(abs x)
```

Returns the absolute value of the number

```
(abs 10)
=> 10

(abs -10)
=> 10

(abs -10I)
=> 10I

(abs -10.1)
=> 10.1

(abs -10.12M)
=> 10.12M
```

acopy

```
(acopy src src-pos dest dest-pos dest-len)
```

Copies an array from the src array, beginning at the specified position, to the specified position of the dest array. Returns the modified destination array

```
(acopy (long-array '(1 2 3 4 5)) 2 (long-array 20) 10 3)
=> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 4, 5, 0, 0, 0, 0, 0, 0, 0]
```

add-watch

```
(add-watch ref key fn)
```

Adds a watch function to an agent/atom reference. The watch fn must be a fn of 4 args: a key, the reference, its old-state, its new-state.

```
(do
   (def x (agent 10))
   (defn watcher [key ref old new]
        (println "watcher: " key))
   (add-watch x :test watcher))
=> nil
```

agent

```
(agent state & options)
```

Creates and returns an agent with an initial value of state and zero or more options.

```
Options:
  :error-handler handler-fn
  :error-mode mode-keyword
```
The handler-fn is called if an action throws an exception. It's afunction taking two args the agent and the exception. The mode-keyword may be either :continue (the default) or :fail

```
(do
   (def x (agent 100))
   (send x + 5)
   (sleep 100)
   (deref x))
=> 105
```

agent-error

```
(agent-error agent)
```

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

```
(do
   (def x (agent 100 :error-mode :fail))
   (send x (fn [n] (/ n 0)))
   (sleep 500)
   (agent-error x))
=> com.github.jlangch.venice.VncException: / by zero
```

aget

```
(aget array idx)
```

Returns the value at the index of an array of Java Objects

```
(aget (long-array '(1 2 3 4 5)) 1)
=> 2
```

alength

```
(alength array)
```

Returns the length of an array

```
(alength (long-array '(1 2 3 4 5)))
=> 5
```

amap

```
(amap f arr)
```

Applys f to each item in the array arr. Returns a new array with the mapped values.

```
(str (amap (fn [x] (+ 1 x)) (long-array 6 0)))
=> "[1, 1, 1, 1, 1, 1]"
```

and

```
(and x)
(and x & next)
```

Ands the predicate forms

```
(and true true)
=> true

(and true false)
=> false
```

any-pred

```
(any-pred p1 & p)
```

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.

```
((any-pred number?) 1)
=> true

((any-pred number?) 1 "a")
=> true

((any-pred number? string?) 2 "a")
=> true
```

any?

```
(any? pred coll)
```

Returns true if the predicate is true for at least one collection item, false otherwise.

```
(any? number? nil)
=> false

(any? number? [])
=> false

(any? number? [1 :a :b])
=> true

(any? number? [1 2 3])
=> true

(any? #(== % 10) [10 20 30])
=> true

(any? #(>= % 10) [1 5 10])
=> true
```

app/build

```
(app/build name main-file file-map dest-dir)
```

Creates a Venice application archive that can be distributed
and executed as a single file.

```
E.g.:
(app/build "test"
           "chart.venice"
           { "chart.venice" "./foo/chart.venice"
             "utils.venice" "./foo/utils.venice" }
           ".")
```

Loading Venice files works relative to the application. You can
only load files that are in the app archive.
If for instances "chart.venice" in the above example requires
"utils.venice" just add (load-file "utils") to "chart.venice".

The app can be run from the command line as:
```
> java -jar venice-1.7.17.jar -app test.zip
```

Or with additional Java libraries (all JARs in 'libs' dir):
```
> java -cp "libs/*" com.github.jlangch.venice.Launcher -app test.zip
```

app/manifest

```
(app/manifest app)
```

Returns the manifest of a Venice application archive.

apply

```
(apply f args* coll)
```

Applies f to all arguments composed of args and coll

```
(apply + [1 2 3])
=> 6

(apply + 1 2 [3 4 5])
=> 15

(apply str [1 2 3 4 5])
=> "12345"

(apply inc [1])
=> 2
```

as->

```
(as-> expr name & forms)
```

Binds name to expr, evaluates the first form in the lexical context of
that binding, then binds name to that result, repeating for each
successive form, returning the result of the last form. This allows a
value to thread into any argument position.

```
; allows to use arbitrary positioning of the argument
(as-> [:foo :bar] v
      (map name v)
      (first v)
      (str/subs v 1))
=> "oo"

; allows the use of if statements in the thread
(as-> {:a 1 :b 2} m
      (update m :a #(+ % 10))
      (if true
        (update m :b #(+ % 10))
        m))
=> {:a 11 :b 12}
```

aset

```
(aset array idx val)
```

Sets the value at the index of an array

```
(aset (long-array '(1 2 3 4 5)) 1 20)
=> [1, 20, 3, 4, 5]
```

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 <= (count vector).

```
(assoc {} :a 1 :b 2)
=> {:a 1 :b 2}

(assoc nil :a 1 :b 2)
=> {:a 1 :b 2}

(assoc [1 2 3] 0 10)
=> [10 2 3]

(assoc [1 2 3] 3 10)
=> [1 2 3 10]

(assoc [1 2 3] 6 10)
=> [1 2 3 10]
```

assoc!

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

Associates key/vals with a mutable map, returns the map

```
(assoc! (mutable-map ) :a 1 :b 2)
=> {:a 1 :b 2}

(assoc! nil :a 1 :b 2)
=> {:a 1 :b 2}
```

assoc-in

```
(assoc-in m 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 or vectors will be created.

```
(do
  (def users [ {:name "James" :age 26}
               {:name "John" :age 43}] )
  (assoc-in users [1 :age] 44))
=> [{:age 26 :name "James"} {:age 44 :name "John"}]

(do
  (def users [ {:name "James" :age 26}
               {:name "John" :age 43}] )
  (assoc-in users [2] {:name "Jack" :age 19}) )
=> [{:age 26 :name "James"} {:age 43 :name "John"} {:age 19 :name "Jack"}]
```

asub

```
(asub array start len)
```

Returns a sub array

```
(asub (long-array '(1 2 3 4 5)) 2 3)
=> [3, 4, 5]
```

atom

```
(atom x)
```

Creates an atom with the initial value x

```
(do
  (def counter (atom 0))
  (swap! counter inc)
  (deref counter))
=> 1

(do
  (def counter (atom 0))
  (reset! counter 9)
  @counter)
=> 9
```

atom?

```
(atom? x)
```

Returns true if x is an atom, otherwise false

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

```
   (atom? counter))
=> true
```

await

```
(await agents)
```

Blocks the current thread (indefinitely) until all actions dispatched thus far (from this thread or agent) to the agents have occurred.

```
(do
   (def x1 (agent 100))
   (def x2 (agent {}))
   (send-off x1 + 5)
   (send-off x2 (fn [state]
               (sleep 100)
               (assoc state :done true)))
   ;; blocks till the agent actions are finished
   (await x1 x2))
=> true
```

await-for

```
(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 logical false if returning due to timeout, logical true otherwise.

```
(do
   (def x1 (agent 100))
   (def x2 (agent {}))
   (send-off x1 + 5)
   (send-off x2 (fn [state]
               (sleep 100)
               (assoc state :done true)))
   ;; blocks till the agent actions are finished
   (await-for 500 x1 x2))
=> true
```

await-termination-agents

```
(shutdown-agents )
```

Blocks until all actions have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

```
(do
   (def x1 (agent 100))
   (def x2 (agent 100))
```

```
    (shutdown-agents )
    (await-termination-agents 1000))
```

await-termination-agents?

```
(await-termination-agents?)
```

Returns true if all tasks have been completed following agent shut down

```
(do
   (def x1 (agent 100))
   (def x2 (agent 100))
   (shutdown-agents )
   (await-termination-agents 1000))
   (sleep 300)
   (await-termination-agents? ))
```

bases

```
(bases class)
```

Returns the immediate superclass and interfaces of class, if any.

```
(bases :java.util.ArrayList)
=> (:java.util.AbstractList :java.util.List :java.util.RandomAccess :java.lang.Cloneable :java.io.Serializable)
```

binding

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

Evaluates the expressions and binds the values to dynamic (thread-local) symbols

```
(do
   (binding [x 100]
     (println x)
     (binding [x 200]
       (println x))
     (println x)))
100
200
100
=> nil
```

boolean

```
(boolean x)
```

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

```
(boolean false)
=> false

(boolean true)
=> true

(boolean nil)
=> false

(boolean 100)
=> true
```

boolean?

```
(boolean? n)
```

Returns true if n is a boolean

```
(boolean? true)
=> true

(boolean? false)
=> true

(boolean? nil)
=> false

(boolean? 0)
=> false
```

bound?

```
(bound? s)
```

Returns true if the symbol is bound to a value else false

```
(bound? 'test)
=> false

(let [test 100] (bound? 'test))
=> true
```

butlast

```
(butlast coll)
```

Returns a collection with all but the last list element

```
(butlast nil)
=> nil

(butlast [])
=> []

(butlast [1])
=> []

(butlast [1 2 3])
=> [1 2]

(butlast '())
=> ()

(butlast '(1))
=> ()

(butlast '(1 2 3))
=> (1 2)

(butlast "1234")
=> ("1" "2" "3")
```

bytebuf

```
(bytebuf x)
```

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

```
(bytebuf [0 1 2])
=> [0 1 2]

(bytebuf '(0 1 2))
=> [0 1 2]

(bytebuf "abc")
=> [97 98 99]
```

bytebuf-allocate

```
(bytebuf-allocate length)
```

Allocates a new bytebuf. The values will be all zero.

```
(bytebuf-allocate 20)
=> [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

bytebuf-capacity

```
(bytebuf-capacity buf)
```

Returns the capacity of a bytebuf.

```
(bytebuf-capacity (bytebuf-allocate 100))
=> 100
```

bytebuf-from-string

```
(bytebuf-from-string s encoding)
```

Converts a string to a bytebuf using an optional encoding. The encoding defaults to :UTF-8

```
(bytebuf-from-string "abcdef" :UTF-8)
=> [97 98 99 100 101 102]
```

bytebuf-limit

```
(bytebuf-limit buf)
```

Returns the limit of a bytebuf.

```
(bytebuf-limit (bytebuf-allocate 100))
=> 100
```

bytebuf-pos

```
(bytebuf-pos buf)
```

Returns the buffer's current position.

```
(bytebuf-pos (bytebuf-allocate 10))
=> 0
```

bytebuf-pos!

```
(bytebuf-pos! buf pos)
```

Sets the buffer's position.

```
(-> (bytebuf-allocate 10)
    (bytebuf-pos! 4)
    (bytebuf-put-byte! 1)
    (bytebuf-pos! 8)
    (bytebuf-put-byte! 2))
=> [0 0 0 0 1 0 0 0 2 0]
```

bytebuf-put-buf!

```
(bytebuf-put-buf! dst src src-offset length)
```

This method transfers bytes from the src to the dst buffer at the current position, and then increments the position by length.

```
(-> (bytebuf-allocate 10)
    (bytebuf-pos! 4)
    (bytebuf-put-buf! (bytebuf [1 2 3]) 0 2))
=> [0 0 0 0 1 2 0 0 0 0]
```

bytebuf-put-byte!

```
(bytebuf-put-byte! buf b)
```

Writes a byte to the buffer at the current position, and thenincrements the position by one.

```
(-> (bytebuf-allocate 4)
    (bytebuf-put-byte! 1)
    (bytebuf-put-byte! 2))
=> [1 2 0 0]
```

bytebuf-put-int!

```
(bytebuf-put-int! buf i)
```

Writes an integer (4 bytes) to buffer at the current position, and thenincrements the position by four.

```
(-> (bytebuf-allocate 8)
    (bytebuf-put-int! 4I)
    (bytebuf-put-int! 8I))
=> [0 0 0 4 0 0 0 8]
```

bytebuf-put-long!

```
(bytebuf-put-long! buf l)
```

Writes a long (8 bytes) to buffer at the current position, and thenincrements the position by eight.

```
(-> (bytebuf-allocate 16)
    (bytebuf-put-long! 4)
    (bytebuf-put-long! 8))
=> [0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 8]
```

bytebuf-sub

```
(bytebuf-sub x start) (bytebuf-sub 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 bytebuffer)

```
(bytebuf-sub (bytebuf [1 2 3 4 5 6]) 2)
=> [3 4 5 6]
```

```
(bytebuf-sub (bytebuf [1 2 3 4 5 6]) 4)
=> [5 6]
```

bytebuf-to-string

```
(bytebuf-to-string buf encoding)
```

Converts a bytebuf to a string using an optional encoding. The encoding defaults to :UTF-8

```
(bytebuf-to-string (bytebuf [97 98 99]) :UTF-8)
=> "abc"
```

bytebuf?

```
(bytebuf? x)
```

Returns true if x is a bytebuf

```
(bytebuf? (bytebuf [1 2]))
=> true
```

```
(bytebuf? [1 2])
=> false
```

```
(bytebuf? nil)
=> false
```

callstack

```
(callstack )
```

Returns the current callstack.

```
(do
   (defn f1 [x] (f2 x))
   (defn f2 [x] (f3 x))
```

```
    (defn f3 [x] (f4 x))
    (defn f4 [x] (callstack))
    (f1 100))
=> [{:fn-name "callstack" :file "example" :line 5 :col 18} {:fn-name "user/f4" :file "example" :line 4 :col 18}
{:fn-name "user/f3" :file "example" :line 3 :col 18} {:fn-name "user/f2" :file "example" :line 2 :col 18} {:fn-
name "user/f1" :file "example" :line 6 :col 5}]
```

case

```
(case expr & clauses)
```

Takes an expression and a set of clauses. Each clause takes the form
of test-constant result-expr

```
(case (+ 1 9)
  10  :ten
  20  :twenty
  30  :thirty
  :dont-know)
=> :ten
```

cast

```
(cast class object)
```

Casts a Java object

```
(do
   (import :java.awt.image.BufferedImage)
   (import :java.awt.Graphics)

   ;; cast the graphics context to 'java.awt.Graphics' instead of the
   ;; implicit cast to 'java.awt.Graphics2D' as Venice is doing
   (let [img (. :BufferedImage :new 40 40 1)
         gd (cast :Graphics (. img :createGraphics))]
     (. gd :fillOval 10 20 5 5)
     img))
=> BufferedImage@13969fbe: type = 1 DirectColorModel: rmask=ff0000 gmask=ff00 bmask=ff amask=0
IntegerInterleavedRaster: width = 40 height = 40 #Bands = 3 xOff = 0 yOff = 0 dataOffset[0] 0
```

ceil

```
(ceil x)
```

Returns the largest integer that is greater than or equal to x

```
(ceil 1.4)
=> 2.0
```

```
(ceil -1.4)
```

```
=> -1.0

(ceil 1.23M)
=> 2.00M

(ceil -1.23M)
=> -1.00M
```

char

```
(char c)
```

Converts a number or s single char string to a char.

```
(char 65)
=> "A"

(char "A")
=> "A"

(str/join (map char [65 66 67 68]))
=> "ABCD"
```

char?

```
(char? s)
```

Returns true if s is a char.

```
(char? (char "x"))
=> true
```

charset-default-encoding

```
(charset-default-encoding)
```

Returns the default charset of this Java virtual machine.

```
(charset-default-encoding)
=> :UTF-8
```

cidr/in-range?

```
(cidr/in-range? ip cidr)
```

Returns true if the ip adress is within the ip range of the cidr else false. ip may be a string or a :java.net. InetAddress, cidr may be a string or a CIDR Java object obtained from 'cidr/parse'.

```
(cidr/in-range? "222.220.0.0" "222.220.0.0/11")
=> true

(cidr/in-range? (cidr/inet-addr "222.220.0.0") "222.220.0.0/11")
=> true

(cidr/in-range? "222.220.0.0" (cidr/parse "222.220.0.0/11"))
=> true
```

cidr/inet-addr

```
(cidr/inet-addr addr)
```

Converts an stringified IPv4 or IPv6 to a Java InetAddress.

```
(cidr/inet-addr "222.192.0.0")
=> /222.192.0.0

(cidr/inet-addr "2001:0db8:85a3:08d3:1319:8a2e:0370:7347")
=> /2001:db8:85a3:8d3:1319:8a2e:370:7347
```

cidr/parse

```
(cidr/parse cidr)
```

Parses CIDR IP blocks to an IP address range. Supports both IPv4 and IPv6.

```
(cidr/parse "222.192.0.0/11")
=> 222.192.0.0/11: [/222.192.0.0 .. /222.223.255.255]

(cidr/parse "2001:0db8:85a3:08d3:1319:8a2e:0370:7347/64")
=> 2001:0db8:85a3:08d3:1319:8a2e:0370:7347/64: [/2001:db8:85a3:8d3:0:0:0:0 .. /2001:db8:85a3:8d3:ffff:ffff:ffff:
ffff]
```

class

```
(class name)
```

Returns the Java class for the given name. Throws an exception if the class is not found.

```
(class :java.util.ArrayList)
=> class java.util.ArrayList
```

coalesce

```
(coalesce args*)
```

Returns null if all of its arguments are null, otherwise it returns
the first non nil argument. The arguments are evaluated lazy.

```
(coalesce)
=> nil

(coalesce 2)
=> 2

(coalesce nil 1 2)
=> 1
```

coll?

```
(coll? obj)
```

Returns true if obj is a collection

```
(coll? {:a 1})
=> true

(coll? [1 2])
=> true
```

comment

```
(comment & body)
```

Ignores body, yields nil

```
(comment
  (println 1)
  (println 5))
=> nil
```

comp

```
(comp f*)
```

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.

```
((comp str +) 8 8 8)
=> "24"
```

```
(map (comp - (partial + 3) (partial * 2)) [1 2 3 4])
=> (-5 -7 -9 -11)

((reduce comp [(partial + 1) (partial * 2) (partial + 3)]) 100)
=> 207

(filter (comp not zero?) [0 1 0 2 0 3 0 4])
=> [1 2 3 4]

(do
    (def fifth (comp first rest rest rest rest))
    (fifth [1 2 3 4 5]))
=> 5
```

compare

```
(compare x y)
```

Comparator. Returns -1, 0, or 1 when x is logically 'less than', 'equal to', or 'greater than' y. For list and vectors the longer sequence is always 'greater' regardless of its contents. For sets and maps only the size of the collection is compared.

```
(compare nil 0)
=> -1

(compare 0 nil)
=> 1

(compare 1 0)
=> 1

(compare 1 1)
=> 0

(compare 1M 2M)
=> -1

(compare 1 nil)
=> 1

(compare nil 1)
=> -1

(compare "aaa" "bbb")
=> -1

(compare [0 1 2] [0 1 2])
=> 0

(compare [0 1 2] [0 9 2])
=> -1

(compare [0 9 2] [0 1 2])
=> 1

(compare [1 2 3] [0 1 2 3])
=> -1

(compare [0 1 2] [3 4])
=> 1
```

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

```
(do
    (def counter (atom 2))
    (compare-and-set! counter 2 4)
    (deref counter))
=> 4
```

complement

```
(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.

```
(complement even?)
=> function anonymous-53e6bd9d-3ddd-42fb-b5dc-3e6dc8566dd4 {visibility :public, ns "core"}

(filter (complement even?) '(1 2 3 4))
=> (1 3)
```

concat

```
(concat coll)
(concat coll & colls)
```

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

```
(concat [1 2])
=> (1 2)

(concat [1 2] [4 5 6])
=> (1 2 4 5 6)

(concat '(1 2))
=> (1 2)

(concat '(1 2) [4 5 6])
=> (1 2 4 5 6)

(concat {:a 1})
=> ([:a 1])

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

```
=> ([:a 1] [:b 2] [:c 3])

(concat "abc")
=> ("a" "b" "c")

(concat "abc" "def")
=> ("a" "b" "c" "d" "e" "f")
```

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.

```
(let [n 5]
  (cond
    (< n 0) "negative"
    (> n 0) "positive"
    :else "zero"))
=> "positive"
```

condp

```
(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, a VncException
is thrown.

```
(condp some [1 2 3 4]
   #{0 6 7} :>> inc
   #{4 5 9} :>> dec
   #{1 2 3} :>> #(* % 10))
=> 3

(condp some [-10 -20 0 10]
  pos?  1
  neg? -1
  (constantly true)  0)
=> 1
```

conj

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

Returns a new collection with the x, xs 'added'. (conj nil item) returns (item). For list, vectors and ordered
maps the values are added at the end. For all other sets and maps the position is undefined.

```
(conj [1 2 3] 4)
=> [1 2 3 4]

(conj [1 2 3] 4 5)
=> [1 2 3 4 5]

(conj [1 2 3] [4 5])
=> [1 2 3 [4 5]]

(conj '(1 2 3) 4)
=> (1 2 3 4)

(conj '(1 2 3) 4 5)
=> (1 2 3 4 5)

(conj '(1 2 3) '(4 5))
=> (1 2 3 (4 5))

(conj (set 1 2 3) 4)
=> #{1 2 3 4}

(conj {:a 1 :b 2} [:c 3])
=> {:a 1 :b 2 :c 3}

(conj {:a 1 :b 2} {:c 3})
=> {:a 1 :b 2 :c 3}

(conj {:a 1 :b 2} (map-entry :c 3))
=> {:a 1 :b 2 :c 3}

(conj )
=> []

(conj 4)
=> 4
```

conj!

```
(conj!)
(conj! x)
(conj! coll x)
(conj! coll x & xs)
```

Returns a new mutable collection with the x, xs 'added'. (conj! nil item) returns (item). For mutable list the
values are added at the end. For all mutable sets and maps the position is undefined.

```
(conj! (mutable-list 1 2 3) 4)
=> (1 2 3 4)

(conj! (mutable-list 1 2 3) 4 5)
=> (1 2 3 4 5)

(conj! (mutable-list 1 2 3) '(4 5))
=> (1 2 3 (4 5))

(conj! (mutable-set 1 2 3) 4)
=> #{1 2 3 4}

(conj! (mutable-map :a 1 :b 2) [:c 3])
=> {:a 1 :b 2 :c 3}

(conj! (mutable-map :a 1 :b 2) {:c 3})
=> {:a 1 :b 2 :c 3}

(conj! (mutable-map :a 1 :b 2) (map-entry :c 3))
=> {:a 1 :b 2 :c 3}

(conj! )
=> ()

(conj! 4)
=> 4
```

cons

```
(cons x coll)
```

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

```
(cons 1 '(2 3 4 5 6))
=> (1 2 3 4 5 6)

(cons [1 2] [4 5 6])
=> [[1 2] 4 5 6]

(cons 3 (set 1 2))
=> #{1 2 3}

(cons {:c 3} {:a 1 :b 2})
=> {:a 1 :b 2 :c 3}

(cons (map-entry :c 3) {:a 1 :b 2})
=> {:a 1 :b 2 :c 3}
```

cons!

```
(cons! x coll)
```

Adds x to the mutable coll

```
(cons! 1 (mutable-list 2 3))
=> (1 2 3)

(cons! 3 (mutable-set 1 2))
=> #{1 2 3}

(cons! {:c 3} (mutable-map :a 1 :b 2))
=> {:a 1 :b 2 :c 3}

(cons! (map-entry :c 3) (mutable-map :a 1 :b 2))
=> {:a 1 :b 2 :c 3}
```

constantly

```
(constantly x)
```

Returns a function that takes any number of arguments and returns
always the value x.

```
(do
  (def fix (constantly 10))
  (fix 1 2 3)
  (fix 1)
  (fix ))
=> 10
```

contains?

```
(contains? coll key)
```

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

```
(contains? #{:a :b} :a)
=> true

(contains? {:a 1 :b 2} :a)
=> true

(contains? [10 11 12] 1)
=> true

(contains? [10 11 12] 5)
=> false

(contains? "abc" 1)
=> true

(contains? "abc" 5)
=> false
```

cos

```
(cos x)
```

```
cos x
```

```
(cos 1)
=> 0.5403023058681398

(cos 1.23)
=> 0.3342377271245026

(cos 1.23M)
=> 0.3342377271245026
```

count

```
(count coll)
```

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

```
(count {:a 1 :b 2})
=> 2

(count [1 2])
=> 2

(count "abc")
=> 3
```

crypt/decrypt

```
(crypt/decrypt algorithm passphrase & options)
```

Returns a new function to decrypt a string or a bytebuf given the
algorithm and passphrase. If a string is passed it is base64 decoded,
decrypted, and returned as string. If a bytebuf is passed the decrypted
bytebuf is returned.
Supported algorithms: "DES", "3DES", "AES256"

Options:
  :url-safe enabled
      The boolean option directs the base64 decoder to decode
      standard or URL safe base64 encoded strings.
      If enabled (true) the base64 decoder will convert '-' and '_'
      characters back to '+' and '/' before decoding.
      Defaults to false.

```
(do
  (load-module :crypt)
  (def decrypt (crypt/encrypt "3DES" "secret" :url-safe true))
  (decrypt "ndmW1NLsDHA") ; => "hello"
  (decrypt "KPYjndkZ8vM") ; => "world"
```

```
    (decrypt  (bytebuf [234 220 237 189 12 176 242 147])))

=> [192 19 255 241 144 162 159 77 53 176 196 254 163 194 211 219]
```

crypt/encrypt

```
(crypt/encrypt algorithm passphrase & options)
```

Returns a new function to encrypt a string or a bytebuf given
the algorithm and passphrase. If a string is passed it is
encrypted and returned as a base64 encoded string. If a bytebuf
is passed the encryped bytebuf is returned.
Supported algorithms: "DES", "3DES", "AES256"

```
Options:
  :url-safe enabled
      The boolean option directs the base64 encoder to emit
      standard or URL safe base64 enoded strings.
      If enabled (true) the base64 encoder will emit '-' and '_'
      instead of the usual '+' and '/' characters.
      Defaults to false.
      Note: no padding is added when encoding using the
            URL-safe alphabet.
```

```
(do
  (load-module :crypt)
  (def encrypt (crypt/encrypt "3DES" "secret" :url-safe true))
  (encrypt "hello") ; => "ndmW1NLsDHA"
  (encrypt "world") ; => "KPYjndkZ8vM"
  (encrypt (bytebuf [1 2 3 4 5])))

=> [234 220 237 189 12 176 242 147]
```

crypt/md5-hash

```
(crypt/md5-hash data)
```

Hashes a string or a bytebuf using MD5. Note: MD5 is not safe any more use PBKDF2.

```
(do
  (load-module :crypt)
  (str/bytebuf-to-hex
    (crypt/md5-hash "hello world")
    :upper))

=> "5EB63BBBE01EEED093CB22BB8F5ACDC3"
```

crypt/pbkdf2-hash

```
(crypt/pbkdf2-hash text salt)
(crypt/pbkdf2-hash text salt iterations key-length)
```

Hashes a string using PBKDF2. iterations default to 1000, key-length defaults to 256

```
(do
  (load-module :crypt)
  (str/bytebuf-to-hex
    (crypt/pbkdf2-hash "hello world" "-salt-")
    :upper))

=> "54F2B4411E8817C2A0743B2A7DD7EAE5AA3F748D1DDDCE00766380914AFFE995"

(do
  (load-module :crypt)
  (str/bytebuf-to-hex
    (crypt/pbkdf2-hash "hello world" "-salt-" 1000 256)
    :upper))
=> "54F2B4411E8817C2A0743B2A7DD7EAE5AA3F748D1DDDCE00766380914AFFE995"
```

crypt/sha512-hash

```
(crypt/sha512-hash data)
(crypt/sha512-hash data salt)
```

Hashes a string or a bytebuf using SHA512 with an optional salt.

```
(do
  (load-module :crypt)
  (str/bytebuf-to-hex
    (crypt/sha512-hash "hello world")
    :upper))

=>
"309ECC489C12D6EB4CC40F50C902F2B4D0ED77EE511A7C7A9BCD3CA86D4CD86F989DD35BC5FF499670DA34255B45B0CFD830E81F605DCF7DC55
```

csv/read

```
(csv/read source & options)
```

Reads CSV-data from a source. The source may be a a string, a bytebuf, a file, a Java InputStream, or a Java
Reader.

```
Options:
   :encoding enc - used when reading from a binary data source
                 e.g :encoding :utf-8, defaults to :utf-8
   :separator val - e.g. ",", defaults to a comma
       :quote val - e.g. "'", defaults to a double quote
```

```
(csv/read "1,\"ab\",false")
=> (("1" "ab" "false"))

(csv/read "1:::'ab':false" :separator ":" :quote "'")
=> (("1" nil nil "ab" "false"))
```

csv/write

```
(csv/write writer records & options)
```

```
Writes data to a writer in CSV format. The writer is a
Java java.io.Writer

Options:
   :separator val - e.g. ",", defaults to a comma
   :quote val - e.g. "'", defaults to a double quote
   :newline val (:lf (default) or :cr+lf)
```

```
(let [file (io/file "test.csv")
      fs (. :java.io.FileOutputStream :new file)]
  (try-with [writer (. :java.io.OutputStreamWriter :new fs "utf-8")]
    (csv/write writer [[1 "AC" false] [2 "WS" true]])))
```

csv/write-str

```
(csv/write-str records & options)
```

```
Writes data to a string in CSV format.

Options:
   :separator val - e.g. ",", defaults to a comma
   :quote val - e.g. "'", defaults to a double quote
   :newline val (:lf (default) or :cr+lf)
```

```
(csv/write-str [[1 "AC" false] [2 "WS" true]])
=> "1,AC,false\n2,WS,true"

(csv/write-str [[1 "AC" false] [2 "WS, '-1'" true]]
               :quote "'"
               :separator ","
               :newline :cr+lf)
=> "1,AC,false\r\n2,'WS, ''-1''',true"
```

current-time-millis

```
(current-time-millis)
```

```
Returns the current time in milliseconds.
```

```
(current-time-millis)
=> 1591172767265
```

dec

```
(dec x)
```

Decrements the number x

```
(dec 10)
=> 9

(dec 10I)
=> 9I

(dec 10.1)
=> 9.1

(dec 10.12M)
=> 9.12M
```

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/add 2.44697M 1.79882M 3 :HALF_UP)
=> 4.246M
```

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/div 2.44697M 1.79882M 5 :HALF_UP)
=> 1.36032M
```

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/mul 2.44697M 1.79882M 5 :HALF_UP)
=> 4.40166M
```

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/scale 2.44697M 0 :HALF_UP)
=> 2M

(dec/scale 2.44697M 1 :HALF_UP)
=> 2.4M

(dec/scale 2.44697M 2 :HALF_UP)
=> 2.45M

(dec/scale 2.44697M 3 :HALF_UP)
=> 2.447M

(dec/scale 2.44697M 10 :HALF_UP)
=> 2.4469700000M
```

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)

```
(dec/sub 2.44697M 1.79882M 3 :HALF_UP)
=> 0.648M
```

decimal

```
(decimal x) (decimal x scale rounding-mode)
```

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

```
(decimal 2)
=> 2M

(decimal 2 3 :HALF_UP)
=> 2.000M

(decimal 2.5787 3 :HALF_UP)
=> 2.579M

(decimal 2.5787M 3 :HALF_UP)
=> 2.579M

(decimal "2.5787" 3 :HALF_UP)
=> 2.579M
```

```
(decimal nil)
=> 0M
```

decimal?

```
(decimal? n)
```

Returns true if n is a decimal

```
(decimal? 4.0M)
=> true

(decimal? 4.0)
=> false

(decimal? 3)
=> false

(decimal? 3I)
=> false
```

dedupe

```
(dedupe coll)
```

Returns a collection with all consecutive duplicates removed. Returns a stateful transducer when no collection is provided.

```
(dedupe [1 2 2 2 3 4 4 2 3])
=> [1 2 3 4 2 3]

(dedupe '(1 2 2 2 3 4 4 2 3))
=> (1 2 3 4 2 3)
```

def

```
(def name expr)
```

Creates a global variable.

```
(def x 5)
=> user/x

(def sum (fn [x y] (+ x y)))
=> user/sum
```

def-dynamic

```
(def-dynamic name expr)
```

Creates a dynamic variable that starts off as a global variable and can be bound with 'binding' to a new value
on the local thread.

```
(do
   (def-dynamic x 100)
   (println x)
   (binding [x 200]
      (println x))
   (println x)))
100
200
100
=> nil
```

defmacro

```
(defmacro name [params*] body)
```

Macro definition

```
(defmacro unless [pred a b]
  `(if (not ~pred) ~a ~b))
=> macro user/unless {visibility :public, ns "user"}
```

defmethod

```
(defmethod multifn-name dispatch-val & fn-tail)
```

Creates a new method for a multimethod associated with a dispatch-value.

```
(do
   ;;defmulti with dispatch function
   (defmulti salary (fn[amount] (amount :t)))

   ;;defmethod provides a function implementation for a particular value
   (defmethod salary "com" [amount] (+ (:b amount) (/ (:b amount) 2)))
   (defmethod salary "bon" [amount] (+ (:b amount) 99))
   (defmethod salary :default  [amount] (:b amount))

   [(salary {:t "com" :b 1000})
    (salary {:t "bon" :b 1000})
    (salary {:t "xxx" :b 1000})]
)
=> [1500 1099 1000]
```

defmulti

```
(defmulti name dispatch-fn)
```

Creates a new multimethod with the associated dispatch function.

```
(do
   ;;defmulti with dispatch function
   (defmulti salary (fn[amount] (amount :t)))

   ;;defmethod provides a function implementation for a particular value
   (defmethod salary "com" [amount] (+ (:b amount) (/ (:b amount) 2)))
   (defmethod salary "bon" [amount] (+ (:b amount) 99))
   (defmethod salary :default  [amount] (:b amount))

   [(salary {:t "com" :b 1000})
    (salary {:t "bon" :b 1000})
    (salary {:t "xxx" :b 1000})]
)
=> [1500 1099 1000]
```

defn

```
(defn name [args*] condition-map? expr*)
(defn name ([args*] condition-map? expr*)+)
```

Same as (def name (fn name [args*] condition-map? expr*)) or
(def name (fn name ([args*] condition-map? expr*)+))

```
(defn sum [x y] (+ x y))
=> user/sum
```

```
(defn sum [x y] { :pre [(> x 0)] } (+ x y))
=> user/sum
```

defn-

```
(defn- name [args*] condition-map? expr*)
(defn- name ([args*] condition-map? expr*)+)
```

Same as defn, yielding non-public def

```
(defn- sum [x y] (+ x y))
=> user/sum
```

defonce

```
(defonce name expr)
```

Creates a global variable that can not be overwritten

```
(defonce x 5)
=> user/x
```

deftype

```
(deftype name fields)
(deftype name fields validator)
```

Defines a new custom type for the name with the fields.

```
(do
  (ns foo)
  (deftype :complex [real :long, imaginary :long])
  ; explicitly creating a custom type value
  (def x (.: :complex 100 200))
  ; Venice implicitly creates a builder function
  ; suffixed with a '.'
  (def y (complex. 200 300))
  ; ... and a type check function
  (complex? y)
  y)
=> #:foo/complex{:real 200 :imaginary 300}

(do
  (ns foo)
  (deftype :complex [real :long, imaginary :long])
  (def x (complex. 100 200))
  (type x))
=> :foo/complex

(do
  (ns foo)
  (deftype :complex
          [real :long, imaginary :long]
          (fn [t]
            (assert (pos? (:real t)) "real must be positive")
            (assert (pos? (:imaginary t)) "imaginary must be positive")))
  (def x (complex. 100 200))
  [(:real x) (:imaginary x)])
=> [100 200]

(do
  (ns foo)
  (deftype :named [name :string, value :any])
  (def x (named. "count" 200))
  (def y (named. "seq" [1 2]))
  [x y])
=> [#:foo/named{:name "count" :value 200} #:foo/named{:name "seq" :value [1 2]}]
```

deftype-of

```
(deftype-of name base-type)
```

```
(deftype-of name base-type validator)
```

Defines a new custom type wrapper based on a base type.

```
(do
  (ns foo)
  (deftype-of :email-address :string)
  ; explicitly creating a wrapper type value
  (def x (.: :email-address "foo@foo.org"))
  ; Venice implicitly creates a builder function
  ; suffixed with a '.'
  (def y (email-address. "foo@foo.org"))
  ; ... and a type check function
  (email-address? y)
  y)
=> "foo@foo.org"

(do
  (ns foo)
  (deftype-of :email-address :string)
  (str "Email: " (email-address. "foo@foo.org")))
=> "Email: foo@foo.org"

(do
  (ns foo)
  (deftype-of :email-address :string)
  (def x (email-address. "foo@foo.org"))
  [(type x) (supertype x)])
=> [:foo/email-address :core/string]

(do
  (ns foo)
  (deftype-of :email-address
              :string
              (fn [e]
                (assert (str/valid-email-addr? e)
                        "invalid email address")))
  (email-address. "foo@foo.org"))
=> "foo@foo.org"

(do
  (ns foo)
  (deftype-of :contract-id :long)
  (contract-id. 100000))
=> 100000

(do
  (ns foo)
  (deftype-of :my-long :long)
  (+ 10 (my-long. 100000)))
=> 100010
```

deftype-or

```
(deftype-or name val*)
```

Defines a new custom or type.

```
(do
  (ns foo)
  (deftype-or :color :red :green :blue)
  ; explicitly creating a wrapper type value
  (def x (.: :color :red))
  ; Venice implicitly creates a builder function
  ; suffixed with a '.'
  (def y (color. :red))
  ; ... and a type check function
  (color? y)
  y)
=> "red"

(do
  (ns foo)
  (deftype-or :digit 0 1 2 3 4 5 6 7 8 9)
  (digit. 1))
=> 1

(do
  (ns foo)
  (deftype-or :long-or-double :long :double)
  (long-or-double. 1000))
=> 1000
```

deftype?

```
(deftype? type)
```

Returns true if type is a custom type else false.

```
(do
  (ns foo)
  (deftype :complex [real :long, imaginary :long])
  (deftype? :complex))
=> true

(do
  (ns foo)
  (deftype-of :email-address :string)
  (deftype? :email-address))
=> true

(do
  (ns foo)
  (deftype :complex [real :long, imaginary :long])
  (def x (complex. 100 200))
  (deftype? (type x)))
=> true
```

delay

```
(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.
```

```
(do
   (def x (delay (println "working...") 100))
   (deref x))
working...
=> 100
```

delay?

```
(delay? x)
```

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

```
(do
   (def x (delay (println "working...") 100))
   (delay? x))
=> true
```

deliver

```
(deliver ref value)
```

```
Delivers the supplied value to the promise, releasing any pending derefs. A subsequent call to deliver on a
promise will have no effect.
```

```
(do
   (def p (promise))
   (deliver p 10)
   (deliver p 20)
   @p)
=> 10
```

deref

```
(deref x)
(deref x timeout-ms timeout-val)
```

```
Dereferences an atom, a future or a promise object. When applied to an atom, returns its current state. When
applied to a future, will block if computation not complete. The variant taking a timeout can be used for
futures and will return timeout-val if the timeout (in milliseconds) is reached before a value is available. If
a future is deref'd and the waiting thread is interrupted the futures are cancelled.
```

```
(do
   (def counter (atom 10))
   (deref counter))
=> 10
```

```
(do
   (def counter (atom 10))
   @counter)
=> 10

(do
   (defn task [] 100)
   (let [f (future task)]
      (deref f)))
=> 100

(do
   (defn task [] 100)
   (let [f (future task)]
      @f))
=> 100

(do
   (defn task [] 100)
   (let [f (future task)]
      (deref f 300 :timeout)))
=> 100

(do
   (def x (delay (println "working...") 100))
   @x)
working...
=> 100

(do
   (def p (promise))
   (deliver p 10)
   @p)
=> 10

(do
   (def x (agent 100))
   @x)
=> 100

(do
   (def counter (volatile 10))
   @counter)
=> 10
```

deref?

```
(deref? x)
```

Returns true if x is dereferencable.

```
(deref? (atom 10))
=> true

(deref? (delay 100))
=> true

(deref? (promise))
=> true
```

```
(deref? (future (fn [] 10)))
=> true

(deref? (volatile 100))
=> true

(deref? (agent 100))
=> true

(deref? (just 100))
=> true
```

difference

```
(difference s1)
(difference s1 s2)
(difference s1 s2 & sets)
```

Return a set that is the first set without elements of the remaining sets

```
(difference (set 1 2 3))
=> #{1 2 3}

(difference (set 1 2) (set 2 3))
=> #{1}

(difference (set 1 2) (set 1) (set 1 4) (set 3))
=> #{2}
```

disj

```
(disj set x)
(disj set x & xs)
```

Returns a new set with the x, xs removed.

```
(disj (set 1 2 3) 3)
=> #{1 2}
```

dissoc

```
(dissoc coll key)
(dissoc coll key & ks)
```

Returns a new coll of the same type, that does not contain a mapping for key(s)

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

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

dissoc!

```
(dissoc! coll key)
(dissoc! coll key & ks)
```

Dissociates keys from a mutable map, returns the map

```
(dissoc! (mutable-map :a 1 :b 2 :c 3) :b)
=> {:a 1 :c 3}

(dissoc! (mutable-map :a 1 :b 2 :c 3) :c :b)
=> {:a 1}
```

dissoc-in

```
(dissoc-in m ks)
```

Dissociates an entrye in a nested associative structure, where ks is a sequence of keys and returns a new nested structure.

```
(do
  (def users [ {:name "James" :age 26}
               {:name "John" :age 43} ] )
  (dissoc-in users [1]))
=> [{:age 26 :name "James"}]

(do
  (def users [ {:name "James" :age 26}
               {:name "John" :age 43} ] )
  (dissoc-in users [1 :age]))
=> [{:age 26 :name "James"} {:name "John"}]
```

distinct

```
(distinct coll)
```

Returns a collection with all duplicates removed. Returns a stateful transducer when no collection is provided.

```
(distinct [1 2 3 4 2 3 4])
=> [1 2 3 4]

(distinct '(1 2 3 4 2 3 4))
=> (1 2 3 4)
```

do

```
(do exprs)
```

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

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

dobench

```
(dobench count expr)
```

Runs the expr count times in the most effective way and returns a list of elapsed nanoseconds for each invocation. It's main purpose is supporting benchmark test.

```
(dobench 10 (+ 1 1))
=> (3426 1685 2105 1569 1494 2271 1421 1279 1095 1032)
```

doc

```
(doc name)
```

Prints documentation for a var or special form given its name

```
(doc +)
(+), (+ x), (+ x y), (+ x y & more)

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

Examples:
    (+)

    (+ 1)

    (+ 1 2)

    (+ 1 2 3 4)

    (+ 1I 2I)

    (+ 1 2.5)

    (+ 1 2.5M)

=> nil
```

docoll

```
(docoll f coll)
```

Applies f to the items of the collection presumably for side effects. Returns nil.

```
(docoll #(println %) [1 2 3 4])
1
2
3
4
=> nil

(docoll
    (fn [[k v]] (println (pr-str k v)))
    {:a 1 :b 2 :c 3 :d 4})
:a 1
:b 2
:c 3
:d 4
=> nil
```

dorun

```
(dorun count expr)
```

Runs the expr count times in the most effective way. It's main purpose is supporting benchmark test. Returns the expression result of the first invocation.

```
(dorun 10 (+ 1 1))
=> 2
```

dotimes

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

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

```
(dotimes [n 3] (println (str "n is " n)))
n is 0
n is 1
n is 2
=> nil
```

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.

```
(doto (. :java.util.HashMap :new)
      (. :put :a 1)
      (. :put :b 2))
=> {"a" 1 "b" 2}
```

double

```
(double x)
```

Converts to double

```
(double 1)
=> 1.0

(double nil)
=> 0.0

(double false)
=> 0.0

(double true)
=> 1.0

(double 1.2)
=> 1.2

(double 1.2M)
=> 1.2

(double "1.2")
=> 1.2
```

double-array

```
(double-array coll)
(double-array len)
(double-array len init-val)
```

Returns an array of Java primitive doubles containing the contents of coll or returns an array with the given
length and optional init value

```
(double-array '(1.0 2.0 3.0))
=> [1.0, 2.0, 3.0]

(double-array '(1I 2 3.2 3.56M))
=> [1.0, 2.0, 3.2, 3.56]

(double-array 10)
=> [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
(double-array 10 42.0)
=> [42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0]
```

double?

```
(double? n)
```

```
Returns true if n is a double
```

```
(double? 4.0)
=> true

(double? 3)
=> false

(double? 3I)
=> false

(double? 3.0M)
=> false

(double? true)
=> false

(double? nil)
=> false

(double? {})
=> false
```

drop

```
(drop n coll)
```

Returns a collection of all but the first n items in coll. Returns a stateful transducer when no collection is provided.

```
(drop 3 [1 2 3 4 5])
=> [4 5]

(drop 10 [1 2 3 4 5])
=> []
```

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. Returns a stateful transducer when no collection is provided.

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

empty

```
(empty coll)
```

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

```
(empty {:a 1})
=> {}

(empty [1 2])
=> []

(empty '(1 2))
=> ()
```

empty-to-nil

```
(empty-to-nil x)
```

Returns nil if x is empty

```
(empty-to-nil "")
=> nil

(empty-to-nil [])
=> nil

(empty-to-nil '())
=> nil

(empty-to-nil {})
=> nil
```

empty?

```
(empty? x)
```

Returns true if x is empty

```
(empty? {})
=> true

(empty? [])
=> true

(empty? '())
=> true
```

```
(empty? "")
=> true
```

entries

```
(entries m)
```

Returns a collection of the map entries.

```
(entries {:a 1 :b 2 :c 3})
=> ([:a 1] [:b 2] [:c 3])
```

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

(let [s "(+ 2 x)" x 10]
    (eval (read-string s))))
=> 12
```

even?

```
(even? n)
```

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

```
(even? 4)
=> true

(even? 3)
=> false

(even? (int 3))
=> false
```

every-pred

```
(every-pred p1 & p)
```

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.

```
((every-pred number?) 1)
=> true

((every-pred number?) 1 2)
=> true

((every-pred number? even?) 2 4 6)
=> true
```

every?

```
(every? pred coll)
```

Returns true if the predicate is true for all collection items, false otherwise.

```
(every? number? nil)
=> false

(every? number? [])
=> false

(every? number? [1 2 3 4])
=> true

(every? number? [1 2 3 :a])
=> false

(every? #(>= % 10) [10 11 12])
=> true
```

exists-class?

```
(exists-class? name)
```

Returns true the Java class for the given name exists otherwise returns false.

```
(exists-class? :java.util.ArrayList)
=> true
```

false?

```
(false? x)
```

Returns true if x is false, false otherwise

```
(false? true)
=> false

(false? false)
=> true

(false? nil)
=> false

(false? 0)
=> false

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

filter

```
(filter predicate coll)
```

Returns a collection of the items in coll for which (predicate item) returns logical true. Returns a transducer when no collection is provided.

```
(filter even? [1 2 3 4 5 6 7])
=> [2 4 6]
```

filter-k

```
(filter-k f map)
```

Returns a map with entries for which the predicate (f key) returns logical true. f is a function with one arguments.

```
(filter-k #(= % :a) {:a 1 :b 2 :c 3})
=> {:a 1}
```

filter-kv

```
(filter-kv f map)
```

Returns a map with entries for which the predicate (f key value) returns logical true. f is a function with two arguments.

```
(filter-kv (fn [k v] (= k :a)) {:a 1 :b 2 :c 3})
=> {:a 1}

(filter-kv (fn [k v] (= v 2)) {:a 1 :b 2 :c 3})
=> {:b 2}
```

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)
=> nil
```

first

```
(first coll)
```

Returns the first element of coll or nil if coll is nil or empty.

```
(first nil)
=> nil

(first [])
=> nil

(first [1 2 3])
=> 1

(first '())
=> nil

(first '(1 2 3))
=> 1

(first "abc")
=> "a"
```

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.Returns a transducer when no collection is provided.

```
(flatten [])
=> []

(flatten [[1 2 3] [4 [5 6]] [7 [8 [9]]]])
=> [1 2 3 4 5 6 7 8 9]

(flatten [1 2 {:a 3 :b [4 5 6]}])
=> [1 2 {:a 3 :b [4 5 6]}]
```

```
(flatten (seq {:a 1 :b 2}))
=> (:a 1 :b 2)
```

float-array

```
(float-array coll)
(float-array len)
(float-array len init-val)
```

Returns an array of Java primitive floats containing the contents of coll or returns an array with the given length and optional init value

```
(float-array '(1.0 2.0 3.0))
=> [1.0, 2.0, 3.0]

(float-array '(1I 2 3.2 3.56M))
=> [1.0, 2.0, 3.200000047683716, 3.559999942779541]

(float-array 10)
=> [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

(float-array 10 42.0)
=> [42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0, 42.0]
```

floor

```
(floor x)
```

Returns the largest integer that is less than or equal to x

```
(floor 1.4)
=> 1.0

(floor -1.4)
=> -2.0

(floor 1.23M)
=> 1.00M

(floor -1.23M)
=> -2.00M
```

flush

```
(flush)
(flush os)
```

Without arg flushes the output stream that is the current value
of *out*. With arg flushes the passed output stream. Returns nil.

```
(flush)
=> nil

(flush *out*)
=> nil

(flush *err*)
=> nil
```

fn

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

Defines an anonymous function.

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

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

(map #(* 2 %) (range 1 5))
=> (2 4 6 8)

(map #(* 2 %1) (range 1 5))
=> (2 4 6 8)

;; anonymous function with two params, the second is destructured
(reduce (fn [m [k v]] (assoc m v k)) {} {:b 2 :a 1 :c 3})
=> {1 :a 2 :b 3 :c}

;; defining a pre-condition
(do
   (def square-root
        (fn [x]
             { :pre [(>= x 0)] }
             (. :java.lang.Math :sqrt x)))
   (square-root 4))
=> 2.0

;; higher-order function
(do
   (def discount
        (fn [percentage]
             { :pre [(and (>= percentage 0) (<= percentage 100))] }
             (fn [price] (- price (* price percentage 0.01)))))
   ((discount 50) 300))
=> 150.0
```

fn?

```
(fn? x)
```

Returns true if x is a function

```
(do
    (def sum (fn [x] (+ 1 x)))
    (fn? sum))
=> true
```

fnil

```
(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.

```
((fnil + 10) nil)
=> 10

((fnil + 10) nil 1)
=> 11

((fnil + 10) nil 1 2)
=> 13

((fnil + 10) 20 1 2)
=> 23

((fnil + 10) nil 1 2 3 4)
=> 20

((fnil + 1000 100) nil nil)
=> 1100

((fnil + 1000 100) 2000 nil 1)
=> 2101

((fnil + 1000 100) nil 200 1 2)
=> 1203

((fnil + 1000 100) nil nil 1 2 3 4)
=> 1110
```

force

```
(force x)
```

If x is a Delay, returns its value, else returns x

```
(do
    (def x (delay (println "working...") 100))
    (force x))
working...
=> 100
```

formal-type

```
(formal-type object)
```

```
Returns the formal type of a Java object
```

```
(do
   (import :java.awt.image.BufferedImage)
   (import :java.awt.Graphics)

   ;; cast the graphics context to 'java.awt.Graphics' instead of the
   ;; implicit cast to 'java.awt.Graphics2D' as Venice is doing
   (let [img (. :BufferedImage :new 40 40 1)
         gd (cast :Graphics (. img :createGraphics))]
     (formal-type gd)))
=> :java.awt.Graphics
```

format-nano-time

```
(format-nano-time time)
(format-nano-time time & options)
```

```
Formats a time given in nanoseconds as long or double.

Options:
  :precision p - e.g :precision 4 (defaults to 3)
```

```
(format-nano-time 203)
=> "203 ns"

(format-nano-time 20389.0 :precision 2)
=> "20.39 µs"

(format-nano-time 20389 :precision 2)
=> "20.39 µs"

(format-nano-time 20389 :precision 0)
=> "20 µs"

(format-nano-time 203867669)
=> "203.868 ms"

(format-nano-time 20386766988 :precision 2)
=> "20.39 s"

(format-nano-time 20386766988 :precision 6)
=> "20.386767 s"
```

fourth

```
(fourth coll)
```

Returns the fourth element of coll.

```
(fourth nil)
=> nil

(fourth [])
=> nil

(fourth [1 2 3 4 5])
=> 4

(fourth '())
=> nil

(fourth '(1 2 3 4 5))
=> 4
```

frequencies

```
(frequencies coll)
```

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

```
(frequencies [:a :b :a :a])
=> {:a 3 :b 1}
```

future

```
(future fn)
```

Takes a function 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.
Thread local vars will be inherited by the future child thread. Changes of the child's thread local vars will not be seen on the parent.

```
(do
   (def wait (fn [] (do (sleep 500) 100)))
   (let [f (future wait)]
       (deref f)))
=> 100

;; demonstrates the use of thread locals with futures
(do
   ;; parent thread locals
   (binding [a 10 b 20]
      ;; future with child thread locals
      (let [f (future (fn [] (binding [b 90] {:a a :b b})))]
         {:child @f :parent {:a a :b b}})))
=> {:parent {:a 10 :b 20} :child {:a 10 :b 90}}
```

future-cancel

```
(future-cancel f)
```

Cancels the future

```
(do
   (def wait (fn [] (do (sleep 400) 100)))
   (let [f (future wait)]
      (sleep 50)
      (printf "After 50ms: cancelled=%b\n" (future-cancelled? f))
      (future-cancel f)
      (sleep 100)
      (printf "After 150ms: cancelled=%b\n" (future-cancelled? f))))
After 50ms: cancelled=false
After 150ms: cancelled=true
=> nil
```

future-cancelled?

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

Returns true if f is a Future is cancelled otherwise false

```
(future-cancelled? (future (fn [] 100)))
=> false
```

future-done?

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

Returns true if f is a Future is done otherwise false

```
(do
   (def wait (fn [] (do (sleep 200) 100)))
   (let [f (future wait)]
      (sleep 50)
      (printf "After 50ms: done=%b\n" (future-done? f))
      (sleep 300)
      (printf "After 300ms: done=%b\n" (future-done? f))))
After 50ms: done=false
After 300ms: done=true
=> nil
```

future?

```
(future? f)
```

```
Returns true if f is a Future otherwise false

(future? (future (fn [] 100)))
=> true
```

futures-fork

```
(futures-fork count worker-factory-fn)
```

Creates a list of count futures. The worker factory is single argument function that gets the worker index (0..
count-1) as argument and returns a worker function. Returns a list with the created futures.

```
(do
  (def mutex 0)
  (defn log [& xs]
    (locking mutex (println (apply str xs))))
  (defn factory [n]
    (fn [] (log "Worker" n)))
  (apply futures-wait (futures-fork 3 factory)))
Worker0
Worker1
Worker2
=> nil
```

futures-wait

```
(futures-wait & futures)
```

Waits for all futures to get terminated. If the waiting thread is interrupted the futures are cancelled.

```
(do
  (def mutex 0)
  (defn log [& xs]
    (locking mutex (println (apply str xs))))
  (defn factory [n]
    (fn [] (log "Worker" n)))
  (apply futures-wait (futures-fork 3 factory)))
Worker0
Worker1
Worker2
=> nil
```

gc

```
(gc)
```

Run the Java garbage collector. Runs the finalization methods of any objects pending finalization prior to the
GC.

```
(gc)
=> nil
```

gensym

```
(gensym)
(gensym prefix)
```

Generates a symbol.

```
(gensym )
=> G__20706

(gensym "prefix_")
=> prefix_20733
```

get

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

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

```
(get {:a 1 :b 2} :b)
=> 2

;; keywords act like functions on maps
(:b {:a 1 :b 2})
=> 2
```

get-in

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

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

```
(get-in {:a 1 :b {:c 2 :d 3}} [:b :c])
=> 2

(get-in [:a :b :c] [0])
=> :a

(get-in [:a :b [:c :d :e]] [2 1])
=> :d

(get-in {:a 1 :b {:c [4 5 6]}} [:b :c 1])
=> 5
```

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.

```
(group-by count ["a" "as" "asd" "aa" "asdf" "qwer"])
=> {1 ["a"] 2 ["as" "aa"] 3 ["asd"] 4 ["asdf" "qwer"]}

(group-by odd? (range 10))
=> {false [0 2 4 6 8] true [1 3 5 7 9]}
```

halt-when

```
(halt-when pred)
(halt-when pred retf)
```

Returns a transducer that ends transduction when pred returns true for an input. When retf is supplied it must be a fn of 2 arguments - it will be passed the (completed) result so far and the input that triggered the predicate, and its return value (if it does not throw an exception) will be the return value of the If retf is not supplied, the input that triggered the predicate will be returned. If the predicate never returns true the transduction is unaffected.

```
(do
  (def xf (comp (halt-when #(== % 10)) (filter odd?)))
  (transduce xf conj [1 2 3 4 5 6 7 8 9]))
=> [1 3 5 7 9]

(do
  (def xf (comp (halt-when #(> % 5)) (filter odd?)))
  (transduce xf conj [1 2 3 4 5 6 7 8 9]))
=> 6
```

hash-map

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

Creates a new hash map containing the items.

```
(hash-map :a 1 :b 2)
=> {:a 1 :b 2}

(hash-map (sorted-map :a 1 :b 2))
=> {:a 1 :b 2}
```

hash-map?

```
(hash-map? obj)
```

Returns true if obj is a hash map

```
(hash-map? (hash-map :a 1 :b 2))
=> true
```

host-address

```
(host-address)
```

Returns this host's ip address.

```
(host-address)
=> "127.0.0.1"
```

host-name

```
(host-name)
```

Returns this host's name.

```
(host-name)
=> "saturn.local"
```

identity

```
(identity x)
```

Returns its argument.

```
(identity 4)
=> 4

(filter identity [1 2 3 nil 4 false true 1234])
=> [1 2 3 4 true 1234]
```

if

```
(if test true-expr false-expr)
```

```
Evaluates test.

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

if-let

```
(if-let bindings then)
(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
```

```
(if-let [value (* 100 2)]
  (str "The expression is true. value=" value)
  (str "The expression is false."))
=> "The expression is true. value=200"
```

if-not

```
(if-not then)
(if-not then else)
```

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

```
(if-not (== 1 2) 100 0)
=> 100

(if-not (== 1 2) 100)
=> 100
```

import

```
(import class)
```

```
Imports a Java class. Imports are bound to the current namespace.
```

```
(do
  (import :java.lang.Math)
  (. :Math :max 2 10))
=> 10

(do
  (ns alpha)
  (import :java.lang.Math)
  (println "alpha:" (any? #(== % :java.lang.Math) (imports)))

  (ns beta)
```

```
    (println "beta:" (any? #(== % :java.lang.Math) (imports)))

    (ns alpha)
    (println "alpha:" (any? #(== % :java.lang.Math) (imports)))
)
alpha: true
beta: false
alpha: true
=> nil
```

imports

```
(imports)
```

List the registered imports for the current namespace.

```
(do
  (import :java.lang.Math)
  (imports))
=> (:com.github.jlangch.venice.VncException :com.github.jlangch.venice.impl.ValueException :java.lang.Exception
:java.lang.IllegalArgumentException :java.lang.Math :java.lang.NullPointerException :java.lang.RuntimeException
:java.lang.Throwable)
```

inc

```
(inc x)
```

Increments the number x

```
(inc 10)
=> 11

(inc 10I)
=> 11I

(inc 10.1)
=> 11.1

(inc 10.12M)
=> 11.12M
```

instance?

```
(instance? type x)
```

Returns true if x is an instance of the given type

```
(instance? :long 500)
=> true
```

```
(instance? :java.math.BigInteger 500)
=> false
```

int

```
(int x)
```

Converts to int

```
(int 1)
=> 1I

(int nil)
=> 0I

(int false)
=> 0I

(int true)
=> 1I

(int 1.2)
=> 1I

(int 1.2M)
=> 1I

(int "1")
=> 1I

(int (char "A"))
=> 65I
```

int-array

```
(int-array coll)
(int-array len)
(int-array len init-val)
```

Returns an array of Java primitive ints containing the contents of coll or returns an array with the given
length and optional init value

```
(int-array '(1I 2I 3I))
=> [1I, 2I, 3I]

(int-array '(1I 2 3.2 3.56M))
=> [1I, 2I, 3I, 3I]

(int-array 10)
=> [0I, 0I, 0I, 0I, 0I, 0I, 0I, 0I, 0I, 0I]

(int-array 10 42I)
=> [42I, 42I, 42I, 42I, 42I, 42I, 42I, 42I, 42I, 42I]
```

int?

```
(int? n)
```

Returns true if n is an int

```
(int? 4I)
=> true

(int? 4)
=> false

(int? 3.1)
=> false

(int? true)
=> false

(int? nil)
=> false

(int? {})
=> false
```

interleave

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

Returns a collection of the first item in each coll, then the second etc.

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

interpose

```
(interpose sep coll)
```

Returns a collection of the elements of coll separated by sep.

```
(interpose ", " [1 2 3])
=> (1 ", " 2 ", " 3)

(apply str (interpose ", " [1 2 3]))
=> "1, 2, 3"
```

intersection

```
(intersection s1)
(intersection s1 s2)
(intersection s1 s2 & sets)
```

Return a set that is the intersection of the input sets

```
(intersection (set 1))
=> #{1}

(intersection (set 1 2) (set 2 3))
=> #{2}

(intersection (set 1 2) (set 3 4))
=> #{}
```

into

```
(into)
(into to)
(into to from)
```

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

```
(into (sorted-map) [ [:a 1] [:c 3] [:b 2] ] )
=> {:a 1 :b 2 :c 3}

(into (sorted-map) [ {:a 1} {:c 3} {:b 2} ] )
=> {:a 1 :b 2 :c 3}

(into (sorted-map) [(map-entry :b 2) (map-entry :c 3) (map-entry :a 1)])
=> {:a 1 :b 2 :c 3}

(into (sorted-map) {:b 2 :c 3 :a 1})
=> {:a 1 :b 2 :c 3}

(into [] {1 2, 3 4})
=> [[1 2] [3 4]]

(into '() '(1 2 3))
=> (3 2 1)

(into [1 2 3] '(4 5 6))
=> [1 2 3 4 5 6]

(into '() (bytebuf [0 1 2]))
=> (0 1 2)

(into [] (bytebuf [0 1 2]))
=> [0 1 2]

(into '() "abc")
=> ("a" "b" "c")

(into [] "abc")
=> ["a" "b" "c"]

(do
   (into (. :java.util.concurrent.CopyOnWriteArrayList :new)
         (doto (. :java.util.ArrayList :new)
```

```
            (. :add 3)
            (. :add 4))))

=> (3 4)

(do
   (into (. :java.util.concurrent.CopyOnWriteArrayList :new)
       '(3 4)))

=> (3 4)
```

io/classpath-resource?

```
(io/classpath-resource? name)
```

Returns true if the classpath resource exists otherwise false.

```
(io/classpath-resource? "org/foo/images/foo.png")
```

io/copy-file

```
(io/copy-file source dest & options)
```

Copies source to dest. Returns nil or throws IOException. Source must be a file or a string (file path), dest must be a file, a string (file path), or an OutputStream.

```
Options:
  :replace true/false - e.g if true replace an aexistiong file, defaults to false
```

io/copy-stream

```
(io/copy-file in-stream out-stream)
```

Copies input stream to an output stream. Returns nil or throws IOException. Input and output must be a java.io.InputStream and java.io.OutputStream.

io/default-charset

```
(io/default-charset)
```

Returns the default charset.

io/delete-file

```
(io/delete-file f & files)
```

Deletes one or multiple files. Silently skips delete if the file does not exist. If f is a directory the
directory must be empty. f must be a file or a string (file path)

io/delete-file-on-exit

```
(io/delete-file-on-exit f)
```

Deletes a file on JVM exit. f must be a file or a string (file path).

io/delete-file-tree

```
(io/delete-file-tree f & files)
```

Deletes a file or a directory with all its content. Silently skips delete if the file or directory does not
exist. f must be a file or a string (file path)

io/download

```
(io/download uri & options)
```

```
Downloads the content from the uri and reads it as text (string) or binary (bytebuf).

Options:
  :binary true/false - e.g :binary true, defaults to false
  :user-agent agent  - e.g :user-agent "Mozilla", defaults to nil
  :encoding enc      - e.g :encoding :utf-8, defaults to :utf-8
  :conn-timeout val  - e.g :conn-timeout 10000,
                       connection timeout in milli seconds.
                       0 is interpreted as an infinite timeout.
  :read-timeout val  - e.g :read-timeout 10000,
                       read timeout in milli seconds.
                       0 is interpreted as an infinite timeout.
  :progress-fn fn    - a progress function that takes 2 args
                       [1] progress (0..100%)
                       [2] status {:start :progress :end :failed}

If the server returns a 403 (access denied) sending a user-agent
may fool the website.
```

io/exists-dir?

```
(io/exists-dir? f)
```

Returns true if the file f exists and is a directory. f must be a file or a string (file path).

```
(io/exists-dir? (io/file "/temp"))
=> false
```

io/exists-file?

```
(io/exists-file? f)
```

Returns true if the file f exists. f must be a file or a string (file path).

```
(io/exists-file? "/temp/test.txt")
=> false
```

io/file

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

Returns a java.io.File. path, parent, may be a file or a string (file path) child must be a string

```
(io/file "/temp/test.txt")
=> /temp/test.txt

(io/file "/temp" "test.txt")
=> /temp/test.txt

(io/file "/temp" "test" "test.txt")
=> /temp/test/test.txt

(io/file (. :java.io.File :new "/temp/test.txt"))
=> /temp/test.txt
```

io/file-absolute-path

```
(io/file-absolute-path f)
```

Returns the absolute path of the file f. f must be a file or a string (file path).

```
(io/file-absolute-path (io/file "/tmp/test/x.txt"))
=> "/tmp/test/x.txt"
```

io/file-can-execute?

```
(io/file-can-execute? f)
```

Returns true if the file or directory f exists and can be executed. f must be a file or a string (file path).

```
(io/file-can-execute? "/temp/test.txt")
```

io/file-can-read?

```
(io/file-can-read? f)
```

Returns true if the file or directory f exists and can be read. f must be a file or a string (file path).

```
(io/file-can-read? "/temp/test.txt")
```

io/file-can-write?

```
(io/file-can-write? f)
```

Returns true if the file or directory f exists and can be written. f must be a file or a string (file path).

```
(io/file-can-write? "/temp/test.txt")
```

io/file-canonical-path

```
(io/file-canonical-path f)
```

Returns the canonical path of the file f. f must be a file or a string (file path).

```
(io/file-canonical-path (io/file "/tmp/test/../x.txt"))
=> "/private/tmp/x.txt"
```

io/file-ext?

```
(io/file-ext? f ext)
```

Returns true if the file f hast the extension ext. f must be a file or a string (file path).

```
(io/file-ext? "/tmp/test/x.txt" "txt")
=> true
```

```
(io/file-ext? (io/file "/tmp/test/x.txt") ".txt")
=> true
```

io/file-hidden?

```
(io/file-hidden? f)
```

Returns true if the file or directory f exists and is hidden. f must be a file or a string (file path).

```
(io/file-hidden? "/temp/test.txt")
```

io/file-name

```
(io/file-name f)
```

Returns the name of the file f as a string. f must be a file or a string (file path).

```
(io/file-name (io/file "/tmp/test/x.txt"))
=> "x.txt"
```

io/file-parent

```
(io/file-parent f)
```

Returns the parent file of the file f. f must be a file or a string (file path).

```
(io/file-path (io/file-parent (io/file "/tmp/test/x.txt")))
=> "/tmp/test"
```

io/file-path

```
(io/file-path f)
```

Returns the path of the file f as a string. f must be a file or a string (file path).

```
(io/file-path (io/file "/tmp/test/x.txt"))
=> "/tmp/test/x.txt"
```

io/file-size

```
(io/file-size f)
```

Returns the size of the file f. f must be a file or a string (file path).

```
(io/file-size "/bin/sh")
=> 31440
```

io/file?

```
(io/file? x)
```

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

```
(io/file? (io/file "/temp/test.txt"))
=> true
```

io/gzip

```
(io/gzip f)
```

gzips f. f may be a file, a string (file path), a bytebuf or an InputStream. Returns a bytebuf.

```
(->> (io/gzip "a.txt")
     (io/spit "a.gz"))
```

```
(io/gzip (bytebuf-from-string "abcdef" :utf-8))
```

io/gzip-to-stream

```
(io/gzip f os)
```

gzips f to the OutputStream os. f may be a file, a string (file path), a bytebuf, or an InputStream.

```
(do
  (import :java.io.ByteArrayOutputStream)
  (try-with [os (. :ByteArrayOutputStream :new)]
     (-> (bytebuf-from-string "abcdef" :utf-8)
         (io/gzip-to-stream os))
     (-> (. os :toByteArray)
         (io/ungzip)
         (bytebuf-to-string :utf-8))))
=> "abcdef"
```

io/gzip?

```
(io/gzip? f)
```

Returns true if f is a gzipped file. f may be a file, a string (file path), a bytebuf, or an InputStream

```
(-> (io/gzip (bytebuf-from-string "abc" :utf-8))    (io/gzip?))
=> true
```

io/internet-avail?

```
(io/internet-avail?)
(internet-avail? url)
```

Checks if an internet connection is present for a given url. Defaults to URL http://www.google.com.

```
(io/internet-avail? "http://www.google.com")
```

io/list-file-tree

```
(io/list-file-tree dir filterFn?)
```

Lists all files in a directory tree. dir must be a file or a string (file path). filterFn is an optional filter that filters the files found. The filter gets a java.io.File as argument.

```
(io/list-file-tree /tmp)
(io/list-file-tree /tmp #(io/file-ext? % ".log"))
```

io/list-files

```
(io/list-files dir filterFn?)
```

Lists files in a directory. dir must be a file or a string (file path). filterFn is an optional filter that filters the files found. The filter gets a java.io.File as argument.

```
(io/list-files /tmp)
(io/list-files /tmp #(io/file-ext? % ".log"))
```

io/list-files-glob

```
(io/list-files-glob dir glob)
```

Lists all files in a directory that match the glob pattern. dir must be a file or a string (file path).

```
(io/list-files-glob "." "sample*.txt".
```

io/load-classpath-resource

```
(io/load-classpath-resource name)
```

Loads a classpath resource.

```
(io/load-classpath-resource "org/foo/images/foo.png"
```

io/mime-type

```
(io/mime-type file)
```

Returns the mime-type for the file if available else nil.

```
(io/mime-type "document.pdf")
=> "application/pdf"

(io/mime-type (io/file "document.pdf"))
=> "application/pdf"
```

io/mkdir

```
(io/mkdir dir)
```

Creates the directory. dir must be a file or a string (file path).

io/mkdirs

```
(io/mkdirs dir)
```

Creates the directory including any necessary but nonexistent parent directories. dir must be a file or a string (file path).

io/move-file

```
(io/move-file source target)
```

Moves source to target. Returns nil or throws IOException. Source and target must be a file or a string (file path).

io/slurp

```
(io/slurp f & options)
```

Reads the content of file f as text (string) or binary (bytebuf). f may be a file, a string file path, a Java InputStream, or a Java Reader.

```
Options:
  :binary true/false - e.g :binary true, defaults to false
  :encoding enc - e.g :encoding :utf-8, defaults to :utf-8
```

io/slurp-lines

```
(io/slurp-lines file & options)
```

Read all lines from f. f may be a file, a string file path, a Java InputStream, or a Java Reader.

```
Options:
  :encoding enc - e.g :encoding :utf-8, defaults to :utf-8
```

io/slurp-stream

```
(io/slurp-stream is & options)
```

Slurps binary or string data from a Java InputStream. Supports the option :binary to either slurp binary or string data. For string data an optional encoding can be specified.

```
Options:
  :binary true/false - e.g :binary true, defaults to false
  :encoding enc - e.g :encoding :utf-8, defaults to :utf-8

(do
   (import :java.io.FileInputStream)
   (let [file (io/temp-file "test-", ".txt")]
       (io/delete-file-on-exit file)
       (io/spit file "123456789" :append true)
       (try-with [is (. :FileInputStream :new file)]
           (io/slurp-stream is :binary false)))
)
=> "123456789"
```

io/spit

```
(io/spit f content & options)
```

Opens f, writes content, and then closes f. f may be a file or a string (file path). The content may be a
string or a bytebuf.

```
Options:
   :append true/false - e.g :append true, defaults to false
   :encoding enc - e.g :encoding :utf-8, defaults to :utf-8
```

io/spit-stream

```
(io/spit-stream os content & options)
```

Writes content (string or bytebuf) to the Java OutputStream os. If content is of type string an optional
encoding (defaults to UTF-8) is supported. The stream can optionally be flushed after the operation.

```
Options:
   :flush true/false - e.g :flush true, defaults to false
   :encoding enc - e.g :encoding :utf-8, defaults to :utf-8
```

```
(do
   (import :java.io.FileOutputStream)
   (let [file (io/temp-file "test-", ".txt")]
      (io/delete-file-on-exit file)
      (try-with [os (. :FileOutputStream :new file)]
         (io/spit-stream os "123456789" :flush true)))
)
=> nil
```

io/temp-file

```
(io/temp-file prefix suffix)
```

Creates an empty temp file with prefix and suffix.

```
(do
   (let [file (io/temp-file "test-", ".txt")]
      (io/spit file "123456789" :append true)
      (io/slurp file :binary false :remove true))
)
=> "123456789"
```

io/tmp-dir

```
(io/tmp-dir)
```

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

```
(io/tmp-dir )
=> /var/folders/rm/pjqr5pln3db4mxh5qq1j5yh80000gn/T
```

io/ungzip

```
(io/ungzip f)
```

ungzips f. f may be a file, a string (file path), a bytebuf, or an InputStream. Returns a bytebuf.

```
(-> (bytebuf-from-string "abcdef" :utf-8)
    (io/gzip)
    (io/ungzip))
=> [97 98 99 100 101 102]
```

io/ungzip-to-stream

```
(io/ungzip-to-stream buf)
```

ungzips a bytebuf returning an InputStream to read the deflated data from.

```
(-> (bytebuf-from-string "abcdef" :utf-8)
    (io/gzip)
    (io/ungzip-to-stream)
    (io/slurp-stream :binary false :encoding :utf-8))
=> "abcdef"
```

io/unzip

```
(io/unzip f entry-name)
```

Unzips an entry from zip f the entry's data as a bytebuf. f may be a bytebuf,
a file, a string (file path) or an InputStream.

```
(-> (io/zip "a.txt" (bytebuf-from-string "abcdef" :utf-8))
    (io/unzip "a.txt"))
=> [97 98 99 100 101 102]
```

io/unzip-all

```
(io/unzip-all f)
```

Unzips all entries of the zip f returning a map with the entry names as key and the entry data as bytebuf
values. f may be a bytebuf, a file, a string (file path) or an InputStream.

```
(-> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8)
           "b.txt" (bytebuf-from-string "def" :utf-8)
           "c.txt" (bytebuf-from-string "ghi" :utf-8))
    (io/unzip-all))
=> {"a.txt" [97 98 99] "b.txt" [100 101 102] "c.txt" [103 104 105]}
```

io/unzip-first

```
(io/unzip-first zip)
```

Unzips the first entry of the zip f returning its data as a bytebuf. f may be a bytebuf, a file, a string (file path) or an InputStream.

```
(-> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8)
           "b.txt" (bytebuf-from-string "def" :utf-8))
    (io/unzip-first))
=> [97 98 99]
```

io/unzip-nth

```
(io/unzip-nth zip n)
```

Unzips the nth (zero.based) entry of the zip f returning its data as a bytebuf. f may be a bytebuf, a file, a string (file path) or an InputStream.

```
(-> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8)
           "b.txt" (bytebuf-from-string "def" :utf-8)
           "c.txt" (bytebuf-from-string "ghi" :utf-8))
    (io/unzip-nth 1))
=> [100 101 102]
```

io/unzip-to-dir

```
(io/unzip-to-dir f dir)
```

Unzips f to a directory. f may be a file, a string (file path), a bytebuf, or an InputStream.

```
(-> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8)
           "b.txt" (bytebuf-from-string "def" :utf-8)
           "c.txt" (bytebuf-from-string "ghi" :utf-8))
    (io/unzip-to-dir "."))
```

io/uri-stream

```
(io/uri-stream uri)
```

```
Returns a Java InputStream from the uri.

(-> (io/uri-stream "https://www.w3schools.com/xml/books.xml")
    (io/slurp-stream :binary false :encoding :utf-8))
```

io/user-dir

```
(io/user-dir)
```

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

io/user-home-dir

```
(io/user-home-dir)
```

Returns the user's home dir as a java.io.File.

io/wrap-is-with-buffered-reader

```
(io/wrap-is-with-buffered-reader is encoding?)
```

Wraps an InputStream with a BufferedReader using an optional encoding (defaults to :utf-8).

```
(do
   (import :java.io.ByteArrayInputStream)
   (let [data (byte-array [108 105 110 101 32 49 10 108 105 110 101 32 50])
         is (. :ByteArrayInputStream :new data)
         rd (io/wrap-is-with-buffered-reader is :utf-8)]
     (println (. rd :readLine))
     (println (. rd :readLine))))
line 1
line 2
=> nil
```

io/wrap-os-with-buffered-writer

```
(io/wrap-os-with-buffered-writer os encoding?)
```

Wraps an OutputStream with a BufferedWriter using an optional encoding (defaults to :utf-8).

```
(do
   (import :java.io.ByteArrayOutputStream)
   (let [os (. :ByteArrayOutputStream :new)
```

```
            wr (io/wrap-os-with-buffered-writer os :utf-8)]
      (. wr :write "line 1")
      (. wr :newLine)
      (. wr :write "line 2")
      (. wr :flush)
      (. os :toByteArray)))
=> [108 105 110 101 32 49 10 108 105 110 101 32 50]
```

io/wrap-os-with-print-writer

```
(io/wrap-os-with-print-writer os encoding?)
```

Wraps an OutputStream with a PrintWriter using an optional encoding (defaults to :utf-8).

```
(do
   (import :java.io.ByteArrayOutputStream)
   (let [os (. :ByteArrayOutputStream :new)
         wr (io/wrap-os-with-print-writer os :utf-8)]
      (. wr :println "line 1")
      (. wr :println "line 2")
      (. wr :flush)
      (. os :toByteArray)))
=> [108 105 110 101 32 49 10 108 105 110 101 32 50 10]
```

io/zip

```
(io/zip & entries)
```

Creates a zip containing the entries. An entry is given by a name and data. The entry data may be nil, a bytebuf, a file, a string (file path), or an InputStream. An entry name with a trailing '/' creates a directory. Returns the zip as bytebuf.

```
; single entry
(->> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8))
     (io/spit "test.zip"))


; multiple entries
(->> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8)
             "b.txt" (bytebuf-from-string "def" :utf-8)
             "c.txt" (bytebuf-from-string "ghi" :utf-8))
     (io/spit "test.zip"))


; multiple entries with subdirectories
(->> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8)
             "x/b.txt" (bytebuf-from-string "def" :utf-8)
             "x/y/c.txt" (bytebuf-from-string "ghi" :utf-8))
     (io/spit "test.zip"))


; empty directory z/
(->> (io/zip "a.txt" (bytebuf-from-string "abc" :utf-8)
             "z/" nil)
     (io/spit "test.zip"))
```

io/zip-append

```
(io/zip-append f & entries)
```

Appends entries to an existing zip file f. Overwrites existing entries. An entry is given by a name and data. The entry data may be nil, a bytebuf, a file, a string (file path), or an InputStream.An entry name with a trailing '/' creates a directory.

```
(let [data (bytebuf-from-string "abc" :utf-8)]
    ; create the zip with a first file
    (->> (io/zip "a.txt" data)
         (io/spit "test.zip"))
    ; add text files
    (io/zip-append "test.zip" "b.txt" data "x/c.txt" data)
    ; add an empty directory
    (io/zip-append "test.zip" "x/y/" nil)))
```

io/zip-file

```
(io/zip-file options* zip-file & files)
```

Zips files. The zip-file my be a file, a string (file path) or an OutputStream.

```
Options:
  :filter-fn fn - filters the files to be added to the zip.
```

```
(io/zip-file "test.zip" "a.txt" "x/b.txt")
```

```
(io/zip-file "test.zip" "dir")
```

```
(io/zip-file :filter-fn (fn [dir name] (str/ends-with? name ".txt"))
             "test.zip"
             "test-dir")
```

io/zip-list

```
(io/zip-list f & options)
```

List the content of a the zip f. f may be a bytebuf, a file, a string (file path), or an InputStream.
Options:
  :verbose true/false - e.g :verbose true, defaults to false

```
(io/zip-list "test-file.zip")
```

```
(io/zip-list "test-file.zip" :verbose true)
```

io/zip-list-entry-names

```
(io/zip-list-entry-names)
```

Returns a list of the zip's entry names.

```
(io/zip-list-entry-names "test-file.zip")
```

io/zip-remove

```
(io/zip-remove f & entry-names)
```

Remove entries from a zip file f.

```
; remove files from zip
(io/zip-remove "test.zip" "x/a.txt" "x/b.txt")


; remove directory from zip
(io/zip-remove "test.zip" "x/y/")
```

io/zip?

```
(io/zip? f)
```

Returns true if f is a zipped file. f may be a file, a string (file path), a bytebuf, or an InputStream

```
(-> (io/zip "a" (bytebuf-from-string "abc" :utf-8))    (io/zip?))
=> true
```

java-enumeration-to-list

```
(java-enumeration-to-list e)
```

Converts a Java enumeration to a list

java-iterator-to-list

```
(java-iterator-to-list e)
```

Converts a Java iterator to a list

java-obj?

```
(java-obj? obj)
```

Returns true if obj is a Java object

```
(java-obj? (. :java.math.BigInteger :new "0"))
=> true
```

java-version

```
(java-version)
```

Returns the Java VM version.

```
(java-version)
=> "1.8.0_252"
```

java-version-info

```
(java-version-info)
```

Returns the Java VM version info.

```
(java-version-info)
=> {:version "1.8.0_252" :vendor "AdoptOpenJDK" :vm-version "25.252-b09" :vm-name "OpenJDK 64-Bit Server VM" :
vm-vendor "AdoptOpenJDK"}
```

json/pretty-print

```
(json/pretty-print s)
```

Pretty prints a JSON string

```
(json/pretty-print (json/write-str {:a 100 :b 100}))
=> "{\n  \"a\": 100,\n  \"b\": 100\n}"
```

json/read-str

```
(json/read-str s & options)
```

Reads a JSON string and returns it as a Venice datatype.

Options are:
  :key-fn fn
      Single-argument function called on JSON property names;
      return value will replace the property names in the output.
      Default is 'identity', use 'keyword' to get keyword
      properties.
  :value-fn fn
      Function to transform values in JSON objects in
      the output. For each JSON property, value-fn is called with
      two arguments: the property name (transformed by key-fn) and
      the value. The return value of value-fn will replace the value
      in the output. The default value-fn returns the value unchanged.
  :decimal boolean
      If true use BigDecimal for decimal numbers instead of Double.
      Default is false.

```
(json/read-str (json/write-str {:a 100 :b 100}))
=> {"a" 100 "b" 100}

(json/read-str (json/write-str {:a 100 :b 100}) :key-fn keyword)
=> {:a 100 :b 100}

(json/read-str (json/write-str {:a 100 :b 100})
                  :value-fn (fn [k v] (if (== "a" k) (inc v) v)))
=> {"a" 101 "b" 100}
```

json/slurp

```
(json/slurp in & options)
```

Slurps a JSON string from the input and returns it as a Venice datatype.
in maybe a file, a Java InputStream, or a Java Reader.

Options are:
  :key-fn fn
      Single-argument function called on JSON property names;
      return value will replace the property names in the output.
      Default is 'identity', use 'keyword' to get keyword
      properties.
  :value-fn fn
      Function to transform values in JSON objects in
      the output. For each JSON property, value-fn is called with
      two arguments: the property name (transformed by key-fn) and
      the value. The return value of value-fn will replace the value
      in the output. The default value-fn returns the value unchanged.
  :decimal boolean
      If true use BigDecimal for decimal numbers instead of Double.
      Default is false.
  :encoding enc - e.g :encoding :utf-8, defaults to :utf-8

```
(let [json (json/write-str {:a 100 :b 100})
      data (bytebuf-from-string json :utf-8)
      in (. :java.io.ByteArrayInputStream :new data)]
  (str (json/slurp in)))
=> "{a 100 b 100}"
```

json/spit

```
(json/spit out val & options)
```

```
Spits the JSON converted val to the output.
out maybe a file, a Java OutputStream, or a Java Writer.

Options are:
  :pretty boolean
      Enables/disables pretty printing.
      Defaults to false.
  :decimal-as-double boolean
      If true emit a decimal as double else as string.
      Defaults to false.
  :encoding enc - e.g :encoding :utf-8, defaults to :utf-8
```

```
(let [out (. :java.io.ByteArrayOutputStream :new)]
  (json/spit out {:a 100 :b 100 :c [10 20 30]})
  (. out :flush)
  (. :java.lang.String :new (. out :toByteArray) "utf-8"))
=> "{\"a\":100,\"b\":100,\"c\":[10,20,30]}"
```

json/write-str

```
(json/write-str val & options)
```

```
Writes the val to a JSON string.

Options are:
  :pretty boolean
      Enables/disables pretty printing.
      Defaults to false.
  :decimal-as-double boolean
      If true emit a decimal as double else as string.
      Defaults to false.
```

```
(json/write-str {:a 100 :b 100})
=> "{\"a\":100,\"b\":100}"

(json/write-str {:a 100 :b 100} :pretty true)
=> "{\n  \"a\": 100,\n  \"b\": 100\n}"
```

just

```
(just x)
```

```
Creates a wrapped x, that is dereferenceable
```

```
(just 10)
=> (just 10)
```

```
(just "10")
=> (just "10")

(deref (just 10))
=> 10
```

top

just?

```
(just? x)
```

Returns true if x is of type just

```
(just? (just 1))
=> true
```

top

juxt

```
(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)]

```
((juxt first last) '(1 2 3 4))
=> [1 4]

(do
  (defn index-by [coll key-fn]
    (into {} (map (juxt key-fn identity) coll)))

  (index-by [{:id 1 :name "foo"}
             {:id 2 :name "bar"}
             {:id 3 :name "baz"}]
            :id))
=> {1 {:name "foo" :id 1} 2 {:name "bar" :id 2} 3 {:name "baz" :id 3}}
```

top

keep

```
(keep f coll)
```

Returns a 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. Returns a transducer when no collection is provided.

```
(keep even? (range 1 4))
=> (false true false)
```

```
(keep (fn [x] (if (odd? x) x)) (range 4))
=> (1 3)

(keep #{3 5 7} '(1 3 5 7 9))
=> (3 5 7)
```

key

```
(key e)
```

Returns the key of the map entry.

```
(key (find {:a 1 :b 2} :b))
=> :b

(key (first (entries {:a 1 :b 2 :c 3})))
=> :a
```

keys

```
(keys map)
```

Returns a collection of the map's keys.

```
(keys {:a 1 :b 2 :c 3})
=> (:a :b :c)
```

keyword

```
(keyword name)
```

Returns a keyword from the given name

```
(keyword "a")
=> :a

(keyword :a)
=> :a
```

keyword?

```
(keyword? x)
```

Returns true if x is a keyword

```
(keyword? (keyword "a"))
=> true

(keyword? :a)
=> true

(keyword? nil)
=> false

(keyword? 'a)
=> false
```

last

```
(last coll)
```

Returns the last element of coll.

```
(last nil)
=> nil

(last [])
=> nil

(last [1 2 3])
=> 3

(last '())
=> nil

(last '(1 2 3))
=> 3

(last "abc")
=> "c"
```

let

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

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

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

;; destructured map
(let [{:keys [width height title ]
       :or {width 640 height 500}
       :as styles}
      {:width 1000 :title "Title"}]
    (println "width: " width)
    (println "height: " height)
    (println "title: " title)
    (println "styles: " styles))
width:  1000
```

```
height:  500
title:  Title
styles:  {:width 1000 :title Title}
=> nil
```

list

```
(list & items)
```

Creates a new list containing the items.

```
(list )
=> ()

(list 1 2 3)
=> (1 2 3)

(list 1 2 3 [:a :b])
=> (1 2 3 [:a :b])
```

list*

```
(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 collection.

```
(list* 1 [2 3])
=> (1 2 3)

(list* 1 2 3 [4])
=> (1 2 3 4)

(list* '(1 2) 3 [4])
=> ((1 2) 3 4)

(list* nil)
=> nil

(list* nil [2 3])
=> (nil 2 3)

(list* 1 2 nil)
=> (1 2)
```

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 predicate

```
(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])
=> (["abc" 0] ["abc" 1] ["abc" 2])
```

list?

```
(list? obj)
```

Returns true if obj is a list

```
(list? (list 1 2))
=> true

(list? '(1 2))
=> true
```

load-classpath-file

```
(load-classpath-file name)
(load-classpath-file name force)
```

Sequentially read and evaluate the set of forms contained in the
classpath file. The function is restricted to classpath files with
the extension '.venice'.

```
(do
  (load-classpath-file "com/github/jlangch/venice/test.venice")
  (test/test-fn "hello"))
=> "test: hello"

(do
  (load-classpath-file "com/github/jlangch/venice/test.venice")
  (test/test-fn "hello")
  ; reload the classpath file
  (ns-remove test)
```

```
  (load-classpath-file "com/github/jlangch/venice/test.venice" true)
  (test/test-fn "hello"))
=> "test: hello"
```

load-file

```
(load-file file)
(load-file file force)
```

```
Sequentially read and evaluate the set of forms contained in the file.
The global var *load-path* holds a list of paths that are tried to
load the file. If *load-path* is empty the file is loaded from the
current working directory.
```

```
(load-file "coffee")
```

```
(load-file "coffee.venice")
```

```
(load-file "beverages/coffee")
```

load-module

```
(load-module m)
(load-module m force)
```

```
Loads a Venice predefined extension module.
```

```
(load-module :math)
=> :math
```

```
(do
  (load-module :math)
  ; reload the module
  (ns-remove math)
  (load-module :math true))
=> :math
```

load-resource

```
(load-resource res)
```

```
Loads a resource from the application archive or the load-path.
Returns a bytebuffer or nil if the resource is not found in any
of the two locations.
```

load-string

```
(load-string s)
```

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

```
(do
   (load-string "(def x 1)")
   (+ x 2))
=> 3
```

locking

```
(locking x & exprs)
```

Executes exprs in an implicit do, while holding the monitor of x.
Will release the monitor of x in all circumstances.
Locking operates like the synchronized keyword in Java.

```
(do
   (def x 1)
   (locking x
      (println 100)
      (println 200)))
100
200
=> nil

;; Locks are reentrant
(do
   (def x 1)
   (locking x
      (locking x
         (println "in"))
      (println "out")))
in
out
=> nil
```

log

```
(log x)
```

log x

```
(log 10)
=> 2.302585092994046

(log 10.23)
=> 2.325324579963535
```

```
(log 10.23M)
=> 2.325324579963535
```

log10

```
(log10 x)
```

```
log10 x
```

```
(log10 10)
=> 1.0
```

```
(log10 10.23)
=> 1.0098756337121602
```

```
(log10 10.23M)
=> 1.0098756337121602
```

long

```
(long x)
```

```
Converts to long
```

```
(long 1)
=> 1
```

```
(long nil)
=> 0
```

```
(long false)
=> 0
```

```
(long true)
=> 1
```

```
(long 1.2)
=> 1
```

```
(long 1.2M)
=> 1
```

```
(long "1")
=> 1
```

```
(long (char "A"))
=> 65
```

long-array

```
(long-array coll)
(long-array len)
(long-array len init-val)
```

Returns an array of Java primitive longs containing the contents of coll or returns an array with the given length and optional init value

```
(long-array '(1 2 3))
=> [1, 2, 3]

(long-array '(1I 2 3.2 3.56M))
=> [1, 2, 3, 3]

(long-array 10)
=> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

(long-array 10 42)
=> [42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
```

long?

```
(long? n)
```

Returns true if n is a long

```
(long? 4)
=> true

(long? 4I)
=> false

(long? 3.1)
=> false

(long? true)
=> false

(long? nil)
=> false

(long? {})
=> false
```

loop

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

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

```
;; tail recursion
(loop [x 10]
   (when (> x 1)
      (println x)
      (recur (- x 2))))
10
```

```
8
6
4
2
=> nil

;; tail recursion
(do
    (defn sum [n]
          (loop [cnt n acc 0]
            (if (zero? cnt)
                acc
                (recur (dec cnt) (+ acc cnt)))))
    (sum 10000))
=> 50005000
```

macro?

```
(macro? x)
```

Returns true if x is a macro

```
(macro? and)
=> true
```

macroexpand

```
(macroexpand form)
```

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

To recursively expand all macros in a form use (macroexpand-all form).

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

macroexpand-all

```
(macroexpand-all form)
```

Recursively expands all macros in the form.

```
(macroexpand-all '(and true true))
=> (let [cond__19761__auto true] (if cond__19761__auto true cond__19761__auto))

(macroexpand-all '(and true (or true false) true))
=> (let [cond__19787__auto true] (if cond__19787__auto (let [cond__19787__auto (let [cond__19788__auto true]
(if cond__19788__auto cond__19788__auto false))] (if cond__19787__auto true cond__19787__auto))
cond__19787__auto))
```

```
(macroexpand-all '(let [n 5] (cond (< n 0) -1 (> n 0) 1 :else 0)))
=> (let [n 5] (if (< n 0) -1 (if (> n 0) 1 (if :else 0 nil))))
```

make-array

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

Returns an array of the given type and length

```
(str (make-array :long 5))
=> "[0, 0, 0, 0, 0]"

(str (make-array :java.lang.Long 5))
=> "[nil, nil, nil, nil, nil]"

(str (make-array :long 2 3))
=> "[[0 0 0], [0 0 0]]"

(aset (make-array :java.lang.Long 5) 3 9999)
=> [nil, nil, nil, 9999, nil]
```

map

```
(map f coll colls*)
```

Applys 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. Returns a transducer when no collection is provided.

```
(map inc [1 2 3 4])
=> (2 3 4 5)

(map + [1 2 3 4] [10 20 30 40])
=> (11 22 33 44)
```

map-entry?

```
(map-entry? m)
```

Returns true if m is a map entry

```
(map-entry? (first (entries {:a 1 :b 2})))
=> true
```

map-indexed

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

Retruns a collection of applying f to 0 and the first item of coll, followed by applying f to 1 and the second
item of coll, etc. until coll is exhausted. Returns a stateful transducer when no collection is provided.

```
(map-indexed (fn [idx val] [idx val]) [:a :b :c])
=> ([0 :a] [1 :b] [2 :c])

(map-indexed vector [:a :b :c])
=> ([0 :a] [1 :b] [2 :c])

(map-indexed vector "abcdef")
=> ([0 "a"] [1 "b"] [2 "c"] [3 "d"] [4 "e"] [5 "f"])

(map-indexed hash-map [:a :b :c])
=> ({0 :a} {1 :b} {2 :c})
```

map-invert

```
(map-invert m)
```

Returns the map with the vals mapped to the keys.

```
(map-invert {:a 1 :b 2 :c 3})
=> {1 :a 2 :b 3 :c}
```

map?

```
(map? obj)
```

Returns true if obj is a map

```
(map? {:a 1 :b 2})
=> true
```

mapcat

```
(mapcat fn & colls)
```

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

```
(mapcat reverse [[3 2 1 0] [6 5 4] [9 8 7]])
=> (0 1 2 3 4 5 6 7 8 9)

(mapcat list [:a :b :c] [1 2 3])
=> (:a 1 :b 2 :c 3)

(mapcat #(remove even? %) [[1 2] [2 2] [2 3]])
```

```
=> (1 3)

(mapcat #(repeat 2 %) [1 2])
=> (1 1 2 2)

(mapcat (juxt inc dec)  [1 2 3 4])
=> (2 0 3 1 4 2 5 3)
```

mapv

```
(mapv f coll colls*)
```

Returns a vector 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.

```
(mapv inc [1 2 3 4])
=> [2 3 4 5]

(mapv + [1 2 3 4] [10 20 30 40])
=> [11 22 33 44]
```

match?

```
(match? s regex)
```

Returns true if the string s matches the regular expression regex

```
(match? "1234" "[0-9]+")
=> true

(match? "1234ss" "[0-9]+")
=> false
```

max

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

Returns the greatest of the values

```
(max 1)
=> 1

(max 1 2)
=> 2

(max 4 3 2 1)
=> 4
```

```
(max 1I 2I)
=> 2I

(max 1.0)
=> 1.0

(max 1.0 2.0)
=> 2.0

(max 4.0 3.0 2.0 1.0)
=> 4.0

(max 1.0M)
=> 1.0M

(max 1.0M 2.0M)
=> 2.0M

(max 4.0M 3.0M 2.0M 1.0M)
=> 4.0M

(max 1.0M 2)
=> 2
```

mean

```
(mean x)
(mean x y)
(mean x y & more)
```

Returns the mean value of the values

```
(mean 10 20 30)
=> 20.0

(mean 1.4 3.6)
=> 2.5

(mean 2.8M 6.4M)
=> 4.6000000000000000M
```

median

```
(median coll)
```

Returns the median of the values

```
(median '(3 1 2))
=> 2.0

(median '(3 2 1 4))
=> 2.5

(median '(3.6 1.4 4.8))
```

```
=> 3.6

(median '(3.6M 1.4M 4.8M))
=> 3.6M
```

memoize

```
(memoize f)
```

Returns a memoized version of a referentially transparent function.

```
(do
  (def fibonacci
    (memoize
      (fn [n]
        (cond
          (<= n 0) 0
          (< n 2) 1
          :else (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))))

  (time (fibonacci 25)))
Elapsed time: 7.02 ms
=> 75025
```

merge

```
(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.

```
(merge {:a 1 :b 2 :c 3} {:b 9 :d 4})
=> {:a 1 :b 9 :c 3 :d 4}

(merge {:a 1} nil)
=> {:a 1}

(merge nil {:a 1})
=> {:a 1}

(merge nil nil)
=> nil
```

merge-with

```
(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).

```
(merge-with + {:a 1 :b 2} {:a 9 :b 98 :c 0})
=> {:a 10 :b 100 :c 0}

(merge-with into {:a [1] :b [2]} {:b [3 4] :c [5 6]})
=> {:a [1] :b [2 3 4] :c [5 6]}
```

meta

```
(meta obj)
```

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

```
(meta (vary-meta [1 2] assoc :a 1))
=> {:a 1 :line 1 :column 26 :file "example"}
```

min

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

Returns the smallest of the values

```
(min 1)
=> 1

(min 1 2)
=> 1

(min 4 3 2 1)
=> 1

(min 1I 2I)
=> 1I

(min 1.0)
=> 1.0

(min 1.0 2.0)
=> 1.0

(min 4.0 3.0 2.0 1.0)
=> 1.0

(min 1.0M)
=> 1.0M

(min 1.0M 2.0M)
=> 1.0M
```

```
(min 4.0M 3.0M 2.0M 1.0M)
=> 1.0M

(min 1.0M 2)
=> 1.0M
```

mod

```
(mod n d)
```

Modulus of n and d.

```
(mod 10 4)
=> 2

(mod -1 5)
=> 4

(mod 10I 4I)
=> 2I
```

mutable-list

```
(mutable-list & items)
```

Creates a new mutable threadsafe list containing the items.

```
(mutable-list )
=> ()

(mutable-list 1 2 3)
=> (1 2 3)

(mutable-list 1 2 3 [:a :b])
=> (1 2 3 [:a :b])
```

mutable-list?

```
(mutable-list? obj)
```

Returns true if obj is a mutable list

```
(mutable-list? (mutable-list 1 2))
=> true
```

mutable-map

```
(mutable-map & keyvals)
(mutable-map map)
```

Creates a new mutable threadsafe map containing the items.

```
(mutable-map :a 1 :b 2)
=> {:a 1 :b 2}

(mutable-map (hash-map :a 1 :b 2))
=> {:a 1 :b 2}
```

mutable-map?

```
(mutable-map? obj)
```

Returns true if obj is a mutable map

```
(mutable-map? (mutable-map :a 1 :b 2))
=> true
```

mutable-set

```
(mutable-set & items)
```

Creates a new mutable set containing the items.

```
(mutable-set )
=> #{}

(mutable-set nil)
=> #{nil}

(mutable-set 1)
=> #{1}

(mutable-set 1 2 3)
=> #{1 2 3}

(mutable-set [1 2] 3)
=> #{[1 2] 3}
```

mutable-set?

```
(mutable-set? obj)
```

Returns true if obj is a mutable-set

```
(mutable-set? (mutable-set 1))
=> true
```

name

```
(name x)
```

Returns the name String of a string, symbol, keyword, or function/macro.

```
(name :x)
=> "x"

(name 'x)
=> "x"

(name "x")
=> "x"
```

nano-time

```
(nano-time)
```

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

```
(nano-time)
=> 200745922981560
```

neg?

```
(neg? x)
```

Returns true if x smaller than zero else false

```
(neg? -3)
=> true

(neg? 3)
=> false

(neg? (int -3))
=> true

(neg? -3.2)
=> true

(neg? -3.2M)
=> true
```

negate

```
(negate x)
```

Negates x

```
(negate 10)
=> -10

(negate 10I)
=> -10I

(negate 1.23)
=> -1.23

(negate 1.23M)
=> -1.23M
```

newline

```
(newline)
(newline os)
```

Without arg writes a platform-specific newline to the output stream
that is the current value of *out*. With arg writes a newline to the
passed output stream. Returns nil.

```
(newline)

=> nil

(newline *out*)

=> nil

(newline *err*)

=> nil
```

nfirst

```
(nfirst coll n)
```

Returns a collection of the first n items

```
(nfirst nil 2)
=> ()

(nfirst [] 2)
=> []

(nfirst [1] 2)
=> [1]

(nfirst [1 2 3] 2)
=> [1 2]
```

```
(nfirst '() 2)
=> ()

(nfirst '(1) 2)
=> (1)

(nfirst '(1 2 3) 2)
=> (1 2)

(nfirst "abcdef" 2)
=> "ab"
```

nil?

```
(nil? x)
```

Returns true if x is nil, false otherwise

```
(nil? nil)
=> true

(nil? 0)
=> false

(nil? false)
=> false
```

nlast

```
(nlast coll n)
```

Returns a collection of the last n items

```
(nlast nil 2)
=> ()

(nlast [] 2)
=> []

(nlast [1] 2)
=> [1]

(nlast [1 2 3] 2)
=> [2 3]

(nlast '() 2)
=> ()

(nlast '(1) 2)
=> (1)

(nlast '(1 2 3) 2)
=> (2 3)
```

```
(nlast "abcdef" 2)
=> "ef"
```

not

```
(not x)
```

Returns true if x is logical false, false otherwise.

```
(not true)
=> false

(not (== 1 2))
=> true
```

not-any?

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

Returns false if the predicate is true for at least one collection item, true otherwise

```
(not-any? number? nil)
=> true

(not-any? number? [])
=> true

(not-any? number? [1 :a :b])
=> false

(not-any? number? [1 2 3])
=> false

(not-any? #(>= % 10) [1 5 10])
=> false
```

not-contains?

```
(not-contains? coll key)
```

Returns true if key is not present in the given collection, otherwise returns false.

```
(not-contains? #{:a :b} :c)
=> true

(not-contains? {:a 1 :b 2} :c)
=> true

(not-contains? [10 11 12] 1)
=> false
```

```
(not-contains? [10 11 12] 5)
=> true

(not-contains? "abc" 1)
=> false

(not-contains? "abc" 5)
=> true
```

not-empty?

```
(not-empty? x)
```

```
Returns true if x is not empty
```

```
(not-empty? {:a 1})
=> true

(not-empty? [1 2])
=> true

(not-empty? '(1 2))
=> true
```

not-every?

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

```
Returns false if the predicate is true for all collection items, true otherwise
```

```
(not-every? number? nil)
=> true

(not-every? number? [])
=> true

(not-every? number? [1 2 3 4])
=> false

(not-every? number? [1 2 3 :a])
=> true

(not-every? #(>= % 10) [10 11 12])
=> false
```

not-match?

```
(not-match? s regex)
```

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

```
(not-match? "1234" "[0-9]+")
=> false

(not-match? "1234ss" "[0-9]+")
=> true
```

ns-remove

```
(ns-remove ns)
```

Removes the mappings for all symbols from the namespace.

```
(do
  (ns xxx)
  (def foo 1)
  (def goo 1)
  (ns-remove xxx foo)
  (ns-remove *ns* foo))
=> nil
```

ns-unmap

```
(ns-unmap ns sym)
```

Removes the mappings for the symbol from the namespace.

```
(do
  (ns xxx)
  (def foo 1)
  (ns-unmap xxx foo)
  (ns-unmap *ns* foo))
=> nil
```

nth

```
(nth coll idx)
```

Returns the nth element of coll.

```
(nth nil 1)
=> nil

(nth [1 2 3] 1)
=> 2

(nth '(1 2 3) 1)
=> 2
```

```
(nth "abc" 2)
=> "c"
```

number?

```
(number? n)
```

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

```
(number? 4I))
=> true

(number? 4)
=> true

(number? 4.0M)
=> true

(number? 4.0)
=> true

(number? true)
=> false

(number? "a")
=> false
```

object-array

```
(object-array coll)
(object-array len)
(object-array len init-val)
```

Returns an array of Java Objects containing the contents of coll or returns an array with the given length and optional init value

```
(object-array '(1 2 3 4 5))
=> [1, 2, 3, 4, 5]

(object-array '(1 2.0 3.45M "4" true))
=> [1, 2.0, 3.45M, 4, true]

(object-array 10)
=> [nil, nil, nil, nil, nil, nil, nil, nil, nil, nil]

(object-array 10 42)
=> [42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
```

odd?

```
(odd? n)
```

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

```
(odd? 3)
=> true

(odd? 4)
=> false

(odd? (int 4))
=> false
```

offer!

```
(offer! queue v)
(offer! queue timeout v)
```

Offers an item to a queue with an optional timeout in milliseconds.

```
(let [s (queue)]
   (offer! s 4)
   (offer! s 3)
   (poll! s)
   s)
=> (3)
```

or

```
(or x)
(or x & next)
```

Ors the predicate forms

```
(or true false)
=> true

(or false false)
=> false
```

ordered-map

```
(ordered-map & keyvals)
(ordered-map map)
```

Creates a new ordered map containing the items.

```
(ordered-map :a 1 :b 2)
=> {:a 1 :b 2}
```

```
(ordered-map (hash-map :a 1 :b 2))
=> {:a 1 :b 2}
```

ordered-map?

```
(ordered-map? obj)
```

```
Returns true if obj is an ordered map
```

```
(ordered-map? (ordered-map :a 1 :b 2))
=> true
```

os-arch

```
(os-arch)
```

```
Returns the OS architecture
```

```
(os-arch)
=> "x86_64"
```

os-name

```
(os-name)
```

```
Returns the OS name
```

```
(os-name)
=> "Mac OS X"
```

os-type

```
(os-type)
```

```
Returns the OS type
```

```
(os-type)
=> :mac-osx
```

os-type?

```
(os-type? type)
```

Returns true if the OS id of the type otherwise false. Type is one of :windows, :mac-osx, or :linux

```
(os-type? :mac-osx)
=> true

(os-type? :windows)
=> false
```

os-version

```
(os-version)
```

Returns the OS version

```
(os-version)
=> "10.15.5"
```

partial

```
(partial f args*)
```

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.

```
((partial * 2) 3)
=> 6

(map (partial * 2) [1 2 3 4])
=> (2 4 6 8)

(do
   (def hundred-times (partial * 100))
   (hundred-times 5))
=> 500
```

partition

```
(partition n coll)
(partition n step coll)
(partition n step padcoll coll)
```

Returns a collection 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 padcoll 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.

```
(partition 4 (range 20))
=> ((0 1 2 3) (4 5 6 7) (8 9 10 11) (12 13 14 15) (16 17 18 19))
```

```
(partition 4 6 (range 20))
=> ((0 1 2 3) (6 7 8 9) (12 13 14 15) (18 19))

(partition 3 6 ["a"] (range 20))
=> ((0 1 2) (6 7 8) (12 13 14) (18 19 "a"))

(partition 4 6 ["a" "b" "c" "d"] (range 20))
=> ((0 1 2 3) (6 7 8 9) (12 13 14 15) (18 19 "a" "b"))
```

pdf/available?

```
(pdf/available?)
```

Checks if the 3rd party libraries required for generating PDFs are available.

```
(pdf/available?)
```

pdf/check-required-libs

```
(pdf/check-required-libs)
```

Checks if the 3rd party libraries required for generating PDFs are available. Throws an exception if not.

```
(pdf/check-required-libs)
```

pdf/copy

```
pdf/copy pdf & page-nr
```

Copies pages from a PDF to a new PDF. The PDF is passed a
as bytebuf. Returns the new PDF as a bytebuf.

```
; copy the first and second page
(pdf/copy pdf :1 :2)
```

```
; copy the last and second last page
(pdf/copy pdf :-1 :-2)
```

```
; copy the pages 1, 2, 6-10, and 12
(pdf/copy pdf :1 :2 :6-10 :12)
```

pdf/merge

```
pdf/merge pdfs
```

Merge multiple PDFs into a single PDF. The PDFs are passed
as bytebuf. Returns the new PDF as a bytebuf.

```
(pdf/merge pdf1 pdf2)
```

```
(pdf/merge pdf1 pdf2 pdf3)
```

pdf/pages

```
pdf/pages pdf
```

Returns the number of pages of a PDF. The PDF is passed a bytebuf.

```
(->> (str/lorem-ipsum :paragraphs 30)
     (pdf/text-to-pdf)
     (pdf/pages))
=> 3
```

pdf/render

```
(pdf/render xhtml & options)
```

Renders a PDF.

```
Options:
  :base-url url       - a base url. E.g.: "classpath:/"
  :resources resmap   - a resource map for dynamic resources
```

```
(pdf/render xhtm :base-url "classpath:/")
```

```
(pdf/render xhtm
            :base-url "classpath:/"
            :resources {"/chart_1.png" (chart-create :2018)
                        "/chart_2.png" (chart-create :2019) })
```

pdf/text-to-pdf

```
pdf/text-to-pdf text & options
```

Creates a PDF from simple text. The tool process line-feeds '\n' and form-feeds. To start a new page just
insert a form-feed marker "<form-feed>".

```
Options:
  :font-size n      - font size in pt (double), defaults to 9.0
  :font-weight n    - font weight (0...1000) (long), defaults to 200
  :font-monospace b - monospaced font (true/false) (boolean), defaults to false
```

```
(->> (pdf/text-to-pdf "Lorem Ipsum...")
     (io/spit "text.pdf"))
```

pdf/watermark

```
(pdf/watermark pdf options-map)
(pdf/watermark pdf & options)
```

Adds a watermark text to the pages of a PDF. The passed PDF pdf is a bytebuf. Returns the new PDF as a bytebuf.

```
Options:
  :text s             - watermark text (string), defaults to "WATERMARK"
  :font-size n        - font size in pt (double), defaults to 24.0
  :font-char-spacing n - font character spacing (double), defaults to 0.0
  :color s            - font color (HTML color string), defaults to #000000
  :opacity n          - opacity 0.0 ... 1.0 (double), defaults to 0.4
  :outline-color s    - font outline color (HTML color string), defaults to #000000
  :outline-opacity n  - outline opacity 0.0 ... 1.0 (double), defaults to 0.8
  :outline-witdh n    - outline width  0.0 ... 10.0 (double), defaults to 0.5
  :angle n            - angle 0.0 ... 360.0 (double), defaults to 45.0
  :over-content b     - print text over the content (boolean), defaults to true
  :skip-top-pages n   - the number of top pages to skip (long), defaults to 0
  :skip-bottom-pages n - the number of bottom pages to skip (long), defaults to 0
```

```
(pdf/watermark pdf :text "CONFIDENTIAL" :font-size 64 :font-char-spacing 10.0)
```

```
(let [watermark { :text "CONFIDENTIAL"
                  :font-size 64
                  :font-char-spacing 10.0 } ]
   (pdf/watermark pdf watermark))
```

peek

```
(peek coll)
```

For a list, same as first, for a vector, same as last, for a stack the top element

```
(peek '(1 2 3 4))
=> 1

(peek [1 2 3 4])
=> 4

(let [s (stack)]
   (push! s 4)
   (peek s))
=> 4
```

perf
```

```
(perf expr warmup-iterations test-iterations)
```

Performance test with the given expression.

Runs the test in 3 phases:
1. Runs the expr in a warmup phase to allow the HotSpot compiler
   to do optimizations.
2. Runs the garbage collector.
3. Runs the expression under profiling. Returns nil.

After a test run metrics data can be obtained with (prof :data-formatted)

```
(do
  (perf (+ 120 200) 12000 1000)
  (println (prof :data-formatted)))
```

pid

```
(pid)
```

Returns the PID of this process.

```
(pid)
=> "33531"
```

poll!

```
(poll! queue)
(poll! queue timeout)
```

Polls an item from a queue with an optional timeout in milliseconds. If not no timeout is given returns the item if one is available else returns nil. With a timeout returns the item if one is available withinthe given timeout else returns nil.

```
(let [s (queue)]
  (offer! s 4)
  (offer! s 3)
  (poll! s)
  s)
=> (3)
```

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 '(1 2 3 4))
=> (2 3 4)
```

```
(pop [1 2 3 4])
=> [1 2 3]
```

pop!

```
(pop! stack)
```

Pops an item from a stack.

```
(let [s (stack)]
    (push! s 4)
    (push! s 3)
    (pop! s)
    s)
=> (4)
```

pos?

```
(pos? x)
```

Returns true if x greater than zero else false

```
(pos? 3)
=> true

(pos? -3)
=> false

(pos? (int 3))
=> true

(pos? 3.2)
=> true

(pos? 3.2M)
=> true
```

postwalk

```
(postwalk f form)
```

Performs a depth-first, post-order traversal of form. Calls f on
each sub-form, uses f's return value in place of the original.

```
(postwalk (fn [x] (println "Walked:" (pr-str x)) x)
          '(1 2 {:a 1 :b 2}))
Walked: 1
Walked: 2
Walked: :a
Walked: 1
```

```
Walked: [:a 1]
Walked: :b
Walked: 2
Walked: [:b 2]
Walked: {:a 1 :b 2}
Walked: (1 2 {:a 1 :b 2})
=> (1 2 {:a 1 :b 2})
```

pow

```
(pow x y)
```

Returns the value of x raised to the power of y

```
(pow 10 2)
=> 100.0

(pow 10.23 2)
=> 104.6529

(pow 10.23 2.5)
=> 334.72571990233183
```

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 ' '.

```
(pr-str )
=> ""

(pr-str 1 2 3)
=> "1 2 3"
```

prewalk

```
(prewalk f form)
```

Performs a depth-last, pre-order traversal of form. Calls f on
each sub-form, uses f's return value in place of the original.

```
(prewalk (fn [x] (println "Walked:" (pr-str x)) x)
         '(1 2 {:a 1 :b 2}))
Walked: (1 2 {:a 1 :b 2})
Walked: 1
Walked: 2
Walked: {:a 1 :b 2}
Walked: [:a 1]
```

```
Walked: :a
Walked: 1
Walked: [:b 2]
Walked: :b
Walked: 2
=> (1 2 {:a 1 :b 2})
```

print

```
(print & xs)
(print os & xs)
```

Without output stream prints to the output stream that is the current
value of *out*. With no args, prints the empty string. With one arg x,
prints x.toString(). With more than one arg, prints the concatenation
of the string values of the args with delimiter ' '.
With an output stream prints to that output stream.
Returns nil.

```
(print [10 20 30])
[10 20 30]
=> nil

(print *out* [10 20 30])
[10 20 30]
=> nil

(print *err* [10 20 30])
[10 20 30]
=> nil
```

printf

```
(printf fmt & args)
(printf os fmt & args)
```

Without output stream prints formatted output as per format to the
output stream that is the current value of *out*.
With an output stream prints to that output stream.
Returns nil.

```
(printf "%s: %d" "abc" 100)
abc: 100
=> nil

(printf "line 1: %s%nline 2: %s%n" "123" "456")
line 1: 123
line 2: 456
=> nil

(printf *out* "%s: %d" "abc" 100)
abc: 100
=> nil

(printf *err* "%s: %d" "abc" 100)
```

```
abc: 100
=> nil
```

println

```
(println & xs)
(println os & xs)
```

```
Without output stream prints to the output stream that is the current
value of *out* with a tailing 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 string values of the args with
delimiter ' '.
With an output stream prints to that output stream.
Returns nil.
```

```
(println 200)
200
=> nil

(println [10 20 30])
[10 20 30]
=> nil

(println *out* 200)
200
=> nil

(println *err* 200)
200
=> nil
```

prof

```
(prof opts)
```

```
Controls the code profiling. See the companion functions/macros 'dorun' and 'perf'. The perf macro is built on
prof and dorun and provides all to do simple Venice profiling.
```

```
(do
  (prof :on)   ; turn profiler on
  (prof :off)   ; turn profiler off
  (prof :status)   ; returns the profiler on/off staus
  (prof :clear)   ; clear profiler data captured so far
  (prof :data)   ; returns the profiler data as map
  (prof :data-formatted)   ; returns the profiler data as formatted text
  (prof :data-formatted "Metrics test")   ; returns the profiler data as formatted text with a title
  nil)
=> nil
```

promise

```
(promise)
```

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.

```
(do
   (def p (promise))
   (deliver p 10)
   (deliver p 20)
   @p)
=> 10

(do
   (def p (promise))
   (defn task1 [] (sleep 500) (deliver p 10))
   (defn task2 [] (sleep 800) (deliver p 20))
   (future task1)
   (future task2)
   @p)
=> 10
```

promise?

```
(promise? p)
```

Returns true if f is a Promise otherwise false

```
(promise? (promise)))
=> true
```

proxify

```
(proxify classname method-map)
```

Proxifies a Java interface to be passed as a Callback object to Java functions. The interface's methods are implemented by Venice functions.

```
(do
   (import :java.io.File :java.io.FilenameFilter)

   (def file-filter
      (fn [dir name] (str/ends-with? name ".xxx")))

   (let [dir (io/tmp-dir )]
      ;; create a dynamic proxy for the interface FilenameFilter
      ;; and implement its function 'accept' by 'file-filter'
      (. dir :list (proxify :FilenameFilter {:accept file-filter})))
)
=> []
```

push!

```
(push! stack v)
```

Pushes an item to a stack.

```
(let [s (stack)]
   (push! s 4)
   (push! s 3)
   (pop! s)
   s)
=> (4)
```

quantile

```
(quantile q coll)
```

Returns the quantile [0.0 .. 1.0] of the values

```
(quantile 0.5 '(3, 7, 8, 5, 12, 14, 21, 13, 18))
=> 12.0
```

```
(quantile 0.5 '(3, 7, 8, 5, 12, 14, 21, 15, 18, 14))
=> 13.0
```

quartiles

```
(quartiles coll)
```

Returns the quartiles (1st, 2nd, and 3rd) of the values

```
(quartiles '(3, 7, 8, 5, 12, 14, 21, 13, 18))
=> (6.0 12.0 16.0)
```

```
(quartiles '(3, 7, 8, 5, 12, 14, 21, 15, 18, 14))
=> (7.0 13.0 15.0)
```

queue

```
(queue )
(queue 100)
```

Creates a new mutable threadsafe bounded or unbounded queue.

```
;unbounded queue
(let [q (queue)]
  (offer! q 1)
  (offer! q 2)
```

```
   (offer! q 3)
   (poll! q)
   q)
=> (2 3)

;bounded queue
(let [q (queue 10)]
   (offer! q 1000 1)
   (offer! q 1000 2)
   (offer! q 1000 3)
   (poll! q 1000)
   q)
=> (2 3)
```

queue?

```
(queue? obj)
```

Returns true if obj is a queue

```
(queue? (queue))
=> true
```

rand-double

```
(rand-double)
(rand-double max)
```

Without argument returns a double between 0.0 and 1.0. With argument max returns a random double between 0.0 and max.
This function is based on a cryptographically strong random number generator (RNG).

```
(rand-double)
=> 0.07972303806316494

(rand-double 100.0)
=> 12.278660243587192
```

rand-gaussian

```
(rand-gaussian)
(rand-gaussian mean stddev)
```

Without argument returns a Gaussion distributed double value with mean 0.0 and standard deviation 1.0. With argument mean and stddev returns a Gaussion distributed double value with the given mean and standard deviation.
This function is based on a cryptographically strong random number generator (RNG)

```
(rand-gaussian)
=> -2.744318553715055
```

```
(rand-gaussian 0.0 5.0)
=> 2.913723066082446
```

rand-long

```
(rand-long)
(rand-long max)
```

Without argument returns a random long between 0 and MAX_LONG. With argument max returns a random long between 0 and max exclusive.
This function is based on a cryptographically strong random number generator (RNG).

```
(rand-long)
=> 885321587692611479

(rand-long 100)
=> 38
```

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.

```
(range 10)
=> (0 1 2 3 4 5 6 7 8 9)

(range 10 20)
=> (10 11 12 13 14 15 16 17 18 19)

(range 10 20 3)
=> (10 13 16 19)

(range (int 10) (int 20))
=> (10I 11I 12I 13I 14I 15I 16I 17I 18I 19I)

(range (int 10) (int 20) (int 3))
=> (10I 13I 16I 19I)

(range 10 15 0.5)
=> (10 10.5 11.0 11.5 12.0 12.5 13.0 13.5 14.0 14.5)

(range 1.1M 2.2M 0.1M)
=> (1.1M 1.2M 1.3M 1.4M 1.5M 1.6M 1.7M 1.8M 1.9M 2.0M 2.1M)
```

read-line

```
(read-line)
```

Reads the next line from the stream that is the current value of *in*.
Returns nil if the of the stream is reached.

read-string

```
(read-string s)
(read-string s origin)
```

Reads from s

```
(do
  (eval (read-string "(def x 100)" "test"))
  x)
=> 100
```

realized?

```
(realized? x)
```

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

```
(do
   (def task (fn [] 100))
   (let [f (future task)]
       (println (realized? f))
       (println @f)
       (println (realized? f))))
false
100
true
=> nil

(do
   (def p (promise))
   (println (realized? p))
   (deliver p 123)
   (println @p)
   (println (realized? p)))
false
123
true
=> nil

(do
   (def x (delay 100))
   (println (realized? x))
   (println @x)
   (println (realized? x)))
false
100
true
=> nil
```

recur

```
(recur expr*)
```

Evaluates the exprs and rebinds the bindings of the recursion point to the values of the exprs. The recur expression must be at the tail position. The tail position is a postion which an expression would return a value from.

```
;; tail recursion
(loop [x 10]
    (when (> x 1)
       (println x)
       (recur (- x 2))))
10
8
6
4
2
=> nil

;; tail recursion
(do
    (defn sum [n]
          (loop [cnt n acc 0]
             (if (zero? cnt)
                 acc
                 (recur (dec cnt) (+ acc cnt)))))
    (sum 10000))
=> 50005000
```

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 (fn [x y] (+ x y)) [1 2 3 4 5 6 7])
=> 28

(reduce (fn [x y] (+ x y)) 10 [1 2 3 4 5 6 7])
=> 38

((reduce comp [(partial + 1) (partial * 2) (partial + 3)]) 100)
=> 207

(reduce (fn [m [k v]] (assoc m v k)) {} {:b 2 :a 1 :c 3})
=> {1 :a 2 :b 3 :c}
```

```
(reduce (fn [m c] (assoc m (first c) c)) {} [[:a 1] [:b 2] [:c 3]])
=> {:a [:a 1] :b [:b 2] :c [:c 3]}
```

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.

```
(reduce-kv (fn [x y z] (assoc x z y)) {} {:a 1 :b 2 :c 3})
=> {1 :a 2 :b 3 :c}
```

reduced

```
(reduced x)
```

Wraps x in a way such that a reduce will terminate with the value x.

reduced?

```
(reduced? x)
```

Returns true if x is the result of a call to reduced.

regex/find

```
(regex/find matcher)
```

Returns the next regex match

```
(let [m (regex/matcher "[0-9]+" "672-345-456-3212")]
   (println (regex/find m))
   (println (regex/find m))
   (println (regex/find m))
   (println (regex/find m))
   (println (regex/find m)))

672
345
456
```

```
3212
nil
=> nil
```

regex/find-all

```
(regex/find-all matcher)
```

Returns all regex matches

```
(->> (regex/matcher "\\d+" "672-345-456-3212")
     (regex/find-all)))
=> ("672" "345" "456" "3212")

(->> (regex/matcher "([^\"]\\S*|\".+?\")\\s*" "1 2 \"3 4\" 5")
     (regex/find-all)))
=> ("1 " "2 " "\"3 4\" " "5")
```

regex/find-all-groups

```
(regex/find-all-groups matcher)
```

Returns the all regex matches and returns the groups

```
(let [m (regex/matcher "[0-9]+" "672-345-456-3212")]
  (regex/find-all-groups m))

=> ({:group "672" :start 0 :end 3} {:group "345" :start 4 :end 7} {:group "456" :start 8 :end 11} {:group
"3212" :start 12 :end 16})
```

regex/find-group

```
(regex/find-group matcher)
```

Returns the next regex match and returns the group

```
(let [m (regex/matcher "[0-9]+" "672-345-456-3212")]
   (println (regex/find-group m))
   (println (regex/find-group m))
   (println (regex/find-group m))
   (println (regex/find-group m))
   (println (regex/find-group m)))

{:group 672 :start 0 :end 3}
{:group 345 :start 4 :end 7}
{:group 456 :start 8 :end 11}
{:group 3212 :start 12 :end 16}
nil
=> nil
```

regex/find?

```
(regex/find matcher)
```

Attempts to find the next subsequence that matches the pattern. If the match succeeds then more information can be obtained via the regex/group function

```
(let [p (regex/pattern "[0-9]+")
      m (regex/matcher p "100")]
   (regex/find? m))
=> true
```

regex/group

```
(regex/group matcher group)
```

Returns the input subsequence captured by the given group during theprevious match operation.

```
(let [p (regex/pattern "([0-9]+)(.*)")
      m (regex/matcher p "100abc")]
   (if (regex/matches? m)
      [(regex/group m 1) (regex/group m 2)]
      []))
=> ["100" "abc"]
```

regex/groupcount

```
(regex/groupcount matcher)
```

Returns the matcher's group count.

```
(let [p (regex/pattern "([0-9]+)(.*)")
      m (regex/matcher p "100abc")]
   (regex/groupcount m))
=> 2
```

regex/matcher

```
(regex/matcher pattern str)
```

Returns an instance of java.util.regex.Matcher. The pattern can be either a string or a pattern created by (regex/pattern s)

```
(regex/matcher "[0-9]+" "100")
=> java.util.regex.Matcher[pattern=[0-9]+ region=0,3 lastmatch=]
```

```
(let [p (regex/pattern "[0-9]+")]
  (regex/matcher p "100"))
=> java.util.regex.Matcher[pattern=[0-9]+ region=0,3 lastmatch=]
```

regex/matches

```
(regex/matches pattern str)
```

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

```
(regex/matches "hello, (.*)" "hello, world")
=> ("hello, world" "world")

(regex/matches "([0-9]+)-([0-9]+)-([0-9]+)-([0-9]+)" "672-345-456-212")
=> ("672-345-456-212" "672" "345" "456" "212")
```

regex/matches?

```
(regex/matches? matcher)
```

Attempts to match the entire region against the pattern. If the match succeeds then more information can be obtained via the regex/group function

```
(let [p (regex/pattern "[0-9]+")
      m (regex/matcher p "100")]
  (regex/matches? m))
=> true
```

regex/pattern

```
(regex/pattern s)
```

Returns an instance of java.util.regex.Pattern.

```
(regex/pattern "[0-9]+")
=> [0-9]+
```

regex/reset

```
(regex/reset matcher str)
```

Resets the matcher with a new string

```
(let [p (regex/pattern "[0-9]+")
      m1 (regex/matcher p "100")
      m2 (regex/reset m1 "200")]
```

```
    (regex/find? m2))
=> true
```

remove

```
(remove predicate coll)
```

Returns a collection of the items in coll for which (predicate item) returns logical false. Returns a transducer when no collection is provided.

```
(remove even? [1 2 3 4 5 6 7])
=> [1 3 5 7]

(remove #{3 5} '(1 3 5 7 9))
=> (1 7 9)
```

remove-watch

```
(remove-watch ref key)
```

Removes a watch function from an agent/atom reference.

```
(do
    (def x (agent 10))
    (defn watcher [key ref old new]
        (println "watcher: " key))
    (add-watch x :test watcher)
    (remove-watch x :test))
=> nil
```

repeat

```
(repeat n x)
```

Returns a collection with the value x repeated n times

```
(repeat 5 [1 2])
=> ([1 2] [1 2] [1 2] [1 2] [1 2])
```

repeatedly

```
(repeatedly n fn)
```

Takes a function of no args, presumably with side effects, and returns a collection of n calls to it

```
(repeatedly 5 #(rand-long 11))
=> (3 6 0 5 1)

;; compare with repeat, which only calls the 'rand-long'
;; function once, repeating the value five times.
(repeat 5 (rand-long 11))
=> (8 8 8 8 8)
```

replace

```
(replace smap coll)
```

Given a map of replacement pairs and a collection, returns a
collection with any elements that are a key in smap replaced with the
corresponding value in smap.

```
(replace {2 :two, 4 :four} [4 2 3 4 5 6 2])
=> [:four :two 3 :four 5 6 :two]

(replace {2 :two, 4 :four} #{1 2 3 4 5})
=> #{1 3 5 :four :two}

(replace {[:a 10] [:c 30]} {:a 10 :b 20})
=> {:b 20 :c 30}
```

reset!

```
(reset! box newval)
```

Sets the value of an atom or a volatile to newval without regard for the current value. Returns newval.

```
(do
  (def counter (atom 0))
  (reset! counter 99)
  @counter)
=> 99

(do
  (def counter (atom 0))
  (reset! counter 99))
=> 99

(do
  (def counter (volatile 0))
  (reset! counter 99)
  @counter)
=> 99
```

resolve

```
(resolve symbol)
```

Resolves a symbol.

```
(resolve '+)
=> function + {visibility :public, ns "core"}

(resolve 'y)
=> nil

(resolve (symbol "+"))
=> function + {visibility :public, ns "core"}

((-> "first" symbol resolve) [1 2 3])
=> 1
```

rest

```
(rest coll)
```

Returns a possibly empty collection of the items after the first.

```
(rest nil)
=> nil

(rest [])
=> []

(rest [1])
=> []

(rest [1 2 3])
=> [2 3]

(rest '())
=> ()

(rest '(1))
=> ()

(rest '(1 2 3))
=> (2 3)

(rest "1234")
=> ("2" "3" "4")
```

restart-agent

```
(restart-agent agent state)
```

When an agent is failed, changes the agent state to new-state and then un-fails the agent so that sends are allowed again.

```
(do
   (def x (agent 100))
   (restart-agent x 200)
```

```
    (deref x))
=> 200
```

reverse

```
(reverse coll)
```

Returns a collection of the items in coll in reverse order. Returns a stateful transducer when no collection is provided.

```
(reverse [1 2 3 4 5 6])
=> [6 5 4 3 2 1]

(reverse "abcdef")
=> ("f" "e" "d" "c" "b" "a")
```

rf-any?

```
(rf-any? pred)
```

Returns a reducing function for a transducer that returns true if the predicate is true for at least one the items, false otherwise.

```
(transduce (filter number?) (rf-any? pos?) [true -1 1 2 false])
=> true
```

rf-every?

```
(rf-every? pred)
```

Returns a reducing function for a transducer that returns true if the predicate is true for all the items, false otherwise.

```
(transduce (filter number?) (rf-every? pos?) [1 2 3])
=> true
```

rf-first

```
(rf-first)
```

Returns a reducing function for a transducer that returns the first item.

```
(transduce (filter number?) rf-first [false 1 2])
=> 1
```

```
(transduce identity rf-first [nil 1 2])
=> nil
```

rf-last

```
(rf-last)
```

Returns a reducing function for a transducer that returns the last item.

```
(transduce (filter number?) rf-last [false 1 2])
=> 2
```

```
(transduce identity rf-last [1 2 1.2])
=> 1.2
```

sandboxed?

```
(sandboxed? )
```

Returns true if there is a sandbox otherwise false

```
(sandboxed? )
=> false
```

schedule-at-fixed-rate

```
(schedule-at-fixed-rate fn initial-delay period time-unit)
```

Creates and executes a periodic action that becomes enabled first after the given initial delay, and
subsequently with the given period.
Returns a future. (future? f), (future-cancel f), and (future-done? f) will work on the returned future.
Time unit is one of :milliseconds, :seconds, :minutes, :hours, or :days.

```
(schedule-at-fixed-rate (fn[] (println "test")) 1 2 :seconds)
```

```
(let [s (schedule-at-fixed-rate (fn[] (println "test")) 1 2 :seconds)]
    (sleep 16 :seconds)
    (future-cancel s))
```

schedule-delay

```
(schedule-delay fn delay time-unit)
```

```
Creates and executes a one-shot action that becomes enabled after the given delay.
Returns a future. (deref f), (future? f), (future-cancel f), and (future-done? f) will work on the returned
future.
Time unit is one of :milliseconds, :seconds, :minutes, :hours, or :days.
```

```
(schedule-delay (fn[] (println "test")) 1 :seconds)
```

```
(deref (schedule-delay (fn [] 100) 2 :seconds))
```

second

```
(second coll)
```

```
Returns the second element of coll.
```

```
(second nil)
=> nil

(second [])
=> nil

(second [1 2 3])
=> 2

(second '())
=> nil

(second '(1 2 3))
=> 2
```

send

```
(send agent action-fn args)
```

```
Dispatch an action to an agent. Returns the agent immediately.The state of the agent will be set to the value
of:
 (apply action-fn state-of-agent args)
```

```
(do
   (def x (agent 100))
   (send x + 5)
   (send x (partial + 7))
   (sleep 100)
   (deref x))
=> 112
```

send-off

```
(send-off agent fn args)
```

Dispatch a potentially blocking action to an agent. Returns the agent immediately. The state of the agent will be set to the value of:
 (apply action-fn state-of-agent args)

```
(do
   (def x (agent 100))
   (send-off x + 5)
   (send-off x (partial + 7))
   (sleep 100)
   (deref x))
=> 112
```

seq

```
(seq coll)
```

Returns a seq on the collection. If the collection is empty, returns nil. (seq nil) returns nil. seq also works on Strings.

```
(seq nil)
=> nil

(seq [1 2 3])
=> (1 2 3)

(seq '(1 2 3))
=> (1 2 3)

(seq {:a 1 :b 2})
=> ([:a 1] [:b 2])

(seq "abcd")
=> ("a" "b" "c" "d")
```

sequential?

```
(sequential? obj)
```

Returns true if obj is a sequential collection

```
(sequential? '(1))
=> true

(sequential? [1])
=> true

(sequential? {:a 1})
=> false

(sequential? nil)
=> false

(sequential? "abc")
=> false
```

set

```
(set & items)
```

Creates a new set containing the items.

```
(set )
=> #{}

(set nil)
=> #{nil}

(set 1)
=> #{1}

(set 1 2 3)
=> #{1 2 3}

(set [1 2] 3)
=> #{3 [1 2]}
```

set!

```
(set! var-symbol expr)
```

Sets a global or thread-local variable to the value of the expression.

```
(do
  (def x 10)
  (set! x 20)
  x)
=> 20

(do
   (def-dynamic x 100)
   (set! x 200)
   x)
=> 200

(do
   (def-dynamic x 100)
   (with-out-str
     (print x)
     (binding [x 200]
       (print (str "-" x))
       (set! x (inc x))
       (print (str "-" x)))
     (print (str "-" x))))
=> "100-200-201-100"
```

set-error-handler!

```
(set-error-handler! agent handler-fn)
```

Sets the error-handler of an agent to handler-fn. If an action being run by the agent throws an exception
handler-fn will be called with two arguments: the agent and the exception.

```
(do
   (def x (agent 100))
   (defn err-handler-fn [ag ex]
      (println "error occured: "
                (:message ex)
                " and we still have value"
                @ag))
   (set-error-handler! x err-handler-fn)
   (send x (fn [n] (/ n 0))))
=> (agent :value 100)
```

set?

```
(set? obj)
```

Returns true if obj is a set

```
(set? (set 1))
=> true
```

sh

```
(sh & args)
```

Passes the given strings to Runtime.exec() to launch a sub-process.

Options are
  :in   may be given followed by input source as InputStream,
            Reader, File, ByteBuf, or String, to be fed to the
            sub-process's stdin.
  :in-enc   option may be given followed by a String, used as a
            character encoding name (for example "UTF-8" or
            "ISO-8859-1") to convert the input string specified
            by the :in option to the sub-process's stdin. Defaults
            to UTF-8. If the :in option provides a byte array,
            then the bytes are passed unencoded, and this option
            is ignored.
  :out-enc option may be given followed by :bytes or a String. If
            a String is given, it will be used as a character
            encoding name (for example "UTF-8" or "ISO-8859-1")
            to convert the sub-process's stdout to a String which is
            returned. If :bytes is given, the sub-process's stdout
            will be stored in a Bytebuf and returned. Defaults to
            UTF-8.
  :out-fn   a function with a single string argument that receives
            line by line from the process' stdout. If passed the
            :out value in the return map will be empty.
  :err-fn   a function with a single string argument that receives
            line by line from the process' stderr. If passed the
```

```
            :err value in the return map will be empty.
  :env      override the process env with a map.
  :dir      override the process dir with a String or java.io.File.
  :throw-ex  If true throw an exception if the exit code is not equal
            to zero, if false returns the exit code. Defaults to
            false. It's recommended to use (with-sh-throw (sh "foo"))
            instead.

You can bind :env, :dir for multiple operations using with-sh-env or
with-sh-dir. with-sh-throw is binds :throw-ex as true.

sh returns a map of
  :exit => sub-process's exit code
  :out  => sub-process's stdout (as Bytebuf or String)
  :err  => sub-process's stderr (String via platform default encoding)
```

```clojure
(println (sh "ls" "-l"))


(println (sh "ls" "-l" "/tmp"))


(println (sh "sed" "s/[aeiou]/oo/g" :in "hello there\n"))


(println (sh "cat" :in "x\u25bax\n"))


(println (sh "echo" "x\u25bax"))


(println (sh "/bin/sh" "-c" "ls -l"))


(sh "ls" "-l" :out-fn println)


(sh "ls" "-l" :out-fn println :err-fn println)


;; background process
(println (sh "/bin/sh" "-c" "sleep 30 >/dev/null 2>&1 &"))


(println (sh "/bin/sh" "-c" "nohup sleep 30 >/dev/null 2>&1 &"))


;; reads 4 single-byte chars
(println (sh "echo" "x\u25bax" :out-enc "ISO-8859-1"))


;; reads binary file into bytes[]
(println (sh "cat" "birds.jpg" :out-enc :bytes))


;; working directory
(println (with-sh-dir "/tmp" (sh "ls" "-l") (sh "pwd")))


(println (sh "pwd" :dir "/tmp"))


;; throw an exception if the shell's subprocess exit code is not equal to 0
(println (with-sh-throw (sh "ls" "-l")))
```

```
(println (sh "ls" "-l" :throw-ex true))


;; windows
(println (sh "cmd" "/c dir 1>&2"))
```

shuffle

```
(shuffle coll)
```

Returns a collection of the items in coll in random order.

```
(shuffle '(1 2 3 4 5 6))
=> (2 3 5 1 6 4)

(shuffle [1 2 3 4 5 6])
=> [4 1 2 3 6 5]

(shuffle "abcdef")
=> ("b" "f" "a" "d" "c" "e")
```

shutdown-agents

```
(shutdown-agents )
```

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

```
(do
   (def x1 (agent 100))
   (def x2 (agent 100))
   (shutdown-agents ))
```

shutdown-agents?

```
(shutdown-agents?)
```

Returns true if the thread-pool that backs the agents is shut down

```
(do
   (def x1 (agent 100))
   (def x2 (agent 100))
   (shutdown-agents )
   (sleep 300)
   (shutdown-agents? ))
```

shutdown-hook

```
(shutdown-hook f)
```

Registers the function f as JVM shutdown hook.

```
(shutdown-hook (fn [] (println "shutdown")))
=> nil
```

sin

```
(sin x)
```

sin x

```
(sin 1)
=> 0.8414709848078965

(sin 1.23)
=> 0.9424888019316975

(sin 1.23M)
=> 0.9424888019316975
```

sleep

```
(sleep n)
(sleep n time-unit)
```

Sleep for the time n. The default time unit is milliseconds
Time unit is one of :milliseconds, :seconds, :minutes, :hours, or :days.

```
(sleep 30)
=> nil

(sleep 30 :milliseconds)
=> nil

(sleep 5 :seconds)
=> nil
```

some

```
(some pred coll)
```

Returns the first logical true value of (pred x) for any x in coll, else nil.
Stops processing the collection if the first value is found that meets the predicate.

```
(some even? '(1 2 3 4))
=> true

(some even? '(1 3 5 7))
=> nil

(some #(== 5 %) [1 2 3 4 5])
=> true

(some #(if (even? %) %) [1 2 3 4])
=> 2
```

some?

```
(some? x)
```

Returns true if x is not nil, false otherwise

```
(some? nil)
=> false

(some? 0)
=> true

(some? 4.0)
=> true

(some? false)
=> true

(some? [])
=> true

(some? {})
=> true
```

sort

```
(sort coll)
(sort comparefn coll)
```

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

```
(sort [3 2 5 4 1 6])
=> [1 2 3 4 5 6]

(sort compare [3 2 5 4 1 6])
=> [1 2 3 4 5 6]

; reversed
(sort (comp - compare) [3 2 5 4 1 6])
=> [6 5 4 3 2 1]
```

```
(sort {:c 3 :a 1 :b 2})
=> ([:a 1] [:b 2] [:c 3])
```

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 :id [{:id 2 :name "Smith"} {:id 1 :name "Jones"} ])
=> [{:name "Jones" :id 1} {:name "Smith" :id 2}]

(sort-by count ["aaa" "bb" "c"])
=> ["c" "bb" "aaa"]

; reversed
(sort-by count (comp - compare) ["aaa" "bb" "c"])
=> ["aaa" "bb" "c"]

(sort-by first [[1 2] [3 4] [2 3]])
=> [[1 2] [2 3] [3 4]]

; reversed
(sort-by first (comp - compare) [[1 2] [3 4] [2 3]])
=> [[3 4] [2 3] [1 2]]

(sort-by :rank [{:rank 2} {:rank 3} {:rank 1}])
=> [{:rank 1} {:rank 2} {:rank 3}]

; reversed
(sort-by :rank (comp - compare) [{:rank 2} {:rank 3} {:rank 1}])
=> [{:rank 3} {:rank 2} {:rank 1}]

; sort by :foo, and where :foo is equal, sort by :bar
(def x [ {:foo 2 :bar 11}
         {:foo 1 :bar 99}
         {:foo 2 :bar 55}
         {:foo 1 :bar 77} ] )
(sort-by (juxt :foo :bar) x)
=> user/x [{:foo 1 :bar 77} {:foo 1 :bar 99} {:foo 2 :bar 11} {:foo 2 :bar 55}]
```

sorted

```
(sorted cmp coll)
```

Returns a sorted collection using the compare function cmp. The compare function takes two arguments and
returns -1, 0, or 1. Returns a stateful transducer when no collection is provided.

```
(sorted compare [4 2 1 5 6 3])
=> [1 2 3 4 5 6]
```

```
(sorted (comp (partial * -1) compare) [4 2 1 5 6 3])
=> [6 5 4 3 2 1]
```

sorted-map

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

Creates a new sorted map containing the items.

```
(sorted-map :a 1 :b 2)
=> {:a 1 :b 2}
```

```
(sorted-map (hash-map :a 1 :b 2))
=> {:a 1 :b 2}
```

sorted-map?

```
(sorted-map? obj)
```

Returns true if obj is a sorted map

```
(sorted-map? (sorted-map :a 1 :b 2))
=> true
```

sorted-set

```
(sorted-set & items)
```

Creates a new sorted-set containing the items.

```
(sorted-set )
=> #{}
```

```
(sorted-set nil)
=> #{nil}
```

```
(sorted-set 1)
=> #{1}
```

```
(sorted-set 6 2 4)
=> #{2 4 6}
```

```
(str (sorted-set [2 3] [1 2]))
=> "#{[1 2] [2 3]}"
```

sorted-set?

```
(sorted-set? obj)
```

Returns true if obj is a sorted-set

```
(sorted-set? (sorted-set 1))
=> true
```

split-at

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

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

```
(split-at 2 [1 2 3 4 5])
=> [(1 2) (3 4 5)]
```

```
(split-at 3 [1 2])
=> [(1 2) ()]
```

split-with

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

Splits the collection at the first false/nil predicate result in a vector with two lists

```
(split-with odd? [1 3 5 6 7 9])
=> [(1 3 5) (6 7 9)]
```

```
(split-with odd? [1 3 5])
=> [(1 3 5) ()]
```

```
(split-with odd? [2 4 6])
=> [() (2 4 6)]
```

sqrt

```
(sqrt x)
```

Square root of x

```
(sqrt 10)
=> 3.1622776601683795
```

```
(sqrt 10I)
=> 3.1622776601683795
```

```
(sqrt 10.23)
=> 3.198371183438953

(sqrt 10.23M)
=> 3.1984371183438953245570246508577838540077209472656250M
```

square

```
(square x)
```

```
Square of x
```

```
(square 10)
=> 100

(square 10I)
=> 100I

(square 10.23)
=> 104.6529

(square 10.23M)
=> 104.6529M
```

stack

```
(stack )
```

```
Creates a new mutable threadsafe stack.
```

```
(let [s (stack)]
   (push! s 4)
   (push! s 3)
   (pop! s)
   s)
=> (4)
```

stack?

```
(stack? obj)
```

```
Returns true if obj is a stack
```

```
(stack? (stack))
=> true
```

standard-deviation

```
(standard-deviation type coll)
```

Returns the standard deviation of the values for data sample type :population or :sample.

```
(standard-deviation :sample '(10 8 30 22 15))
=> 9.055385138137417

(standard-deviation :population '(10 8 30 22 15))
=> 8.099382692526634

(standard-deviation :sample '(1.4 3.6 7.8 9.0 2.2))
=> 3.40587727318528

(standard-deviation :sample '(2.8M 6.4M 2.0M 4.4M))
=> 1.942506971244462
```

top

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 )
=> ""

(str 1 2 3)
=> "123"

(str +)
=> "function + {visibility :public, ns \"core\"}"

(str [1 2 3])
=> "[1 2 3]"
```

top

str/blank?

```
(str/blank? s)
```

True if s is nil, empty, or contains only whitespace.

```
(str/blank? nil)
=> true

(str/blank? "")
=> true

(str/blank? "  ")
=> true

(str/blank? "abc")
=> false
```

str/bytebuf-to-hex

```
(str/bytebuf-to-hex data)
(str/bytebuf-to-hex data :upper)
```

Converts byte data to a hex string using the hexadecimal digits: 0123456789abcdef.
If the :upper options is passed the hex digits 0123456789ABCDEF are used.

```
(str/bytebuf-to-hex (bytebuf [0 1 2 3 4 5 6]))
=> "00010203040506"
```

str/char?

```
(str/char s)
```

Returns true if s is a single char string.

```
(str/char? "x")
=> true

(str/char? (char "x"))
=> true
```

str/chars

```
(str/chars s)
```

Converts a string to a char list.

```
(str/chars "abcdef")
=> ("a" "b" "c" "d" "e" "f")

(str/join (str/chars "abcdef"))
=> "abcdef"
```

str/contains?

```
(str/contains? s substr)
```

True if s contains with substr.

```
(str/contains? "abc" "ab")
=> true

(str/contains? "abc" (char "b"))
=> true
```

str/cr-lf

```
(str/cr-lf s mode)
```

Convert a text to use LF or CR-LF.

```
(str/cr-lf "line1
line2
line3" :cr-lf)


(str/cr-lf "line1
line2
line3" :lf)
```

str/decode-base64

```
(str/decode-base64 s)
```

Base64 decode.

```
(str/decode-base64 (str/encode-base64 (bytebuf [0 1 2 3 4 5 6])))
=> [0 1 2 3 4 5 6]
```

str/decode-url

```
(str/decode-url s)
```

URL decode.

```
(str/decode-url "The+string+%C3%BC%40foo-bar")
=> "The string ü@foo-bar"
```

str/digit?

```
(str/digit? s)
```

True if s is a single char string and the char is a digit. Defined by Java Character.isDigit(ch).

```
(str/digit? (char "8"))
=> true

(str/digit? "8")
=> true
```

str/double-quote

```
(str/double-quote str)
```

Double quotes a string.

```
(str/double-quote "abc")
=> "\"abc\""

(str/double-quote "")
=> "\"\""
```

str/double-quoted?

```
(str/double-quoteed? str)
```

Returns true if the string is double quoted.

```
(str/double-quoted? "\"abc\"")
=> true
```

str/double-unquote

```
(str/double-unquote str)
```

Unquotes a double quoted string.

```
(str/double-unquote "\"abc\"")
=> "abc"

(str/double-unquote "\"\"")
=> ""

(str/double-unquote nil)
=> nil
```

str/encode-base64

```
(str/encode-base64 data)
```

Base64 encode.

```
(str/encode-base64 (bytebuf [0 1 2 3 4 5 6]))
=> "AAECAwQFBg=="
```

str/encode-url

```
(str/encode-url s)
```

URL encode.

```
(str/encode-url "The string ü@foo-bar")
=> "The+string+%C3%BC%40foo-bar"
```

str/ends-with?

```
(str/ends-with? s substr)
```

True if s ends with substr.

```
(str/ends-with? "abc"  "bc")
=> true
```

str/equals-ignore-case?

```
(str/equals-ignore-case? s1 s2)
```

Compares two strings ignoring case. True if both are equal.

```
(str/equals-ignore-case? "abc"  "abC")
=> true
```

str/escape-html

```
(str/escape-html s)
```

HTML escape. Escapes &, <, >, ", ', and the non blocking space U+00A0

```
(str/escape-html "1 2 3 & < > \" ' \u00A0")
=> "1 2 3 &amp; &lt; &gt; &quot; &apos;  "
```

str/escape-xml

```
(str/escape-xml s)
```

XML escape. Escapes &, <, >, ", '

```
(str/escape-xml "1 2 3 & < > \" '")
=> "1 2 3 &amp; &lt; &gt; &quot; &apos;"
```

str/expand

```
(str/expand s len fill mode*)
```

Expands a string to the max lenght len. Fills up with the fillstring if the string needs to be expanded. The fill string is added to the start or end of the string depending on the mode:start, :end. The mode defaults to :end

```
(str/expand "abcdefghij" 8 ".")
=> "abcdefghij"

(str/expand "abcdefghij" 20 ".")
=> "abcdefghij.........."

(str/expand "abcdefghij" 20 "." :start)
=> "..........abcdefghij"

(str/expand "abcdefghij" 20 "." :end)
=> "abcdefghij.........."

(str/expand "abcdefghij" 30 "1234" :start)
=> "12341234123412341234abcdefghij"

(str/expand "abcdefghij" 30 "1234" :end)
=> "abcdefghij12341234123412341234"
```

str/format

```
(str/format format args*)
(str/format locale format args*)
```

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

```
(str/format "value: %.4f" 1.45)
=> "value: 1.4500"

(str/format (. :java.util.Locale :new "de" "DE") "value: %.4f" 1.45)
=> "value: 1,4500"

(str/format (. :java.util.Locale :GERMANY) "value: %.4f" 1.45)
=> "value: 1,4500"

(str/format (. :java.util.Locale :new "de" "CH") "value: %,d" 2345000)
=> "value: 2'345'000"

(str/format [ "de"] "value: %.4f" 1.45)
=> "value: 1,4500"

(str/format [ "de" "DE"] "value: %.4f" 1.45)
=> "value: 1,4500"
```

```
(str/format [ "de" "DE"] "value: %,d" 2345000)
=> "value: 2.345.000"
```

str/format-bytebuf

```
(str/format-bytebuf data delimiter & options)
```

```
Formats a bytebuffer.

Options
  :prefix0x - prefix with 0x
```

```
(str/format-bytebuf (bytebuf [0 34 67 -30 -1]) nil)
=> "002243E2FF"
```

```
(str/format-bytebuf (bytebuf [0 34 67 -30 -1]) "")
=> "002243E2FF"
```

```
(str/format-bytebuf (bytebuf [0 34 67 -30 -1]) ", ")
=> "00, 22, 43, E2, FF"
```

```
(str/format-bytebuf (bytebuf [0 34 67 -30 -1]) ", " :prefix0x)
=> "0x00, 0x22, 0x43, 0xE2, 0xFF"
```

str/hex-to-bytebuf

```
(str/hex-to-bytebuf hex)
```

```
Converts a hex string to a bytebuf
```

```
(str/hex-to-bytebuf "005E4AFF")
=> [0 94 74 255]
```

```
(str/hex-to-bytebuf "005e4aff")
=> [0 94 74 255]
```

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/index-of "abcdefabc" "ab")
=> 0
```

str/join

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

Joins all elements in coll separated by an optional separator.

```
(str/join [1 2 3])
=> "123"

(str/join "-" [1 2 3])
=> "1-2-3"

(str/join "-" [(char "a") 1 "xyz" 2.56M])
=> "a-1-xyz-2.56M"
```

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/last-index-of "abcdefabc" "ab")
=> 6
```

str/letter?

```
(str/letter? s)
```

True if s is a single char string and the char is a letter. Defined by Java Character.isLetter(ch).

```
(str/letter? (char "x"))
=> true

(str/letter? "x")
=> true
```

str/linefeed?

```
(str/linefeed? s)
```

True if s is a single char string and the char is a linefeed.

```
(str/linefeed? (char "
"))
=> true
```

```
(str/linefeed? "
")
=> true
```

str/lorem-ipsum

```
(str/lorem-ipsum & options)
```

```
Creates an arbitrary length Lorem Ipsum text.

Options:
  :chars n - returns n characters (limited to 1000000)
  :paragraphs n - returns n paragraphs (limited to 100
```

```
(str/lorem-ipsum :chars 250)
=> "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent ac iaculis turpis. Duis dictum id sem et
consectetur. Nullam lobortis, libero non consequat aliquet, lectus diam fringilla velit, finibus eleifend ipsum
urna at lacus. Phasellus sit am"

(str/lorem-ipsum :paragraphs 1)
=> "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent ac iaculis turpis. Duis dictum id sem et
consectetur. Nullam lobortis, libero non consequat aliquet, lectus diam fringilla velit, finibus eleifend ipsum
urna at lacus. Phasellus sit amet nisl fringilla, cursus est in, mollis lacus. Proin dignissim rhoncus dolor.
Cras tellus odio, elementum sed erat sit amet, euismod tincidunt nisl. In hac habitasse platea dictumst. Duis
aliquam sollicitudin tempor. Sed gravida tincidunt felis at fringilla. Morbi tempor enim at commodo vulputate.
Aenean et ultrices lorem, placerat pretium augue. In hac habitasse platea dictumst. Cras fringilla ligula quis
interdum hendrerit. Etiam at massa tempor, facilisis lacus placerat, congue erat."
```

str/lower-case

```
(str/lower-case s)
(str/lower-case locale s)
```

```
Converts s to lowercase
```

```
(str/lower-case "aBcDeF")
=> "abcdef"

(str/lower-case (. :java.util.Locale :new "de" "DE") "aBcDeF")
=> "abcdef"

(str/lower-case (. :java.util.Locale :GERMANY) "aBcDeF")
=> "abcdef"

(str/lower-case (. :java.util.Locale :new "de" "CH") "aBcDeF")
=> "abcdef"

(str/lower-case [ "de"] "aBcDeF")
=> "abcdef"

(str/lower-case [ "de" "DE"] "aBcDeF")
=> "abcdef"
```

```
(str/lower-case [ "de" "DE"] "aBcDeF")
=> "abcdef"
```

str/lower-case?

```
(str/lower-case? s)
```

True if s is a single char string and the char is a lower case char. Defined by Java Character.isLowerCase(ch).

```
(str/lower-case? (char "x"))
=> true

(str/lower-case? "x")
=> true
```

str/quote

```
(str/quote str q)
(str/quote str start end)
```

Quotes a string.

```
(str/quote "abc" "-")
=> "-abc-"

(str/quote "abc" "<" ">")
=> "<abc>"
```

str/quoted?

```
(str/quoted? str q)
(str/quoted? str start end)
```

Returns true if the string is quoted.

```
(str/quoted? "-abc-" "-")
=> true

(str/quoted? "<abc>" "<" ">")
=> true
```

str/repeat

```
(str/repeat s n)
(str/repeat s n sep)
```

Repeats s n times with an optional separator.

```
(str/repeat "abc" 0)
=> ""

(str/repeat "abc" 3)
=> "abcabcabc"

(str/repeat "abc" 3 "-")
=> "abc-abc-abc"
```

str/replace-all

```
(str/replace-all s search replacement)
```

Replaces the all occurrances of search in s. The search arg may be a string or a regex pattern

```
(str/replace-all "abcdefabc" "ab" "__")
=> "__cdef__c"

(str/replace-all "a0b01c012d" (regex/pattern "[0-9]+") "_")
=> "a_b_c_d"
```

str/replace-first

```
(str/replace-first s search replacement & options)
```

Replaces the first occurrence of search in s. The search arg may be astring or a regex pattern. If the search arg is of type string the options :ignore-case and :nfirst are supported.

```
Options:
  :ignore-case true/false - e.g :ignore-case true, defaults to false
  :nfirst n - e.g :nfirst 2, defaults to 1
```

```
(str/replace-first "ab-cd-ef-ab-cd" "ab" "XYZ")
=> "XYZ-cd-ef-ab-cd"

(str/replace-first "AB-CD-EF-AB-CD" "ab" "XYZ" :ignore-case true)
=> "XYZ-CD-EF-AB-CD"

(str/replace-first "ab-ab-cd-ab-ef-ab-cd" "ab" "XYZ" :nfirst 3)
=> "XYZ-XYZ-cd-XYZ-ef-ab-cd"

(str/replace-first "a0b01c012d" (regex/pattern "[0-9]+") "_")
=> "a_b01c012d"
```

str/replace-last

```
(str/replace-last s search replacement & options)
```

Replaces the last occurrance of search in s

```
(str/replace-last "abcdefabc" "ab" "XYZ")
=> "abcdefXYZc"

(str/replace-last "foo.JPG" ".jpg" ".png" :ignore-case true)
=> "foo.png"
```

str/reverse

```
(str/reverse s)
```

Reverses a string

```
(str/reverse "abcdef")
=> "fedcba"
```

str/split

```
(str/split s regex)
```

Splits string on a regular expression.

```
(str/split "abc,def,ghi" ",")
=> ("abc" "def" "ghi")

(str/split "abc , def , ghi" "[ *],[ *]")
=> ("abc" "def" "ghi")

(str/split "abc,def,ghi" "((?<=,)|(?=,))")
=> ("abc" "," "def" "," "ghi")

(str/split nil ",")
=> ()
```

str/split-lines

```
(str/split-lines s)
```

Splits s into lines.

```
(str/split-lines "line1
line2
line3")
=> ("line1" "line2" "line3")
```

str/starts-with?

```
(str/starts-with? s substr)
```

True if s starts with substr.

```
(str/starts-with? "abc"  "ab")
=> true
```

str/strip-end

```
(str/strip-end s substr)
```

Removes a substr only if it is at the end of a s, otherwise returns s.

```
(str/strip-end "abcdef" "def")
=> "abc"
```

```
(str/strip-end "abcdef" "abc")
=> "abcdef"
```

str/strip-indent

```
(str/strip-indent s)
```

Strip the indent of a multi-line string. The first line's leading whitespaces define the indent.

```
(str/strip-indent "  line1
    line2
    line3")
=> "line1\n  line2\n  line3"
```

str/strip-margin

```
(str/strip-margin s)
```

Strips leading whitespaces upto and including the margin '|' from each line in a multi-line string.

```
(str/strip-margin "line1
  |  line2
  |  line3")
=> "line1\n  line2\n  line3"
```

str/strip-start

```
(str/strip-start s substr)
```

Removes a substr only if it is at the beginning of a s, otherwise returns s.

```
(str/strip-start "abcdef" "abc")
=> "def"

(str/strip-start "abcdef" "def")
=> "abcdef"
```

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/subs "abcdef" 2)
=> "cdef"

(str/subs "abcdef" 2 5)
=> "cde"
```

str/trim

```
(str/trim s)
```

Trims leading and trailing spaces from s.

```
(str/trim " abc  ")
=> "abc"
```

str/trim-to-nil

```
(str/trim-to-nil s)
```

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

```
(str/trim-to-nil "")
=> nil

(str/trim-to-nil "    ")
=> nil

(str/trim-to-nil nil)
=> nil

(str/trim-to-nil " abc   ")
=> "abc"
```

str/truncate

```
(str/truncate s maxlen marker mode*)
```

Truncates a string to the max lenght maxlen and adds the marker if the string needs to be truncated. The marker is added to the start, middle, or end of the string depending on the mode :start, :middle, :end. The mode defaults to :end

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

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

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

(str/truncate "abcdefghij" 7 "..." :start)
=> "...ghij"

(str/truncate "abcdefghij" 7 "..." :middle)
=> "ab...ij"

(str/truncate "abcdefghij" 7 "..." :end)
=> "abcd..."
```

str/upper-case

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

Converts s to uppercase

```
(str/upper-case "aBcDeF")
=> "ABCDEF"

(str/upper-case (. :java.util.Locale :new "de" "DE") "aBcDeF")
=> "ABCDEF"

(str/upper-case (. :java.util.Locale :GERMANY) "aBcDeF")
=> "ABCDEF"

(str/upper-case (. :java.util.Locale :new "de" "CH") "aBcDeF")
=> "ABCDEF"

(str/upper-case [ "de"] "aBcDeF")
=> "ABCDEF"

(str/upper-case [ "de" "DE"] "aBcDeF")
=> "ABCDEF"

(str/upper-case [ "de" "DE"] "aBcDeF")
=> "ABCDEF"
```

str/upper-case?

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

True if s is a single char string and the char is an upper case char. Defined by Java Character.isUpperCase(ch).

```
(str/upper-case? (char "X"))
=> true

(str/upper-case? "X")
=> true
```

str/valid-email-addr?

```
(str/valid-email-addr? e)
```

Returns true if e is a valid email address according to RFC5322, else returns false

```
(str/valid-email-addr? "user@domain.com")
=> true

(str/valid-email-addr? "user@domain.co.in")
=> true

(str/valid-email-addr? "user.name@domain.com")
=> true

(str/valid-email-addr? "user_name@domain.com")
=> true

(str/valid-email-addr? "username@yahoo.corporate.in")
=> true
```

str/whitespace?

```
(str/whitespace? s)
```

True if s is a single char string and the char is a whitespace. Defined by Java Character.isWhitespace(ch).

```
(str/whitespace? (char " "))
=> true

(str/whitespace? " ")
=> true
```

string-array

```
(string-array coll)
```

```
(string-array len)
(string-array len init-val)
```

Returns an array of Java strings containing the contents of coll or returns an array with the given length and optional init value

```
(string-array '("1" "2" "3"))
=> [1, 2, 3]

(string-array 10)
=> [nil, nil, nil, nil, nil, nil, nil, nil, nil, nil]

(string-array 10 "42")
=> [42, 42, 42, 42, 42, 42, 42, 42, 42, 42]
```

string?

```
(string? x)
```

Returns true if x is a string

```
(string? "abc")
=> true

(string? 1)
=> false

(string? nil)
=> false
```

subvec

```
(subvec v start) (subvec v start end)
```

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

```
(subvec [1 2 3 4 5 6] 2)
=> [3 4 5 6]

(subvec [1 2 3 4 5 6] 4)
=> [5 6]
```

supers

```
(supers class)
```

Returns the immediate and indirect superclasses and interfaces of class, if any.

```
(supers :java.util.ArrayList)
=> (:java.util.AbstractList :java.util.AbstractCollection :java.util.List :java.util.Collection :java.lang.
Iterable)
```

supertype

```
(supertype x)
```

Returns the super type of x.

```
(supertype 5)
=> :core/val

(supertype [1 2])
=> :core/sequence

(supertype (. :java.math.BigInteger :valueOf 100))
=> :java.lang.Number
```

swap!

```
(swap! box f & args)
```

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

```
(do
   (def counter (atom 0))
   (swap! counter inc)
   @counter)
=> 1

(do
   (def counter (atom 0))
   (swap! counter inc))
=> 1

(do
   (def counter (volatile 0))
   (swap! counter (partial + 6))
   @counter)
=> 6
```

symbol

```
(symbol name)
```

Returns a symbol from the given name

```
(symbol "a")
=> a

(symbol 'a)
=> a
```

symbol?

```
(symbol? x)
```

```
Returns true if x is a symbol
```

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

(symbol? 'a)
=> true

(symbol? nil)
=> false

(symbol? :a)
=> false
```

system-env

```
(system-env name default-val)
```

```
Returns the system env variable with the given name. Returns the default-val if the variable does not exist or
it's value is nil
```

```
(system-env :SHELL)
=> "/bin/bash"

(system-env :FOO "test")
=> "test"
```

system-exit-code

```
(system-exit-code code)
```

```
Defines the exit code that is used if the Java VM exits. Defaults to 0.
```

```
Note: The exit code is only used when the Venice launcher has been used to run a script file, a command line
script, a Venice app archive, or the REPL.
```

```
(system-exit-code 0)
=> nil
```

system-prop

```
(system-prop name default-val)
```

Returns the system property with the given name. Returns the default-val if the property does not exist or it's value is nil

```
(system-prop :os.name)
=> "Mac OS X"

(system-prop :foo.org "abc")
=> "abc"

(system-prop "os.name")
=> "Mac OS X"
```

take

```
(take n coll)
```

Returns a collection of the first n items in coll, or all items if there are fewer than n. Returns a stateful transducer when no collection is provided.

```
(take 3 [1 2 3 4 5])
=> [1 2 3]

(take 10 [1 2 3 4 5])
=> [1 2 3 4 5]
```

take-while

```
(take-while predicate coll)
```

Returns a list of successive items from coll while (predicate item) returns logical true. Returns a transducer when no collection is provided.

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

tan

```
(tan x)
```

tan x

```
(tan 1)
=> 1.5574077246549023
```

```
(tan 1.23)
=> 2.819815734268152

(tan 1.23M)
=> 2.819815734268152
```

third

```
(third coll)
```

```
Returns the third element of coll.
```

```
(third nil)
=> nil
```

```
(third [])
=> nil
```

```
(third [1 2 3])
=> 3
```

```
(third '())
=> nil
```

```
(third '(1 2 3))
=> 3
```

thread-id

```
(thread-id)
```

```
Returns the identifier of this Thread. The thread ID is a positive number generated when this thread was
created. The thread ID  is unique and remains unchanged during its lifetime. When a thread is terminated, this
thread ID may be reused.
```

```
(thread-id)
=> 1
```

thread-interrupted

```
(thread-interrupted)
```

```
Tests whether the current thread has been interrupted. The interrupted status of the thread is cleared by this
method. In other words, if this method were to be called twice in succession, the second call would return
false (unless the current thread were interrupted again, after the first call had cleared its interrupted
status and before the second call had examined it).
Returns true if the current thread has been interrupted else false.
```

```
(thread-interrupted)
=> false
```

thread-interrupted?

```
(thread-interrupted?)
```

Tests whether this thread has been interrupted. The interrupted status of the thread is unaffected by this
method.
Returns true if the current thread has been interrupted else false.

```
(thread-interrupted?)
=> false
```

thread-local

```
(thread-local)
```

Creates a new thread-local accessor

```
(thread-local :a 1 :b 2)
=> ThreadLocal

(thread-local { :a 1 :b 2 })
=> ThreadLocal

(do
    (thread-local-clear)
    (assoc (thread-local) :a 1 :b 2)
    (dissoc (thread-local) :a)
    (get (thread-local) :b 100)
)
=> 2
```

thread-local-clear

```
(thread-local-clear)
```

Removes all thread local vars

```
(thread-local-clear)
=> function thread-local-clear {visibility :public, ns "core"}
```

thread-local?

```
(thread-local? x)
```

Returns true if x is a thread-local, otherwise false

```
(do
   (def x (thread-local))
   (thread-local? x))
=> true
```

thread-name

```
(thread-name)
```

Returns this thread's name.

```
(thread-name)
=> "main"
```

throw

```
(throw)
(throw x)
```

Throws exception with passed value x

```
(do
   (try
      (+ 100 200)
      (catch :Exception ex (:message ex))))
=> 300

(do
   (try
      (throw 100)
      (catch :ValueException ex (:value ex))))
=> 100

(do
   (try
      (throw [100 {:a 3}])
      (catch :ValueException ex (:value ex))
      (finally (println "#finally"))))
#finally
=> [100 {:a 3}]

(do
   (import :java.lang.RuntimeException)
   (try
      (throw (. :RuntimeException :new "#test"))
      (catch :RuntimeException ex (:message ex))))
=> "#test"

;; Venice wraps thrown checked exceptions with a RuntimeException!
(do
   (import :java.lang.RuntimeException)
   (import :java.io.IOException)
```

```
   (try
      (throw (. :IOException :new "#test"))
      (catch :RuntimeException ex (:message (:cause ex)))))
=> "#test"
```

time

```
(time expr)
```

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

```
(time (+ 100 200))
Elapsed time: 12.15 µs
=> 300
```

time/after?

```
(time/after? date1 date2)
```

Returns true if date1 is after date2 else false

```
(time/after? (time/local-date) (time/minus (time/local-date) :days 2))
=> true
```

time/before?

```
(time/before? date1 date2)
```

Returns true if date1 is before date2 else false

```
(time/before? (time/local-date) (time/minus (time/local-date) :days 2))
=> false
```

time/date

```
(time/date)
(time/date x)
```

Creates a new date. A date is represented by 'java.util.Date'

```
(time/date)
=> Wed Jun 03 10:26:00 CEST 2020
```

time/date?

```
(time/date? date)
```

```
Returns true if date is a date else false
```

```
(time/date? (time/date))
=> true
```

time/day-of-month

```
(time/day-of-month date)
```

```
Returns the day of the month (1..31)
```

```
(time/day-of-month (time/local-date))
=> 3
```

```
(time/day-of-month (time/local-date-time))
=> 3
```

```
(time/day-of-month (time/zoned-date-time))
=> 3
```

time/day-of-week

```
(time/day-of-week date)
```

```
Returns the day of the week (:MONDAY ... :SUNDAY)
```

```
(time/day-of-week (time/local-date))
=> :WEDNESDAY
```

```
(time/day-of-week (time/local-date-time))
=> :WEDNESDAY
```

```
(time/day-of-week (time/zoned-date-time))
=> :WEDNESDAY
```

time/day-of-year

```
(time/day-of-year date)
```

```
Returns the day of the year (1..366)
```

```
(time/day-of-year (time/local-date))
=> 155
```

```
(time/day-of-year (time/local-date-time))
=> 155

(time/day-of-year (time/zoned-date-time))
=> 155
```

top

time/earliest

```
(time/earliest coll)
```

Returns the earliest date from a collection of dates. All dates must be of equal type. The coll may be empty or
nil.

```
(time/earliest [(time/local-date 2018 8 4) (time/local-date 2018 8 3)])
=> 2018-08-03
```

top

time/first-day-of-month

```
(time/first-day-of-month date)
```

Returns the first day of a month as a local-date.

```
(time/first-day-of-month (time/local-date))
=> 2020-06-01

(time/first-day-of-month (time/local-date-time))
=> 2020-06-01

(time/first-day-of-month (time/zoned-date-time))
=> 2020-06-01
```

top

time/format

```
(time/format date format locale?)
(time/format date formatter locale?)
```

Formats a date with a format

```
(time/format (time/local-date) "dd-MM-yyyy")
=> "03-06-2020"

(time/format (time/zoned-date-time) "yyyy-MM-dd'T'HH:mm:ss.SSSz")
=> "2020-06-03T10:26:00.573CEST"

(time/format (time/zoned-date-time) :ISO_OFFSET_DATE_TIME)
=> "2020-06-03T10:26:00.577+02:00"

(time/format (time/zoned-date-time) (time/formatter "yyyy-MM-dd'T'HH:mm:ss.SSSz"))
=> "2020-06-03T10:26:00.580CEST"
```

```
(time/format (time/zoned-date-time) (time/formatter :ISO_OFFSET_DATE_TIME))
=> "2020-06-03T10:26:00.583+02:00"
```

time/formatter

```
(time/formatter format locale?)
```

Creates a formatter

```
(time/formatter "dd-MM-yyyy")
=> Value(DayOfMonth,2)'-'Value(MonthOfYear,2)'-'Value(YearOfEra,4,19,EXCEEDS_PAD)

(time/formatter "dd-MM-yyyy" :en_EN)
=> Value(DayOfMonth,2)'-'Value(MonthOfYear,2)'-'Value(YearOfEra,4,19,EXCEEDS_PAD)

(time/formatter "dd-MM-yyyy" "en_EN")
=> Value(DayOfMonth,2)'-'Value(MonthOfYear,2)'-'Value(YearOfEra,4,19,EXCEEDS_PAD)

(time/formatter "yyyy-MM-dd'T'HH:mm:ss.SSSz")
=> Value(YearOfEra,4,19,EXCEEDS_PAD)'-'Value(MonthOfYear,2)'-'Value(DayOfMonth,2)'T'Value(HourOfDay,2)':'Value
(MinuteOfHour,2)':'Value(SecondOfMinute,2)'.'Fraction(NanoOfSecond,3,3)ZoneText(SHORT)

(time/formatter :ISO_OFFSET_DATE_TIME)
=> ParseCaseSensitive(false)(ParseCaseSensitive(false)(Value(Year,4,10,EXCEEDS_PAD)'-'Value(MonthOfYear,
2)'-'Value(DayOfMonth,2))'T'(Value(HourOfDay,2)':'Value(MinuteOfHour,2)[':'Value(SecondOfMinute,2)[Fraction
(NanoOfSecond,0,9,DecimalPoint)]]))Offset(+HH:MM:ss,'Z')
```

time/hour

```
(time/hour date)
```

Returns the hour of the date 1..24

```
(time/hour (time/local-date))
=> 0

(time/hour (time/local-date-time))
=> 10

(time/hour (time/zoned-date-time))
=> 10
```

time/last-day-of-month

```
(time/last-day-of-month date)
```

Returns the last day of a month as a local-date.

```
(time/last-day-of-month (time/local-date))
=> 2020-06-30
```

```
(time/last-day-of-month (time/local-date-time))
=> 2020-06-30

(time/last-day-of-month (time/zoned-date-time))
=> 2020-06-30
```

time/latest

```
(time/latest coll)
```

Returns the latest date from a collection of dates. All dates must be of equal type. The coll may be empty or nil.

```
(time/latest [(time/local-date 2018 8 1) (time/local-date 2018 8 3)])
=> 2018-08-03
```

time/leap-year?

```
(time/leap-year? date)
```

Checks if the year is a leap year.

```
(time/leap-year? 2000)
=> true

(time/leap-year? (time/local-date 2000 1 1))
=> true

(time/leap-year? (time/local-date-time))
=> true

(time/leap-year? (time/zoned-date-time))
=> true
```

time/length-of-month

```
(time/length-of-month date)
```

Returns the length of the month represented by this date.

```
(time/length-of-month (time/local-date 2000 2 1))
=> 29

(time/length-of-month (time/local-date 2001 2 1))
=> 28

(time/length-of-month (time/local-date-time))
=> 30
```

```
(time/length-of-month (time/zoned-date-time))
=> 30
```

time/length-of-year

```
(time/length-of-year date)
```

Returns the length of the year represented by this date.

```
(time/length-of-year (time/local-date 2000 1 1))
=> 366

(time/length-of-year (time/local-date 2001 1 1))
=> 365

(time/length-of-year (time/local-date-time))
=> 366

(time/length-of-year (time/zoned-date-time))
=> 366
```

time/local-date

```
(time/local-date)
(time/local-date year month day)
(time/local-date date)
```

Creates a new local-date. A local-date is represented by 'java.time.LocalDate'

```
(time/local-date)
=> 2020-06-03

(time/local-date 2018 8 1)
=> 2018-08-01

(time/local-date "2018-08-01")
=> 2018-08-01

(time/local-date 1375315200000)
=> 2013-08-01

(time/local-date (. :java.util.Date :new))
=> 2020-06-03
```

time/local-date-parse

```
(time/local-date-parse str format locale?)
```

Parses a local-date.

```
(time/local-date-parse "2018-08-01" "yyyy-MM-dd")
=> 2018-08-01
```

time/local-date-time

```
(time/local-date-time)
(time/local-date-time year month day)
(time/local-date-time year month day hour minute second)
(time/local-date-time year month day hour minute second millis)
(time/local-date-time date)
```

Creates a new local-date-time. A local-date-time is represented by 'java.time.LocalDateTime'

```
(time/local-date-time)
=> 2020-06-03T10:26:00.252

(time/local-date-time 2018 8 1)
=> 2018-08-01T00:00

(time/local-date-time 2018 8 1 14 20 10)
=> 2018-08-01T14:20:10

(time/local-date-time 2018 8 1 14 20 10 200)
=> 2018-08-01T14:20:10.200

(time/local-date-time "2018-08-01T14:20:10.200")
=> 2018-08-01T14:20:10.200

(time/local-date-time 1375315200000)
=> 2013-08-01T02:00

(time/local-date-time (. :java.util.Date :new))
=> 2020-06-03T10:26:00.281
```

time/local-date-time-parse

```
(time/local-date-time-parse str format locale?
```

Parses a local-date-time.

```
(time/local-date-time-parse "2018-08-01 14:20" "yyyy-MM-dd HH:mm")
=> 2018-08-01T14:20

(time/local-date-time-parse "2018-08-01 14:20:01.000" "yyyy-MM-dd HH:mm:ss.SSS")
=> 2018-08-01T14:20:01
```

time/local-date-time?

```
(time/local-date-time? date)
```

Returns true if date is a local-date-time else false

```
(time/local-date-time? (time/local-date-time))
=> true
```

time/local-date?

```
(time/local-date? date)
```

Returns true if date is a locale date else false

```
(time/local-date? (time/local-date))
=> true
```

time/minus

```
(time/minus date unit n)
```

Subtracts the n units from the date. Units: {:years :months :weeks :days :hours :minutes :seconds :milliseconds}

```
(time/minus (time/local-date) :days 2)
=> 2020-06-01

(time/minus (time/local-date-time) :days 2)
=> 2020-06-01T10:26:00.637

(time/minus (time/zoned-date-time) :days 2)
=> 2020-06-01T10:26:00.640+02:00[Europe/Zurich]
```

time/minute

```
(time/minute date)
```

Returns the minute of the date 0..59

```
(time/minute (time/local-date))
=> 0

(time/minute (time/local-date-time))
=> 26

(time/minute (time/zoned-date-time))
=> 26
```

time/month

```
(time/month date)
```

Returns the month of the date 1..12

```
(time/month (time/local-date))
=> 6

(time/month (time/local-date-time))
=> 6

(time/month (time/zoned-date-time))
=> 6
```

time/not-after?

```
(time/not-after? date1 date2)
```

Returns true if date1 is not-after date2 else false

```
(time/not-after? (time/local-date) (time/minus (time/local-date) :days 2))
=> false
```

time/not-before?

```
(time/not-before? date1 date2)
```

Returns true if date1 is not-before date2 else false

```
(time/not-before? (time/local-date) (time/minus (time/local-date) :days 2))
=> true
```

time/period

```
(time/period from to unit)
```

Returns the period interval of two dates in the specified unit. Units: {:years :months :weeks :days :hours :
minutes :seconds :milliseconds}

```
(time/period (time/local-date) (time/plus (time/local-date) :days 3) :days)
=> 3

(time/period (time/local-date-time) (time/plus (time/local-date-time) :days 3) :days)
=> 3

(time/period (time/zoned-date-time) (time/plus (time/zoned-date-time) :days 3) :days)
=> 3
```

time/plus

```
(time/plus date unit n)
```

Adds the n units to the date. Units: {:years :months :weeks :days :hours :minutes :seconds :milliseconds}

```
(time/plus (time/local-date) :days 2)
=> 2020-06-05

(time/plus (time/local-date-time) :days 2)
=> 2020-06-05T10:26:00.628

(time/plus (time/zoned-date-time) :days 2)
=> 2020-06-05T10:26:00.631+02:00[Europe/Zurich]
```

time/second

```
(time/second date)
```

Returns the second of the date 0..59

```
(time/second (time/local-date))
=> 0

(time/second (time/local-date-time))
=> 0

(time/second (time/zoned-date-time))
=> 0
```

time/to-millis

```
(time/to-millis date)
```

Converts the passed date to milliseconds since epoch

```
(time/to-millis (time/local-date))
=> 1591135200000
```

time/with-time

```
(time/with-time date hour minute second)
(time/with-time date hour minute second millis)
```

Sets the time of a date. Returns a new date

```
(time/with-time (time/local-date) 22 00 15 333)
=> 2020-06-03T22:00:15.333
```

```
(time/with-time (time/local-date-time) 22 00 15 333)
=> 2020-06-03T22:00:15.333

(time/with-time (time/zoned-date-time) 22 00 15 333)
=> 2020-06-03T22:00:15.333+02:00[Europe/Zurich]
```

time/within?

```
(time/within? date start end)
```

Returns true if the date is after or equal to the start and is before or equal to the end. All three dates must be of the same type. The start and end date may each be nil meaning start is -infinity and end is +infinity.

```
(time/within? (time/local-date 2018 8 4) (time/local-date 2018 8 1) (time/local-date 2018 8 31))
=> true

(time/within? (time/local-date 2018 7 4) (time/local-date 2018 8 1) (time/local-date 2018 8 31))
=> false
```

time/year

```
(time/year date)
```

Returns the year of the date

```
(time/year (time/local-date))
=> 2020

(time/year (time/local-date-time))
=> 2020

(time/year (time/zoned-date-time))
=> 2020
```

time/zone

```
(time/zone date)
```

Returns the zone of the date

```
(time/zone (time/zoned-date-time))
=> :Europe/Zurich
```

time/zone-ids

```
(time/zone-ids)
```

Returns all available zone ids with time offset

```
(nfirst (seq (time/zone-ids)) 10)
=> ([:Africa/Abidjan "+00:00"] [:Africa/Accra "+00:00"] [:Africa/Addis_Ababa "+03:00"] [:Africa/Algiers "+01:
00"] [:Africa/Asmara "+03:00"] [:Africa/Asmera "+03:00"] [:Africa/Bamako "+00:00"] [:Africa/Bangui "+01:00"] [:
Africa/Banjul "+00:00"] [:Africa/Bissau "+00:00"])
```

time/zone-offset

```
(time/zone-offset date)
```

Returns the zone-offset of the date in minutes

```
(time/zone-offset (time/zoned-date-time))
=> 120
```

time/zoned-date-time

```
(time/zoned-date-time )
(time/zoned-date-time year month day)
(time/zoned-date-time year month day hour minute second)
(time/zoned-date-time year month day hour minute second millis)
(time/zoned-date-time date)
(time/zoned-date-time zone-id)
(time/zoned-date-time zone-id year month day)
(time/zoned-date-time zone-id year month day hour minute second)
(time/zoned-date-time zone-id year month day hour minute second millis)
(time/zoned-date-time zone-id date)
```

Creates a new zoned-date-time. A zoned-date-time is represented by 'java.time.ZonedDateTime'

```
(time/zoned-date-time)
=> 2020-06-03T10:26:00.301+02:00[Europe/Zurich]

(time/zoned-date-time 2018 8 1)
=> 2018-08-01T00:00+02:00[Europe/Zurich]

(time/zoned-date-time 2018 8 1 14 20 10)
=> 2018-08-01T14:20:10+02:00[Europe/Zurich]

(time/zoned-date-time 2018 8 1 14 20 10 200)
=> 2018-08-01T14:20:10.200+02:00[Europe/Zurich]

(time/zoned-date-time "2018-08-01T14:20:10.200+01:00")
=> 2018-08-01T14:20:10.200+01:00

(time/zoned-date-time 1375315200000)
=> 2013-08-01T02:00+02:00[Europe/Zurich]

(time/zoned-date-time (. :java.util.Date :new))
=> 2020-06-03T10:26:00.329+02:00[Europe/Zurich]

(time/zoned-date-time :UTC)
```

```
=> 2020-06-03T08:26:00.334Z[UTC]

(time/zoned-date-time :UTC 2018 8 1)
=> 2018-08-01T00:00Z[UTC]

(time/zoned-date-time :UTC 2018 8 1 14 20 10)
=> 2018-08-01T14:20:10Z[UTC]

(time/zoned-date-time :UTC 2018 8 1 14 20 10 200)
=> 2018-08-01T14:20:10.200Z[UTC]

(time/zoned-date-time :UTC "2018-08-01T14:20:10.200+01:00")
=> 2018-08-01T14:20:10.200Z[UTC]

(time/zoned-date-time :UTC 1375315200000)
=> 2013-08-01T00:00Z[UTC]

(time/zoned-date-time :UTC (. :java.util.Date :new))
=> 2020-06-03T08:26:00.362Z[UTC]
```

time/zoned-date-time-parse

```
(time/zoned-date-time-parse str format locale?)
```

```
Parses a zoned-date-time.
```

```
(time/zoned-date-time-parse "2018-08-01T14:20:01+01:00" "yyyy-MM-dd'T'HH:mm:ssz")
=> 2018-08-01T14:20:01+01:00

(time/zoned-date-time-parse "2018-08-01T14:20:01.000+01:00" "yyyy-MM-dd'T'HH:mm:ss.SSSz")
=> 2018-08-01T14:20:01+01:00

(time/zoned-date-time-parse "2018-08-01T14:20:01.000+01:00" :ISO_OFFSET_DATE_TIME)
=> 2018-08-01T14:20:01+01:00

(time/zoned-date-time-parse "2018-08-01 14:20:01.000 +01:00" "yyyy-MM-dd' 'HH:mm:ss.SSS' 'z")
=> 2018-08-01T14:20:01+01:00
```

time/zoned-date-time?

```
(time/zoned-date-time? date)
```

```
Returns true if date is a zoned-date-time else false
```

```
(time/zoned-date-time? (time/zoned-date-time))
=> true
```

to-degrees

```
(to-degrees x)
```

```
to-degrees x

(to-degrees 3)
=> 171.88733853924697

(to-degrees 3.1415926)
=> 179.99999692953102

(to-degrees 3.1415926M)
=> 179.99999692953102
```

to-radians

```
(to-radians x)
```

```
to-radians x
```

```
(to-radians 90)
=> 1.5707963267948966

(to-radians 90.0)
=> 1.5707963267948966

(to-radians 90.0M)
=> 1.5707963267948966
```

trampoline

```
(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.

transduce

```
(transduce xform f coll)
(transduce xform f init coll)
```

Reduce with a transformation of a reduction function f (xf). If init is not supplied, (f) will be called to
produce it. f should be a reducing step function that accepts both 1 and 2 arguments. Returns the result of
applying (the transformed) xf to init and the first item in coll, then applying xf to that result and the 2nd
item, etc. If coll contains no items, returns init and f is not called.

```
(do
  (def xform (map #(+ % 1)))
  (transduce xform + [1 2 3 4]))
=> 14

(do
  (def xform (map #(+ % 1)))
  (transduce xform conj [1 2 3 4]))
=> [2 3 4 5]

(do
  (def xform (comp (drop 2) (take 3)))
  (transduce xform conj [1 2 3 4 5 6]))
=> [3 4 5]

(do
  (def xform (comp
               (map #(* % 10))
               (map #(- % 5))
               (sorted compare)
               (drop 3)
               (take 2)
               (reverse)))
  (def coll [5 2 1 6 4 3])
  (str (transduce xform conj coll)))
=> "[45 35]"
```

true?

```
(true? x)
```

Returns true if x is true, false otherwise

```
(true? true)
=> true

(true? false)
=> false

(true? nil)
=> false

(true? 0)
=> false

(true? (== 1 1))
=> true
```

try

```
(try expr)
(try expr (catch exClass exSym expr))
(try expr (catch exClass exSym expr) (finally expr))
```

Exception handling: try - catch -finally

```
(try (throw))
=> JavaValueException: Venice value exception

(try
   (throw "test message"))
=> JavaValueException: Venice value exception

(try
   (throw 100)
   (catch :java.lang.Exception ex -100))
=> -100

(try
   (throw 100)
   (finally (println "...finally")))
...finally
=> JavaValueException: Venice value exception

(try
   (throw 100)
   (catch :java.lang.Exception ex -100)
   (finally (println "...finally")))
...finally
=> -100

(do
   (import :java.lang.RuntimeException)
   (try
      (throw (. :RuntimeException :new "message"))
      (catch :RuntimeException ex (:message ex))))

=> "message"

(do
   (try
      (throw [1 2 3])
      (catch :ValueException ex (str (:value ex)))
      (catch :RuntimeException ex "runtime ex")
      (finally (println "...finally"))))
...finally
=> "[1 2 3]"
```

try-with

```
(try-with [bindings*] expr)
(try-with [bindings*] expr (catch :java.lang.Exception ex expr))
(try-with [bindings*] expr (catch :java.lang.Exception ex expr) (finally expr))
```

try-with resources allows the declaration of resources to be used in a try block with the assurance that the resources will be closed after execution of that block. The resources declared must implement the Closeable or

```
(do
   (import :java.io.FileInputStream)
   (let [file (io/temp-file "test-", ".txt")]
        (io/spit file "123456789" :append true)
        (try-with [is (. :FileInputStream :new file)]
           (io/slurp-stream is :binary false))))
=> "123456789"
```

type

```
(type x)
```

Returns the type of x.

```
(type 5)
=> :core/long

(type [1 2])
=> :core/vector

(type (. :java.math.BigInteger :valueOf 100))
=> :java.math.BigInteger
```

union

```
(union s1)
(union s1 s2)
(union s1 s2 & sets)
```

Return a set that is the union of the input sets

```
(union (set 1 2 3))
=> #{1 2 3}

(union (set 1 2) (set 2 3))
=> #{1 2 3}

(union (set 1 2 3) (set 1 2) (set 1 4) (set 3))
=> #{1 2 3 4}
```

update

```
(update m k f)
```

Updates a value in an associative structure, where k is a key and f is a function that will take the old value
return the new value. Returns a new structure.

```
(update [] 0 (fn [x] 5))
=> [5]

(update [0 1 2] 0 (fn [x] 5))
=> [5 1 2]

(update [0 1 2] 0 (fn [x] (+ x 1)))
=> [1 1 2]

(update {} :a (fn [x] 5))
=> {:a 5}
```

```
(update {:a 0} :b (fn [x] 5))
=> {:a 0 :b 5}

(update {:a 0 :b 1} :a (fn [x] 5))
=> {:a 5 :b 1}
```

update!

```
(update! m k f)
```

Updates a value in a mutable map, where k is a key and f is a function that will take the old value return the new value.

```
(update! (mutable-map) :a (fn [x] 5))
=> {:a 5}

(update! (mutable-map :a 0) :b (fn [x] 5))
=> {:a 0 :b 5}

(update! (mutable-map :a 0 :b 1) :a (fn [x] 5))
=> {:a 5 :b 1}
```

update-in

```
(update-in [m 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.

```
(do
  (def users [ {:name "James" :age 26}
               {:name "John" :age 43} ] )
  (update-in users [1 :age] inc))
=> [{:age 26 :name "James"} {:age 44 :name "John"}]

(update-in {:a 12} [:a] / 4)
=> {:a 3}
```

uuid

```
(uuid)
```

Generates a UUID.

```
(uuid )
=> "63ff620e-663c-414a-bd01-32986cf7a500"
```

val

```
(val e)
```

Returns the val of the map entry.

```
(val (find {:a 1 :b 2} :b))
=> 2

(val (first (entries {:a 1 :b 2 :c 3})))
=> 1
```

vals

```
(vals map)
```

Returns a collection of the map's values.

```
(vals {:a 1 :b 2 :c 3})
=> (1 2 3)
```

var-get

```
(var-get sym)
```

Returns the var associated with the symbol

```
(var-get '+)
=> function + {visibility :public, ns "core"}
```

vary-meta

```
(vary-meta obj f & args)
```

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

```
(meta (vary-meta [1 2] assoc :a 1))
=> {:a 1 :line 1 :column 26 :file "example"}
```

vector

```
(vector & items)
```

Creates a new vector containing the items.

```
(vector )
=> []

(vector 1 2 3)
=> [1 2 3]

(vector 1 2 3 [:a :b])
=> [1 2 3 [:a :b]]
```

vector?

```
(vector? obj)
```

Returns true if obj is a vector

```
(vector? (vector 1 2))
=> true

(vector? [1 2])
=> true
```

version

```
(version)
```

Returns the version.

```
(version )
=> "0.0.0"
```

volatile

```
(volatile x)
```

Creates a volatile with the initial value x

```
(do
  (def counter (volatile 0))
  (swap! counter inc)
  (deref counter))
=> 1

(do
  (def counter (volatile 0))
  (reset! counter 9)
  @counter)
=> 9
```

volatile?

```
(volatile? x)
```

Returns true if x is a volatile, otherwise false

```
(do
   (def counter (volatile 0))
   (volatile? counter))
=> true
```

when

```
(when test & body)
```

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

```
(when (== 1 1) true)
=> true
```

when-let

```
(when-let bindings & body)
```

bindings is a vector with 2 elements: binding-form test.
If test is true, evaluates the body expressions with
binding-form bound to the value of test, if not, yields
nil

```
(when-let [value (* 100 2)]
   (str "The expression is true. value=" value))
=> "The expression is true. value=200"
```

when-not

```
(when-not test & body)
```

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

```
(when-not (== 1 2) true)
=> true
```

while

```
(while test & body)
```

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

```
(do
  (def a (atom 5))
  (while (pos? (deref a))
    (do (println (deref a)) (swap! a dec))))
5
4
3
2
1
=> nil
```

with-err-str

```
(with-err-str & forms)
```

Evaluates exprs in a context in which *err* is bound to a capturing
output stream. Returns the string created by any nested printing
calls.

```
(with-err-str (println *err* "a string"))
=> "a string\n"
```

with-meta

```
(with-meta obj m)
```

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

with-out-str

```
(with-out-str & forms)
```

Evaluates exprs in a context in which *out* is bound to a capturing
output stream. Returns the string created by any nested printing
calls.

```
(with-out-str (println "a string"))
=> "a string\n"
```

with-sh-dir

```
(with-sh-dir dir & forms)
```

Sets the directory for use with sh, see sh for details.

```
(with-sh-dir "/tmp" (sh "ls" "-l"))
```

with-sh-env

```
(with-sh-env env & forms)
```

Sets the environment for use with sh, see sh for details.

```
(with-sh-env {"NAME" "foo"} (sh "ls" "-l"))
```

with-sh-throw

```
(with-sh-throw forms)
```

If true throws an exception if the spawned shell process returns an
exit code other than 0. If false return the exit code. Defaults to
false. For use with sh, see sh for details.

```
(with-sh-throw (sh "ls" "-l"))
```

xml/children

```
(xml/children nodes)
```

Returns the children of the XML nodes collection

```
(do
  (load-module :xml)
  (xml/children
    (list (xml/parse-str "<a><b>B</b></a>"))))
=> ({:content ["B"] :tag "b"})
```

xml/parse

```
(xml/parse s)
(xml/parse s handler)
```

Parses and loads the XML from the source s with the parser
XMLHandler handler. The source may be an InputSource, an
InputStream, a File, or a string describing an URI.
Returns a tree of XML element maps with the
keys :tag, :attrs, and :content.

xml/parse-str

```
(xml/parse-str s)
(xml/parse-str s handler)
```

Parses an XML from the string s. Returns a tree of XML element
maps with the keys :tag, :attrs, and :content.

```
(do
  (load-module :xml)
  (xml/parse-str "<a><b>B</b></a>"))
=> {:content [{:content ["B"] :tag "b"}] :tag "a"}
```

xml/path->

```
(xml/path-> path nodes)
```

Applies the path to a node or a collection of nodes

```
(do
  (load-module :xml)
  (let [nodes (xml/parse-str "<a><b><c>C</c></b></a>")
        path [(xml/tag= "b")
              (xml/tag= "c")
              xml/text
              first]]
    (xml/path-> path nodes)))
=> "C"
```

xml/text

```
(xml/text nodes)
```

Returns a list of text contents of the XML nodes collection

```
(do
  (load-module :xml)
  (let [nodes (xml/parse-str "<a><b>B</b></a>")
        path [(xml/tag= "b")
              xml/text]]
    (xml/path-> path nodes)))
=> ("B")
```

zero?

```
(zero? x)
```

```
Returns true if x zero else false
```

```
(zero? 0)
=> true

(zero? 2)
=> false

(zero? (int 0))
=> true

(zero? 0.0)
=> true

(zero? 0.0M)
=> true
```

zipmap

```
(zipmap keys vals)
```

```
Returns a map with the keys mapped to the corresponding vals.
```

```
(zipmap [:a :b :c :d :e] [1 2 3 4 5])
=> {:a 1 :b 2 :c 3 :d 4 :e 5}

(zipmap [:a :b :c] [1 2 3 4 5])
=> {:a 1 :b 2 :c 3}
```

{}

```
Creates a hash map.
```

```
{:a 10 :b 20}
=> {:a 10 :b 20}
```