### Clojure Cheat Sheet (Clojure 1.4 - 1.7, sheet v32)

#### **Documentation**

clojure.repl/ doc find-doc apropos dir source pst javadoc (foo.bar/ is

namespace for later syms)

### **Primitives**

Numbers

Long: 7, hex Oxff, oct 017, base 2 2r1011, base 36 36rCRAZY
BigInt: 7N Ratio: -22/7 Double: 2.78 -1.2e-5 BigDecimal: 4.2M
+ - \* / quot rem mod inc dec max min +' -' \*' inc' dec' Literals Arithmetic

== < > <= >= compare Compare

Bitwise bit-and bit-or bit-xor bit-not bit-flip bit-set bit-shift-right

bit-shift-left bit-and-not bit-clear bit-test (1.6) unsigned-bit-shift-right (see BigInteger for integers larger

than Long)

Cast byte short int long float double bigdec bigint num rationalize

biginteger

Test zero? pos? neg? even? odd? number? rational? integer? ratio? decimal? float?

Random rand rand-int

BigDecimal with-precision Unchecked \*unchecked-math\* unchecked-add unchecked-dec unchecked-inc unchecked-multiply unchecked-negate unchecked-subtract

Strings

str format "a string" "escapes \b\f\n\t\r\" octal \377 hex \ucafe" Create

See also section IO/to string

Use count get subs compare (clojure.string/) join escape split split-lines replace replace-first reverse (1.5) re-quote-replacement (String)

.indexOf .lastIndexOf

Regex #"pattern" re-find re-seq re-matches re-pattern re-matcher re-groups

(clojure.string/) replace replace-first (1.5) re-quote-replacement
Note: \ in #"" is not escape char. (re-pattern "\\s\*\\d+") can be

written #"\s\*\d+"

(clojure.string/) capitalize lower-case upper-case Letters Trim (clojure.string/) trim trim-newline triml trimr

char char? string? (clojure.string/) blank? (String) .startsWith Test

.endsWith .contains

Other

char char-name-string char-escape-string literals: \a \newline Characters

(more at link)

Keywords keyword keyword? find-keyword literals: :kw :my.ns/kw ::in-cur-ns

Symbols symbol symbol? gensym literals: my-sym my.ns/foo

Misc literals: true false mil

### Collections

Collections Generic ops

count empty not-empty into conj (clojure.walk/) walk prewalk prewalk-demo prewalk-replace postwalk postwalk-demo

postwalk-replace

Content tests distinct? empty? every? not-every? some not-any? Capabilities sequential? associative? sorted? counted? reversible? Type tests coll? list? vector? set? map? seq? (1.6) record?

Lists (conj, pop, & peek at beginning)

() list list\* Create

Examine  $\verb|first nth peek .indexOf .lastIndexOf| \\$ 

'Change cons conj rest pop

Vectors (conj, pop, & peek at end)

[] vector vec vector-of mapv filterv (clojure.core.rrb-vector/) vector Create

vec vector-of

Examine  $(my\text{-vec idx}) \rightarrow (\text{nth my-vec idx}) \text{ get peek .indexOf .lastIndexOf}$ assoc pop subvec replace conj rseq update-in (1.7) update 'Change

Ops reduce-kv

Create unsorted #{} set hash-set (clojure.data.int-map/) int-set dense-int-set

Create sorted sorted-set sorted-set-by (clojure.data.avl/) sorted-set

sorted-set-by (flatland.ordered.set/) ordered-set  $(my\text{-set item}) \rightarrow (\text{get my-set item}) \text{ contains}?$ 

'Change' conj disj

(clojure.set/) union difference intersection select See also sec-Set ops

tion Relations

Test (clojure.set/) subset? superset?

Sorted sets rseq subseq rsubseq

Maps

Examine

Examine

Create unsorted {} hash-map array-map zipmap bean frequencies group-by (clo-

jure.set/) index (clojure.data.int-map/) int-map

sorted-map sorted-map-by (clojure.data.avl/) sorted-map Create sorted sorted-map-by (flatland.ordered.map/) ordered-map

ordering-map

(clojure.data.priority-map/) priority-map (flatland.useful.map/)  $\texttt{(my-map k)} \xrightarrow{\text{$k$}} \texttt{(get my-map k) also (:key my-map)} \xrightarrow{\text{$k$}} \texttt{(get my-map k)}$ 

my-map :key) get-in contains? find keys vals

assoc assoc-in dissoc merge merge-with select-keys update-in 'Change (1.7) update (clojure.set/) rename-keys map-invert GitHub:

Medley reduce-kv Ops Entry key val

Sorted maps rseq subseq rsubseq Queues (conj at end, peek & pop from beginning)

clojure.lang.PersistentQueue/EMPTY (no literal syntax or Create

constructor fn)

Examine peek 'Change' conj pop

Relations (set of maps, each with same keys, aka rels)

(clojure.set/) join select project union difference intersection index rename

Transients (clojure.org/transients)

Create transient persistent! Change conj! pop! assoc! dissoc! disj! Note: always use return value for later

changes, never original!

Misc = identical? not= not compare clojure.data/diff Compare

true? false? instance? nil? (1.6) some? Test

#### Sequences

Rel algebra

Creating a Lazy Seq

From collection seq vals keys rseq subseq rsubseq sequence

From producer fn lazy-seq repeatedly iterate

From constant repeat range

From other file-seq line-seq resultset-seq re-seq tree-seq xml-seq

iterator-seq enumeration-seq

From seq keep keep-indexed

Seq in, Seq out Get shorter

distinct filter remove take-nth for

Get longer cons conj concat lazy-cat mapcat cycle interleave interpose Tail-items rest nthrest next fnext nnext drop drop-while take-last for

Head-items take take-while butlast drop-last for 'Change'

conj concat distinct flatten group-by partition partition-all

partition-by split-at split-with filter remove replace shuffle

reverse sort sort-by compare Rearrange

Process items map pmap map-indexed mapcat for replace seque

Using a Seq

Extract item first second last rest next ffirst nfirst fnext nnext nth

nthnext rand-nth when-first max-key min-key

Construct coll zipmap into reduce reductions set vec into-array to-array-2d mapv filterv Pass to fn apply

some filter Search Force evaluation doseq dorun doall (1.7) run!

Check for forced realized?

Transducers (clojure.org/transducers)

Off the shelf map mapcat filter remove take take-while take-nth drop drop-while replace partition-by partition-all keep keep-indexed map-indexed distinct interpose (1.7) cat

dedupe random-sample

Create your own (1.7) completing ensure-reduced unreduced See also section Con-

currency/Volatiles

into sequence (1.7) transduce eduction Use

Early termination reduced reduced? deref

Zippers (clojure.zip/)

Create zipper seq-zip vector-zip xml-zip Get loc up down left right leftmost rightmost

Get seq

lefts rights path children

'Change make-node replace edit insert-child insert-left insert-right

append-child remove

Move next prev root node branch? end? Misc

10 to/from

spit slurp (to writer/from reader, Socket, string with file name, URI, etc.)

to \*out\* pr prn print printf println newline (clojure.pprint/) print-table to writer (clojure.pprint/) pprint cl-format also: (binding [\*out\* writer]

to string format with-out-str pr-str prn-str print-str println-str from \*in\* read-line (clojure.tools.reader.edn/) read

 ${\tt line-seq~(clojure.tools.reader.edn/)~read~also:~(binding~[*in*]$ reader] ...) java.io.Reader

with-in-str (clojure.tools.reader.edn/) read-string from string Open with-open (clojure.java.io/) text: reader writer binary: input-stream

Binary (.write ostream byte-arr) (.read istream byte-arr) java.io.OutputStream java.io.InputStream GitHub: gloss

byte-spec

flush (.close s) file-seq \*in\* \*out\* \*err\* (clojure.java.io/) Misc file copy delete-file resource as-file as-url as-relative-path

GitHub: fs

Data readers \*data-readers\* default-data-readers (1.5)

\*default-data-reader-fn\*

**Functions** 

from reader

Create fn defn defn- definline identity constantly memfn comp complement

partial juxt memoize fnil every-pred some-fn

apply -> Call ->> trampoline (1.5) as-> cond-> cond->> some->>

Test fn? ifn?

#### Pair. ->Pair map->Pair Create Test record? Types (clojure.org/datatypes) Define ( deftype Pair [h t]) Access (.h (Pair. 1 2)) $\rightarrow$ 1 Create Pair. ->Pair ( deftype Pair [h t] With methods Object (toString [this] (str "<" h "," t ">"))) Multimethods (clojure.org/multimethods) ( defmulti my-mm dispatch-fn) Method define ( defmethod my-mm :dispatch-value [args] ...) get-method methods Dispatch Remove remove-method remove-all-methods prefer-method prefers Prefer Relation derive underive isa? parents ancestors descendants make-hierarchy Macros Create defmacro definline Debug ${\tt macroexpand-1\ macroexpand\ (clojure.walk/)\ macroexpand-all}$ Branch and or when when-not when-let when-first if-not if-let cond condp case (1.6) when-some if-some Loop for doseq dotimes while .. doto -> ->> (1.5) as-> cond-> cond->> some-> some->> Arrange Scope binding locking time with-in-str with-local-vars with-open with-out-str with-precision with-redefs with-redefs-fn lazy-cat lazy-seq delay Lazy Doc assert comment doc Special Characters (clojure.org/reader, tutorial) Comma reads as white space. Often used between map key/value pairs for readability. quote: 'form $\rightarrow$ ( quote form) Namespace separator (see Primitives/Other section) Character literal (see Primitives/Other section) Keyword (see Primitives/Other section)

Abstractions (Clojure type selection flowchart)

satisfies? extends?

( defrecord Pair [h t])

 $(:h (Pair. 1 2)) \rightarrow 1$ 

( defprotocol Slicey (slice [at]))

( reify Slicey (slice [at] ...))

extend extend-protocol extenders

extend-type String Slicey (slice [at] ...))

( extend-type nil Slicey (slice [\_] nil))

Protocols (clojure.org/protocols)

Records (clojure.org/datatypes)

Define

Extend Extend null

Reify

Test

Other

Define

Access

```
Single line comment
         Metadata (see Metadata section)
         'earmuffs' - convention to indicate dynamic vars, compiler warns if
*foo*
         not dynamic
         Deref: @form \rightarrow (deref form)
         Syntax-quote
         'auto-gensym', consistently replaced with same auto-generated symbol
foo#
         everywhere inside same '( ... )
         Unquote
~@
         Unquote-splicing
          thread first' macro ->
         'thread last' macro ->>
         List literal (see Collections/Lists section)
         Vector literal (see Collections/Vectors section)
         Map literal (see Collections/Maps section)
#'
         Var-quote #'x \rightarrow (var x)
         #"p" reads as regex pattern p (see Strings/Regex section)
         Set literal (see Collections/Sets section)
#{
         Anonymous function literal: \#(...) \rightarrow (fn [args] (...))
%
         Anonymous function argument: %N is value of anonymous function arg N.
         \% short for %1. %& for rest args.
         (1.7) Reader conditional: #?(:clj x :cljs y) reads as x on JVM, y in
#?
         ClojureScript, nothing elsewhere. Other keys: :cljr :default
         (1.7) Splicing reader conditional: [1 #?@(:clj [x y] :cljs [w z])
#?@
         3] reads as [1 x y 3] on JVM, [1 w z 3] in ClojureScript, [1 3]
         elsewhere.
          tagged literal e.g. #inst #uuid
#foo
         JavaContainerClass$InnerClass
foo?
         conventional ending for a predicate, e.g.: zero? vector? instance?
         (unenforced)
         conventional ending for an unsafe operation, e.g.: set! swap!
foo!
         alter-meta! (unenforced)
          conventional name for an unused value (unenforced)
#
         Ignore next form
```

## Metadata (clojure.org/reader, special\_forms)

```
General ^{:key1 val1 :key2 val2 ...}

Abbrevs ^Type → ^{:tag Type}, ^:key → ^{:key true}

Common ^:dynamic ^:private ^:doc ^:const

Examples (defn ^:private ^String my-fn ...) (def ^:dynamic *dyn-var* val)

On Vars meta with-meta vary-meta alter-meta! reset-meta! doc find-doc test
```

### Special Forms (clojure.org/special\_forms)

def if do let letfn quote var fn loop recur set! throw try monitor-enter
monitor-exit

Binding Forms / (examples) let fn defn defnacro loop for doseq if-let
Destructuring when-let (1.6) if-some when-some

### Vars and global environment (clojure.org/vars)

Def variants def defn defn- definline defmacro defmethod defmulti defonce defrecord

Interned vars declare intern binding find-var var

Var objects with-local-vars var-get var-set alter-var-root var? bound? thread-bound?

Var validators set-validator! get-validator

#### Namespace

Current \*ns\*
Create/Switch (tutorial) ns in-ns create-ns
Add alias def import intern refer
Find all-ns find-ns
Examine ns-name ns-aliases ns-map ns-interns ns-publics ns-refers
ns-imports
From symbol resolve ns-resolve namespace the-ns
Remove ns-unalias ns-unmap remove-ns

#### Loading

Load libs (tutorial) require use import refer
List loaded loaded-libs
Load misc load load-file load-reader load-string

#### Concurrency

Atoms atom swap! reset! compare-and-set!

Futures future-call future-done? future-cancel future-cancelled?

future?

Threads bound-fn bound-fn\* get-thread-bindings push-thread-bindings pop-thread-bindings thread-bound?

Volatiles (1.7) volatile! vreset! vswap! volatile?

Misc locking pcalls pvalues pmap seque promise deliver

### Refs and Transactions (clojure.org/refs)

### Agents and Asynchronous Actions (clojure.org/agents)

Create agent agent-error Examine send send-off restart-agent (1.5) send-via Change state set-agent-send-executor! set-agent-send-off-executor! Block waiting await await-for Ref validators set-validator! get-validator Watchers add-watch remove-watch Thread handling shutdown-agents Error error-handler set-error-handler! error-mode set-error-mode! Misc \*agent\* release-pending-sends

### Java Interoperation (clojure.org/java\_interop)

General .. doto Classname/ Classname. new bean comparator enumeration-seq import iterator-seq memfn set! class class? bases supers type gen-class gen-interface definiterface

Cast boolean byte short char int long float double bigdec bigint num cast biginteger

Exceptions throw try catch finally pst ex-info ex-data

### Arrays

Create make-array object-array boolean-array byte-array short-array char-array int-array long-array float-array double-array aclone to-array to-array-2d into-array

Use aget aset-boolean aset-byte aset-short aset-char aset-int aset-long aset-float aset-double alength amap areduce

Cast booleans bytes shorts chars ints longs floats doubles

### Proxy (Clojure type selection flowchart)

Create proxy get-proxy-class construct-proxy init-proxy
Misc proxy-mappings proxy-super update-proxy

# Other