Venice 1.4.1 Cheat Sheet

Primitives

Literals

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

Numbers

Arithmetic	+ - * / mod inc dec min max abs negate sqrt
Convert	long double decimal
Compare	== != < > <= >= compare
Test	zero? pos? neg? even? odd? number? long? double? decimal?
Random	rand-long rand-double rand-gaussian
BigDecimal	dec/add dec/sub dec/mul dec/div dec/scale

Strings

Create	str str/format str/quote
Use	count compare empty-to-nil str/index-of str/last-index-of str/replace-first str/replace-last str/replace-all str/lower-case str/upper-case str/join str/subs str/split str/split-lines str/strip-start str/strip-end str/strip-indent str/strip-margin str/repeat str/truncate str/char
Regex	match match-not
Trim	str/trim str/trim-to-nil
Test	string? empty? str/blank? str/starts-with? str/ends-with? str/contains?

Other

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

Collections

Collections

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

Lists

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

Vectors

Create	[] vector mapv
Access	first second nth last peek butlast rest nfirst nlast subvec
Modify	cons conj rest pop into concat distinct dedupe partition interpose interleave mapcat flatten reverse sort sort-by take take-while drop drop-while assoc-in get-in update update! split-with
Test	contains? every? not-every? any? not-any?

Sets

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

Maps

Create	{} hash-map ordered-map sorted-map mutable-map zipmap

Byte Buffer

Create	bytebuf bytebuf-from-string
Test	empty? not-empty?

Access	find get keys vals key val
Modify	cons conj assoc assoc! assoc-in get-in update update! dissoc dissoc! into concat flatten reduce-kv merge
Test	contains?

Concurrency

Atoms	atom atom? deref reset! swap! compare-and-set! add-watch remove-watch
Futures	future future? future-done? future-cancel future-cancelled? deref realized?
Promises	promise promise? deliver realized?
Delay	delay delay? 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?
ThreadLocal	thread-local thread-local? thread-local-clear assoc dissoc get
Util	thread-id thread-name

System

Test	sandboxed? os-type?
Other	version os-type system-prop java-version uuid sleep gc current-time-millis nano-time coalesce
Shell	sh with-sh-dir with-sh-env with-sh-throw

Java Interoperability

General	. proxify Invoke constructors Invoke static or instance methods Access static or instance fields

IO

to	print println printf flush newline
to-str	pr-str
from	readline read-string
file-io	io/file io/file? io/file-parent io/file-name io/file-path io/exists-file? io/exists-dir? io/list-files io/file-size io/delete-file io/delete-file-on-

Branch	and or when when-not if-let
Loop	list-comp dotimes while
Call	doto -> ->> -<> as->
Loading	load-string load-file load-module load-classpath-file
Test	macro? cond case
Assert	assert
Util	comment gensym time
Profiling	time perf

Special Forms

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

	exit io/copy-file io/move-file io/mkdir io/mkdirs io/slurp io/slurp-lines io/spit io/tmp-dir io/user-dir
stream-io	io/slurp-stream io/spit-stream
file-io temp	io/temp-file io/slurp-temp-file io/spit-temp-file
other	io/load-classpath-resource io/mime-type io/default-charset with-out-str

Miscellaneous

JSON	json/pretty-print json/to-json json/to-pretty-json json/parse json/avail? json/avail-jdk8-module? Available if Jackson libs are on runtime classpath

Embedding in Java

Eval

```
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

```
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

```
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

```
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

```
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

!=

(!= x y)

Returns true if both operands do not have the equivalent type

```
(!= 0 1)
=> true

(!= 0 0)
=> false

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

#{}

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

(* 6.0 2)
=> 12.0

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

+

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

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

```
(+)
=> 0

(+ 1)
=> 1

(+ 1 2)
=> 3

(+ 1 2 3 4)
=> 10
```

-

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

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

```
(- 4)
=> -4

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

(- 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)
=> 2019-03-31T23:21:44.520+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
```

/

```
(/ 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

(/ 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 the equivalent type

(== 0 0)
=> true

(== 0 1)
=> false

(== 0 0.0)
=> false

>

(> 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 -10.1)
=> 10.1

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

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.
 :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
```

and

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

Ands the predicate forms

```
(and true true)
=> true

(and true false)
=> false
```

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) [1 5 10])
=> true
```

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"
```

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.

```
(as-> [:foo :bar] v
    (map name v)
    (first v)
    (str/subs v 1))
=> "oo"
```

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!

```
(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"}]
```

atom

(atom x)

Creates an atom with the initial value x

```
(do
  (def counter (atom 0))
  (deref counter))
=> 0
```

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 100))
  (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 100))
  (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? ))
```

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
100
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?

```
(boolean? n)
```

Returns true if n is a boolean

```
(boolean? true)
=> true

(boolean? false)
=> true

(boolean? nil)
=> false

(boolean? 0)
=> false
```

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)
```

bytebuf

(bytebuf x)

Converts 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-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-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 "f4" :file "example" :line 4 :col 18} {:fn-name "f3" :file "example" :line 3 :col 18} {:fn-name "f2" :file "example" :line 2 :col 18} {:fn-name "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
```

coalesce

(coalesce args*)

Returns the first non nil arg

```
(coalesce )
=> nil

(coalesce 1 2)
=> 1

(coalesce nil)
=> nil

(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
```

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"
```

conj

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

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

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

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

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

contains?

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

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

```
(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
```

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
```

current-time-millis

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

Returns the current time in milliseconds.

```
(current-time-millis)
=> 1554067304504
```

dec

(dec x)

Decrements the number x

(dec 10)
=> 9

(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
```

dedupe

(dedupe coll)

Returns a collection with all consecutive duplicates removed

```
(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)
=> 5

(def sum (fn [x y] (+ x y)))
=> fn anonymous-bebe0709-560e-40c4-9911-014d28c394ce
```

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 unless
```

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))
=> fn sum

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

defonce

```
(defonce name expr)
```

Creates a global variable that can not be overwritten

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

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 123))
=> nil
```

deref

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

Dereferences an atom or a Future 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.
Also reader macro: @atom/@future/@promise.

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

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

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

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

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

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

```
  @x)
working...
=> 100
```

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 coll x)
(disj coll 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}
```

distinct

```
(distinct coll)
```

Returns a collection with all duplicates removed

```
(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
```

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)

=> 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 performance test.

```
(do (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?

```
(double? n)
```

Returns true if n is a double

```
(double? 4.0)
=> true

(double? 3)
=> 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

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

```
(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
```

eval

```
(eval form)
```

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

```
(eval '(let [a 10] (+ 3 4 a)))
=> 17

(eval (list + 1 2 3))
=> 6
```

even?

```
(even? n)
```

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

```
(even? 4)
=> true

(even? 3)
=> false
```

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
```

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.

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

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.

```
(first nil)
=> nil

(first [])
=> nil

(first [1 2 3])
=> 1

(first '())
=> nil

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

flatten

```
(flatten coll)
```

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

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

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

flush

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

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

```
(flush)
=> nil
```

```
(flush *out*)
=> 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
```

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
```

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.

```
(do
  (def wait (fn [] (do (sleep 500) 100)))

  (let [f (future wait)]
       (deref f))
)
=> 100
```

future-cancel

(future-cancel f)

Cancels the future

```
(future-cancel (future (fn [] 100)))
=> java.util.concurrent.FutureTask@71be98f5
```

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

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

future?

(future? f)

Returns true if f is a Future otherwise false

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

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__3

(gensym "prefix_")
=> prefix_4

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]}
```

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
```

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)

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 value=" value))
=> "The expression is true. value=200"
```

inc

(inc x)

Increments the number x

(inc 10)
=> 11

(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? :venice.Long 500)
=> true

(instance? :java.math.BigInteger 500)
=> 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 to-coll from-coll)
```

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 [] {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"]

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

io/copy-file

```
(io/copy-file input output)
```

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

io/default-charset

(io/default-charset)

Returns the default charset.

io/delete-file

(io/delete-file f & files)

Deletes one or multiple files. f must be a java.io.File.

io/delete-file-on-exit

(io/delete-file-on-exit x)

Deletes a file on JVM exit. x must be a string or java.io.File.

io/exists-dir?

(io/exists-dir? x)

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

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

io/exists-file?

(io/exists-file? x)

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

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

io/file

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

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

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

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

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

io/file-name

(io/file-name f)

Returns the name of the file f. f must be a java.io.File.

```
(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 java.io.File.

```
(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. f must be a java.io.File.

```
(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 java.io.File.

```
(io/file-size (io/file "/bin/sh"))
=> 618480
```

io/file?

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

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

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

io/list-files

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

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

io/load-classpath-resource

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

Loads a classpath resource.

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 java.io.File.

io/mkdirs

(io/mkdirs dir)

Creates the directory including any necessary but nonexistent parent directories. dir must be a java.io.File.

io/move-file

(io/move-file source target)

Moves source to target. Returns nil or throws IOException. Source and target must be a java.io.File.

io/slurp

(io/slurp file & options)

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

io/slurp-lines

(io/slurp-lines file & options)

Read all lines from a text file.
Defaults encoding=UTF-8.
Options: :encoding "UTF-8

io/slurp-stream

(io/slurp-stream is & options)

Slurps binary or string data from an input stream. Supports the option :binary to either slurp binary or string data. For string data an optional encoding can be specified.
Options: :encoding "UTF-8" :binary true/false.

```
(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/slurp-temp-file

```
(io/slurp-temp-file file & options)
```

Slurps binary or string data from a previously created temp file. Supports the option :binary to either slurp binary or string data. For string data an optional encoding can be specified.
Ensures that the caller can only read back files he previously created, but no other files. This allows callers to work with files without escaping the sandbox. The file must have been created with io/temp_file.
Options: :encoding "UTF-8" :binary true/false.

```
(do
  (let [file (io/temp-file "test-", ".txt")]
       (io/spit-temp-file file "123456789" :append true)
       (io/slurp-temp-file file :binary false :remove true))
)
=> "123456789"
```

io/spit

```
(io/spit f content & options)
```

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

io/spit-stream

```
(io/spit-stream os content & options)
```

Writes content (string or bytebuf) to the output stream 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 :encoding "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/spit-temp-file

(io/spit-temp-file f content & options)

Spits binary or string data from to previously created temp file.
Ensures that the caller can only write to files he previously created, but no other files. This allows callers to work with files without escaping the sandbox. The file must have been created with io/temp_file.
Defaults to append=true and encoding=UTF-8.
Options: :append true/false, :encoding "UTF-8"

io/temp-file

(io/temp-file prefix suffix)

Creates an empty temp file with prefix and suffix.
io/temp-file with its companions io/spit-temp-file and io/slurp-temp-file provide safe file access in sandboxed environments.

```
(do
  (let [file (io/temp-file "test-", ".txt")]
    (io/spit-temp-file file "123456789" :append true)
    (io/slurp-temp-file 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/user-dir

(io/user-dir)

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

java-version

(java-version)

Returns the Jvav VM version.

(java-version)
=> "1.8.0_201"

json/avail-jdk8-module?

(json/avail-jdk8-module?)

Returns true if JSON jdk8 is available otherwise false

(json/avail-jdk8-module?)

json/avail?

(json/avail?)

Returns true if JSON is available (Jackson libs on classpath) otherwise false

(json/avail?)

json/parse

(json/parse s)

Parses a JSON string

(json/parse (json/to-json [{:a 100 :b 100}]))

json/pretty-print

(json/pretty-print json)

Pretty prints JSON

(json/pretty-print (json/to-json {:a 100 :b 100}))

json/to-json

(json/to-json val)

Converts the val to JSON

(json/to-json {:a 100 :b 100})

json/to-pretty-json

to-pretty-json val

Converts the val to pretty printed JSON

(json/to-pretty-json {:a 100 :b 100})

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.

(keep even? (range 1 4))
=> (false true false)

(keep (fn [x] (if (odd? x) x)) (range 4))
=> (1 3)

key

(key e)

Returns the key of the map entry.

(key (find {:a 1 :b 2} :b))
=> :b

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
```

let

(let [bindings*] exprs*)

Evaluates the expressions and binds the values to symbols to 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 test.

```
(list-comp [x (range 10)] x)
=> (0 1 2 3 4 5 6 7 8 9)

(list-comp [x (range 5)] (* x 2))
=> (0 2 4 6 8)

(list-comp [x (range 10) :when (odd? x)] x)
=> (1 3 5 7 9)

(list-comp [x (range 10) :when (odd? x)] (* x 2))
=> (2 6 10 14 18)

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

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/println "hello"))
```

```
test: hello
=> nil
```

load-file

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

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

load-module

```
(load-module s)
```

Loads a Venice predefined extension module.

```
(do
  (load-module :logger))
=> fn logger/to-string
```

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
```

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?

```
(long? n)
```

Returns true if n is a long

```
(long? 4)
=> true

(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

(macroexpand (-> c (+ 3) (* 2)))
=> (* (+ c 3) 2)

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.

(map inc [1 2 3 4])
=> (2 3 4 5)

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)
```

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]
```

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
```

match-not

```
(match-not s regex)
```

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

```
(match-not "1234" "[0-9]+")
=> false

(match-not "1234ss" "[0-9]+")
=> true
```

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 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
```

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: 6.40 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
```

meta

```
(meta obj)
```

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

min

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

Returns the smallest of the values

```
(min 1)
=> 1

(min 1 2)
=> 1

(min 4 3 2 1)
=> 1

(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

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

name

(name x)

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

(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)
=> 167769501450878

neg?

(neg? x)

Returns true if x smaller than zero else false

(neg? -3)
=> true

(neg? 3)
=> false

(neg? -3.2)
=> true

(neg? -3.2M)
=> true

negate

(negate x)

Negates x

(negate 10)
=> -10

(negate 1.23)
=> -1.23

(negate 1.23M)
=> -1.23M

newline

(newline)

Writes a platform-specific newline to *out*

(newline)

=> 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)

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)
```

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-empty?

(not-empty? x)

Returns true if x is not empty

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

(empty? [1 2])
=> false

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

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
```

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
```

number?

(number? n)

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

(number? 4.0M)
=> true

(number? 4.0)
=> true

(number? 3)
=> true

(number? true)
=> false

(number? "a")
=> false

odd?

(odd? n)

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

(odd? 3)
=> true

(odd? 4)
=> false

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-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
```

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"))
```

peek

```
(peek coll)
```

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

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

(peek [1 2 3 4])
=> 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 printed with (println (prof :data-formatted))

```
(perf (+ 100 200) 12000 1000)
=> nil
```

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]
```

pos?

```
(pos? x)
```

Returns true if x greater than zero else false

```
(pos? 3)
=> true

(pos? -3)
=> false

(pos? 3.2)
=> true

(pos? 3.2M)
=> true
```

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"
```

print

(print & xs)

Prints to stdout, with no args, prints the empty string. With one arg x, prints x.toString(). With more than one arg, prints the concatenation of the str values of the args with delimiter ' '. Returns nil.

```
(print [10 20 30])
[10 20 30]
=> nil
```

printf

(printf fmt & args)

Prints formatted output, as per format

```
(printf "%s: %d" "abc" 100)
abc: 100
=> nil
```

println

(println & xs)

Prints to stdout 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 str values of the args with delimiter ' '. Returns nil.

```
(do (println 200) (println [10 20 30]))
200
[10 20 30]
=> 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
```

```
  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))
  (def task (fn []
              (do
                (sleep 500)
                (deliver p 123))))

  (future task)
  (deref p))
=> 123
```

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})))
)
=> []
```

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.

(rand-double)
=> 0.5350116518817457

(rand-double 100.0)
=> 25.0407519478262

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.

(rand-gaussian)
=> 0.8278887782444729

(rand-gaussian 0.0 5.0)
=> 4.569192627021251

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.

(rand-long)
=> 3425181376646728514

(rand-long 100)
=> 60

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 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-string

```
(read-string x)
```

Reads from x

readline

```
(readline prompt)
```

Reads the next line from stdin. The function is sandboxed

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))))
true
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-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}
```

remove

```
(remove predicate coll)
```

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

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

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))
=> (5 7 5 3 0)

;; compare with repeat, which only calls the 'rand-long'
;; function once, repeating the value five times.
(repeat 5 (rand-long 11))
=> (1 1 1 1 1)
```

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! atom newval)
```

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

```
(do
  (def counter (atom 0))
  (reset! counter 99)
  (deref counter))
=> 99
```

rest

(rest coll)

Returns a collection with second to list element

```
(rest nil)
=> nil

(rest [])
=> []

(rest [1])
=> []

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

(rest '())
=> ()

(rest '(1))
=> ()

(rest '(1 2 3))
=> (2 3)
```

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

```
(reverse [1 2 3 4 5 6])
=> [6 5 4 3 2 1]
```

sandboxed?

(sandboxed?)

Returns true if there is a sandbox otherwise false

```
(sandboxed? )
=> false
```

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)
  (sleep 100)
  (deref x))
=> 105
```

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)
```

```
  (sleep 100)
  (deref x))
=> 105
```

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)
=> #{[1 2] 3}
```

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.
  :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)
```

```
(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"))

;; 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"))
```

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? ))
```

sleep

```
(sleep n)
```

Sleep for n milliseconds.

```
(sleep 30)
=> nil
```

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 (partial * -1) 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 count ["aaa" "bb" "c"])
=> ["c" "bb" "aaa"]

; reversed
(sort-by count (comp (partial * -1) 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 (partial * -1) compare) [[1 2] [3 4] [2 3]])
=> [[3 4] [2 3] [1 2]]

(sort-by (fn [x] (get x :rank)) [{:rank 2} {:rank 3} {:rank 1}])
=> [{:rank 1} {:rank 2} {:rank 3}]

; reversed
(sort-by (fn [x] (get x :rank)) (comp (partial * -1) compare) [{:rank 2} {:rank 3} {:rank 1}])
=> [{:rank 3} {:rank 2} {:rank 1}]
```

```
(sorted-set? (set 1))
=> false
```

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 10.23)
=> 3.1984371183438953

(sqrt 10.23M)
=> 3.198437118343895324557024650857783854007720947265625M
```

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/blank?

(str/blank? s)

True if s is blank.

```
(str/blank? nil)
=> true

(str/blank? "")
=> true

(str/blank? "  ")
=> true

(str/blank? "abc")
=> false
```

str/char

(str/char n)

Converts a number to a single char string.

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

str/contains?

(str/contains? s substr)

True if s contains with substr.

```
(str/contains? "abc"  "ab")
=> true
```

str/ends-with?

(str/ends-with? s substr)

True if s ends with substr.

```
(str/starts-with? "abc"  "bc")
=> false
```

str/format

(str/format format args*)

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

```
(str/format "%s: %d" "abc" 100)
=> "abc: 100"
```

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/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/lower-case

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

Converts s to lowercase

```
(str/lower-case "aBcDeF")
=> "abcdef"
```

str/quote

(str/quote str q)
(str/quote str start end)

Quotes a string.

```
(str/quote "abc" "-")
=> "-abc-"

(str/quote "abc" "<" ">")
=> "<abc>"
```

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

```
(str/replace-all "abcdefabc" "ab" "XYZ")
=> "XYZcdefXYZc"
```

str/replace-first

(str/replace-first s search replacement)

Replaces the first occurrance of search in s

```
(str/replace-first "abcdefabc" "ab" "XYZ")
=> "XYZdefabc"
```

str/replace-last

(str/replace-last s search replacement)

Replaces the last occurrance of search in s

(str/replace-last "abcdefabc" "ab" "XYZ")
=> "abcdefXYZ"

str/split

(str/split s regex)

Splits string on a regular expression.

(str/split "abc , def , ghi" "[*],[*]")
=> ("abc" "def" "ghi")

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 substr)
```

Trims leading and trailing spaces from s.

```
(str/trim " abc  ")
=> "abc"
```

str/trim-to-nil

```
(str/trim-to-nil s substr)
```

Trims leading and trailing spaces from s. Returns nil if the rewsulting string is empry

```
(str/trim "")
=> ""

(str/trim "    ")
=> ""

(str/trim nil)
=> nil

(str/trim " abc   ")
=> "abc"
```

str/truncate

```
(str/truncate s maxlen marker)
```

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

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

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

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

str/upper-case

(str/upper-case s)

Converts s to uppercase

```
(str/upper-case "aBcDeF")
=> "ABCDEF"
```

string?

(string? x)

Returns true if x is a string

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

(bytebuf? [1 2])
=> false

(bytebuf? 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]
```

swap!

(swap! atom f & args)

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

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

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-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.

```
(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.

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

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-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)
=> fn thread-local-clear
```

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: 11.57 us
=> 300
```

time/after?

```
(time/after? date1 date2)
```

Returns t6 mrEhe time it tookt date)

(time/date)
(time/date x)

Creates a new date. A date is represented by 'java.util.Date'

(time/date)
=> Sun Mar 31 23:21:40 CEST 2019

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))
=> 31

(time/day-of-month (time/local-date-time))
=> 31

(time/day-of-month (time/zoned-date-time))
=> 31

time/day-of-week

(time/day-of-week date)

Returns the day of the week (:MONDAY ... :SUNDAY)

(time/day-of-week (time/local-date))
=> :SUNDAY

(time/day-of-week (time/local-date-time))
=> :SUNDAY

(time/day-of-week (time/zoned-date-time))
=> :SUNDAY

time/day-of-year

(time/day-of-year date)

Returns the day of the year (1..366)

(time/day-of-year (time/local-date))
=> 90

(time/day-of-year (time/local-date-time))
=> 90

(time/day-of-year (time/zoned-date-time))
=> 90

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

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))
=> 2019-03-01

(time/first-day-of-month (time/local-date-time))
=> 2019-03-01

(time/first-day-of-month (time/zoned-date-time))
=> 2019-03-01

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")
=> "31-03-2019"

(time/format (time/zoned-date-time) "yyyy-MM-dd'T'HH:mm:ss.SSSz")
=> "2019-03-31T23:21:40.749CEST"

```
(time/format (time/zoned-date-time) :ISO_OFFSET_DATE_TIME)
=> "2019-03-31T23:21:40.752+02:00"

(time/format (time/zoned-date-time) (time/formatter "yyyy-MM-dd'T'HH:mm:ss.SSSz"))
=> "2019-03-31T23:21:40.754CEST"

(time/format (time/zoned-date-time) (time/formatter :ISO_OFFSET_DATE_TIME))
=> "2019-03-31T23:21:40.757+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))
=> 23

(time/hour (time/zoned-date-time))
=> 23
```

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))
=> 2019-03-31

(time/last-day-of-month (time/local-date-time))
=> 2019-03-31

(time/last-day-of-month (time/zoned-date-time))
=> 2019-03-31
```

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))
=> false

(time/leap-year? (time/zoned-date-time))
=> false
```

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))
=> 31
```

```
(time/length-of-month (time/zoned-date-time))
=> 31
```

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))
=> 365

(time/length-of-year (time/zoned-date-time))
=> 365
```

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)
=> 2019-03-31

(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))
=> 2019-03-31
```

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)
=> 2019-03-31T23:21:40.563

(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))
=> 2019-03-31T23:21:40.588
```

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)
=> 2019-03-29

(time/minus (time/local-date-time) :days 2)
=> 2019-03-29T23:21:40.798

(time/minus (time/zoned-date-time) :days 2)
=> 2019-03-29T23:21:40.800+01:00[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))
=> 21

(time/minute (time/zoned-date-time))
=> 21
```

time/month

(time/month date)

Returns the month of the date 1..12

```
(time/month (time/local-date))
=> 3
```

(time/month (time/local-date-time))
=> 3

(time/month (time/zoned-date-time))
=> 3

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)
=> 2019-04-02

(time/plus (time/local-date-time) :days 2)
=> 2019-04-02T23:21:40.791

(time/plus (time/zoned-date-time) :days 2)
=> 2019-04-02T23:21:40.794+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))
=> 40

(time/second (time/zoned-date-time))
=> 40

time/to-millis

(time/to-millis date)

Converts the passed date to milliseconds since epoch

(time/to-millis (time/local-date))
=> 1553986800000

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)
=> 2019-03-31T22:00:15.333

(time/with-time (time/local-date-time) 22 00 15 333)
=> 2019-03-31T22:00:15.333

(time/with-time (time/zoned-date-time) 22 00 15 333)
=> 2019-03-31T22:00:15.333+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))
=> 2019

(time/year (time/local-date-time))
=> 2019

(time/year (time/zoned-date-time))
=> 2019
```

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)
=> 2019-03-31T23:21:40.605+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))
=> 2019-03-31T23:21:40.626+02:00[Europe/Zurich]

(time/zoned-date-time :UTC)
=> 2019-03-31T21:21:40.629Z[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))
=> 2019-03-31T21:21:40.642Z[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
```

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)
=> :venice.Long

(type [1 2])
=> :venice.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}
```

uuid

```
(uuid)
```

Generates a UUID.

```
(uuid )
=> "eb13fb99-a610-44cd-ba0f-ec3b66e5762e"
```

val

```
(val e)
```

Returns the val of the map entry.

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

vals

(vals map)

Returns a collection of the map's values.

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

vary-meta

(vary-meta obj f & args)

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

vector

(vector & items)

Creates a new vector containing the items.

(vector)
=> []

(vector 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 )
=> "1.4.1"
```

when

(when test & body)

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

```
(when (== 1 1) true)
=> true
```

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-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"))

zero?

(zero? x)

Returns true if x zero else false

(zero? 0)
=> true

```
(zero? 2)
=> false

(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}
```