Don’t fear the (((())))()()()

… it’s not all that bad

Cosmin Cremarenco - @ccosmin - github.com/ccosmin

Agenda

- Clojure origins
- (Really) short introduction to the language
- Clojure collections
- A few words about Ring
- Luminus

Ready for battle?

Some introductory stuff

- Invented by Rich Hickey in 2007
    - https://www.youtube.com/watch?v=P76Vbsk_3J0
- Clojure is a modern Lisp
- Clojure is a functional language
    - Functions are first-class objects
    - Data is immutable
    - Functions are pure
- Dynamic language
- Embraces Java
- Homoiconicity
- Debugging code from 60 million miles away
    - https://www.youtube.com/watch?v=_gZK0tW8EhQ

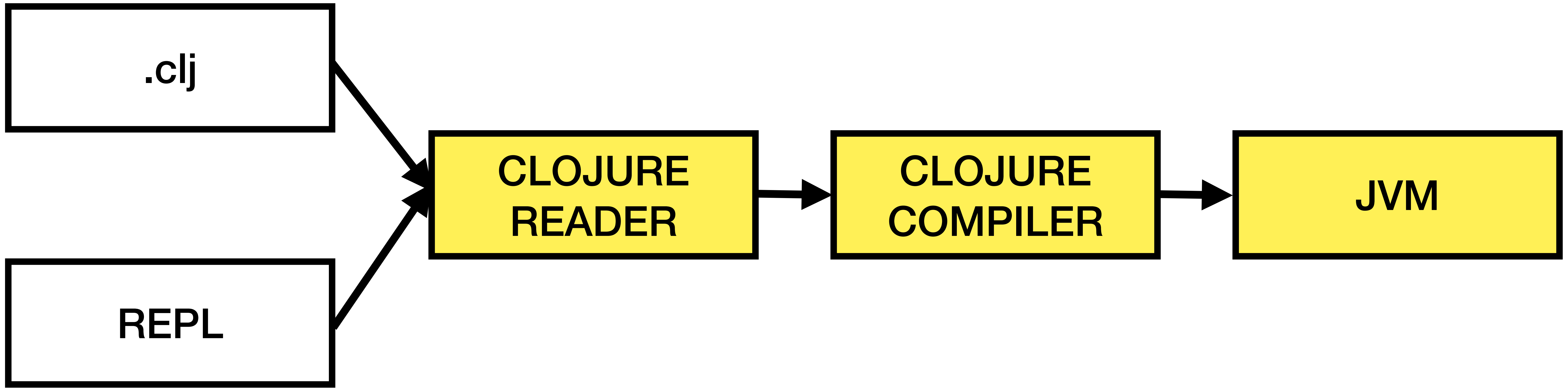
Rich Hickey

The thin red line

```clojure
(defn _ [time]
  (let [now (java.util.Date. (System/currentTimeMillis))
        diff-in-millis (- (.getTime now) (.getTime time))
        diff (quot diff-in-millis 1000)
        units [{:name "second" :limit 60 :in-second 1}
               {:name "minute" :limit 3600 :in-second 60}
               {:name "hour" :limit 86400 :in-second 3600}
               {:name "day" :limit 604800 :in-second 86400}
               {:name "week" :limit 2629743 :in-second 604800}
               {:name "month" :limit 31556926 :in-second 2629743}
               {:name "year" :limit Long/MAX_VALUE :in-second 31556926}]]
    (if (< diff 5)
      "just now"
      (let [unit (first (drop-while #(or (>= diff (:limit %))
                                         (not (:limit %)))
                                    units))]
        (-> (/ diff (:in-second unit))
            Math/floor
            int
            (#(str % " " (:name unit) (when (> % 1) "s") " ago")))))))
```

From https://stackoverflow.com/questions/32511405/how-would-time-ago-function-implementation-look-like-in-clojure

Evaluation



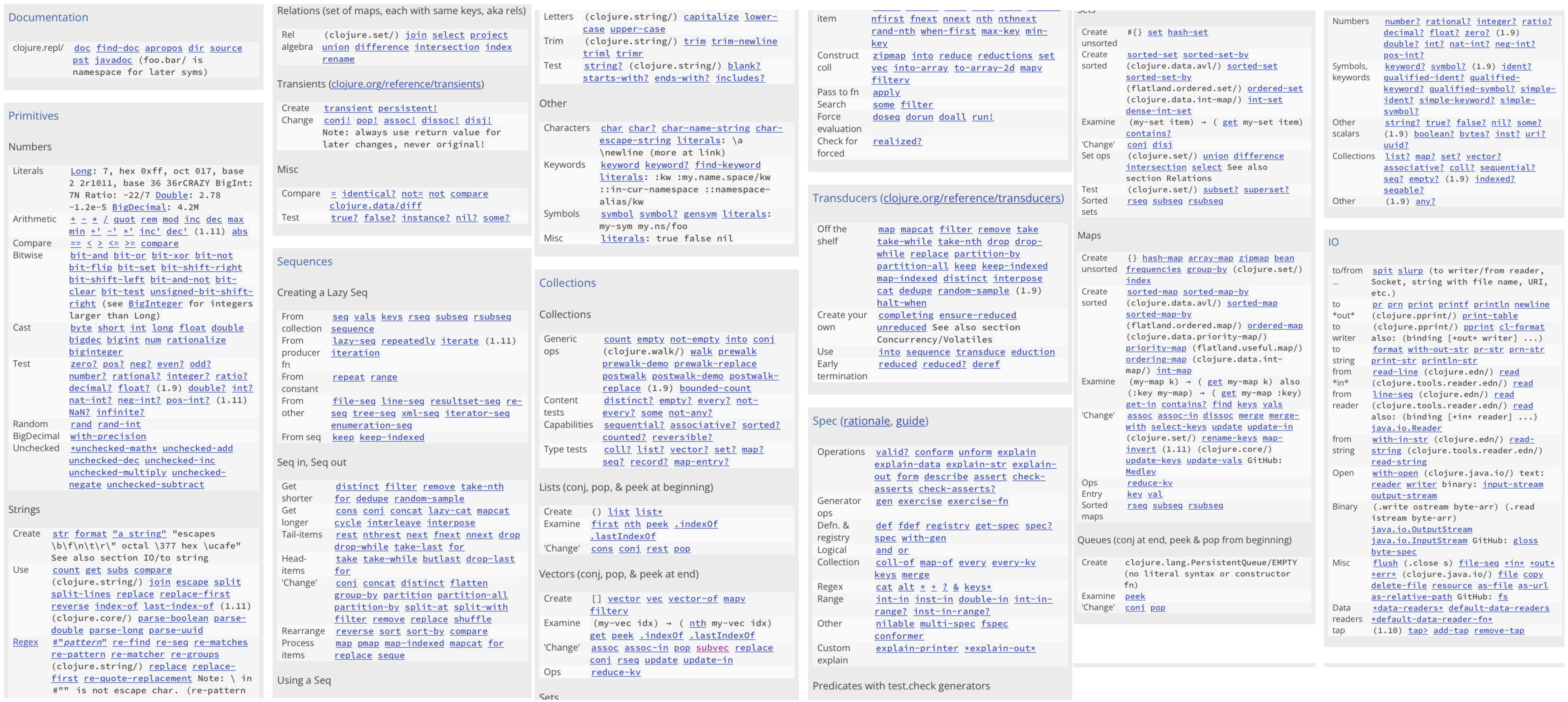
The Language

- Primitive types
    - numeric: 42, -3.4, 1/3
    - character: "hello world", \c, \newline, #”[0-9]*”
- Symbols
    - first, map, reduce
    - + (most punctuation)
    - clojure.core/str (namespaced symbol)
    - nil
    - true false
    - :vega (keywords)
    - mynamespace/:vega (namespaced keyword)

Functions

- Functions are declared using the fn symbol and assigned to a var using defn
- A function call is simply a list whose first element is evaluated to a function
    - +
    - str
    - filter, map, reduce
- No “return”: last expression is the value returned by the function

A lot of functions…



Documentation

clojure.repl/	doc find-doc apropos dir source pst javadoc (foo.bar/ is namespace for later syms)

Primitives

Numbers

Literals	Long: 7, hex 0xff, oct 017, base 2 2r1011, base 36 36rCRAZY BigInt: 7N Ratio: -22/7 Double: 2.78 -1.2e-5 BigDecimal: 4.2M
Arithmetic	+ - * / quot rem mod inc dec max min +' -' *' inc' dec' (1.11) abs
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 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? (1.9) double? int? nat-int? neg-int? pos-int? (1.11) NaN? infinite?
Random	rand rand-int
BigDecimal	with-precision
Unchecked	*unchecked-math* unchecked-add unchecked-dec unchecked-inc unchecked-multiply unchecked-negate unchecked-subtract

Strings

Create	str format "a string" "escapes \b\f\n\t\r\" octal \377 hex \ucafe" See also section IO/to string
Use	count get subs compare (clojure.string/) join escape split split-lines replace replace-first reverse index-of last-index-of (1.11) (clojure.core/) parse-boolean parse-double parse-long parse-uuid
Regex	#"*pattern*" re-find re-seq re-matches re-pattern re-matcher re-groups (clojure.string/) replace replace-first re-quote-replacement Note: \ in #"" is not escape char. (re-pattern

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

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

Transients (clojure.org/reference/transients)

Create	transient persistent!
Change	conj! pop! assoc! dissoc! disj! Note: always use return value for later changes, never original!

Misc

Compare	= identical? not= not compare clojure.data/diff
Test	true? false? instance? nil? some?

Sequences

Creating a Lazy Seq

From collection	seq vals keys rseq subseq rsubseq sequence
From producer fn	lazy-seq repeatedly iterate (1.11) iteration
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 dedupe random-sample
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
Rearrange	reverse sort sort-by compare
Process items	map pmap map-indexed mapcat for replace seque

Using a Seq

Letters	(clojure.string/) capitalize lower-case upper-case
Trim	(clojure.string/) trim trim-newline triml trimr
Test	string? (clojure.string/) blank? starts-with? ends-with? includes?

Other

Characters	char char? char-name-string char-escape-string literals: \a \newline (more at link)
Keywords	keyword keyword? find-keyword literals: :kw :my.name.space/kw ::in-cur-namespace ::namespace-alias/kw
Symbols	symbol symbol? gensym literals: my-sym my.ns/foo
Misc	literals: true false nil

Collections

Collections

Generic ops	count empty not-empty into conj (clojure.walk/) walk prewalk prewalk-demo prewalk-replace postwalk postwalk-demo postwalk-replace (1.9) bounded-count
Content tests	distinct? empty? every? not-every? some not-any?
Capabilities	sequential? associative? sorted? counted? reversible?
Type tests	coll? list? vector? set? map? seq? record? map-entry?

Lists (conj, pop, & peek at beginning)

Create	() list list*
Examine	first nth peek .indexOf .lastIndexOf
'Change'	cons conj rest pop

Vectors (conj, pop, & peek at end)

Create	[] vector vec vector-of mapv filterv
Examine	(my-vec idx) → (nth my-vec idx) get peek .indexOf .lastIndexOf
'Change'	assoc assoc-in pop subvec replace conj rseq update update-in
Ops	reduce-kv

Sets

item	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
Search	some filter
Force evaluation	doseq dorun doall run!
Check for forced	realized?

Transducers (clojure.org/reference/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 cat dedupe random-sample (1.9) halt-when
Create your own	completing ensure-reduced unreduced See also section Concurrency/Volatiles
Use	into sequence transduce eduction
Early termination	reduced reduced? deref

Spec (rationale, guide)

Operations	valid? conform unform explain explain-data explain-str explain-out form describe assert check-asserts check-asserts?
Generator ops	gen exercise exercise-fn
Defn. & registry	def fdef registry get-spec spec? spec with-gen
Logical	and or
Collection	coll-of map-of every every-kv keys merge
Regex	cat alt * + ? & keys*
Range	int-in inst-in double-in int-in-range? inst-in-range?
Other	nilable multi-spec fspec conformer
Custom explain	explain-printer *explain-out*

Predicates with test.check generators

Create unsorted	#{} set hash-set
Create sorted	sorted-set sorted-set-by (clojure.data.avl/) sorted-set sorted-set-by (flatland.ordered.set/) ordered-set (clojure.data.int-map/) int-set dense-int-set
Examine	(my-set item) → (get my-set item) contains?
'Change'	conj disj
Set ops	(clojure.set/) union difference intersection select See also section Relations
Test	(clojure.set/) subset? superset?
Sorted sets	rseq subseq rsubseq

Maps

Create unsorted	{} hash-map array-map zipmap bean frequencies group-by (clojure.set/) index
Create sorted	sorted-map sorted-map-by (clojure.data.avl/) sorted-map sorted-map-by (flatland.ordered.map/) ordered-map (clojure.data.priority-map/) priority-map (flatland.useful.map/) ordering-map (clojure.data.int-map/) int-map
Examine	(my-map k) → (get my-map k) also (:key my-map) → (get my-map :key) get-in contains? find keys vals
'Change'	assoc assoc-in dissoc merge merge-with select-keys update update-in (clojure.set/) rename-keys map-invert (1.11) (clojure.core/) update-keys update-vals GitHub: Medley
Ops	reduce-kv
Entry	key val
Sorted maps	rseq subseq rsubseq

Queues (conj at end, peek & pop from beginning)

Create	clojure.lang.PersistentQueue/EMPTY (no literal syntax or constructor fn)
Examine	peek
'Change'	conj pop

Numbers	number? rational? integer? ratio? decimal? float? zero? (1.9) double? int? nat-int? neg-int? pos-int?
Symbols, keywords	keyword? symbol? (1.9) ident? qualified-ident? qualified-keyword? qualified-symbol? simple-ident? simple-keyword? simple-symbol?
Other scalars	string? true? false? nil? some? (1.9) boolean? bytes? inst? uri? uuid?
Collections	list? map? set? vector? associative? coll? sequential? seq? empty? (1.9) indexed? seqable?
Other	(1.9) any?

IO

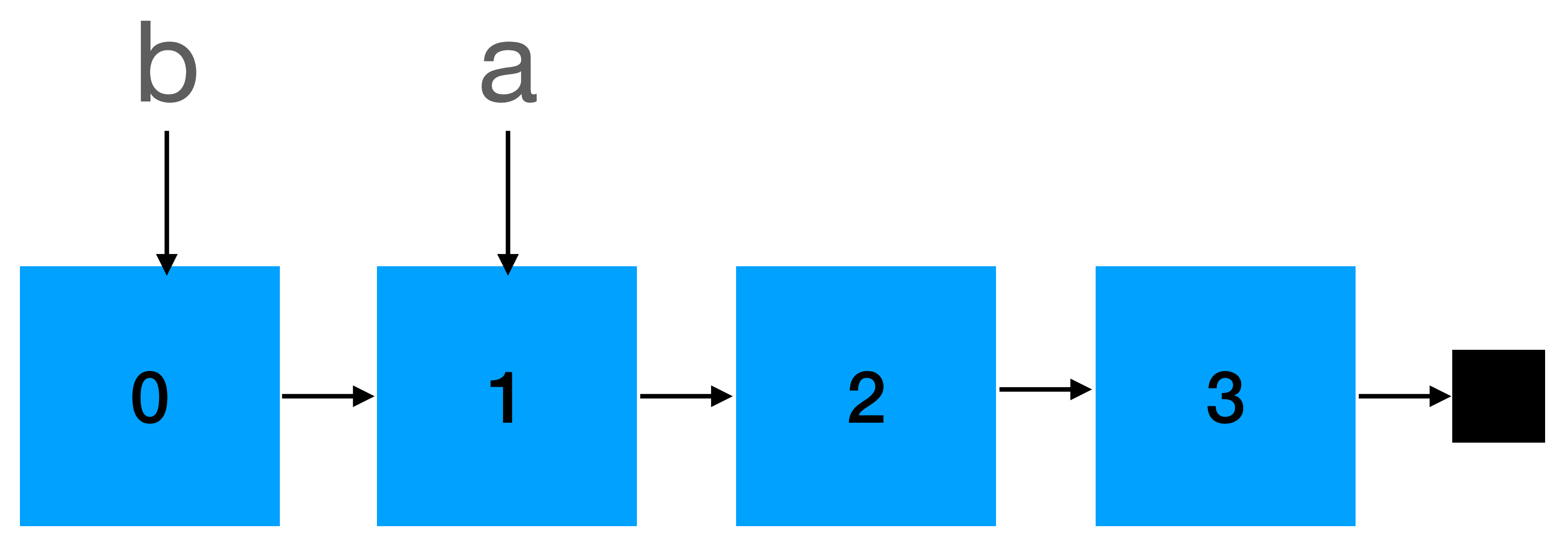
to/from ...	spit slurp (to writer/from reader, Socket, string with file name, URI, etc.)
to *out*	pr prn print printf println newline (clojure.pprint/) print-table
to writer	(clojure.pprint/) pprint cl-format also: (binding [*out* writer] ...)
to string	format with-out-str pr-str prn-str print-str println-str
from *in*	read-line (clojure.edn/) read (clojure.tools.reader.edn/) read
from reader	line-seq (clojure.edn/) read (clojure.tools.reader.edn/) read also: (binding [*in* reader] ...) java.io.Reader
from string	with-in-str (clojure.edn/) read-string (clojure.tools.reader.edn/) read-string
Open	with-open (clojure.java.io/) text: reader writer binary: input-stream output-stream
Binary	(.write ostream byte-arr) (.read istream byte-arr) java.io.OutputStream java.io.InputStream GitHub: gloss byte-spec
Misc	flush (.close s) file-seq *in* *out* *err* (clojure.java.io/) file copy delete-file resource as-file as-url as-relative-path GitHub: fs
Data readers	*data-readers* default-data-readers *default-data-reader-fn*
tap	(1.10) tap> add-tap remove-tap

Sequences

- Clojure defines a lot of algorithms in terms of sequences (seqs)
    - In order for collections to allow access as seq, they must define
        - first
        - rest
        - cons
- Most library functions return lazy sequences

Collections

- Lists, vectors, maps, sets
  - Immutable
    - thus trivially thread-safe
  - Persistent
    - (def a ‘(1 2 3))
    - (def b (conj a 0))
- All collections support: count, conj, seq



Collections

Lists

- (+ 1 2 3) — evaluated as invocation
- ‘(+ 1 2 3) — list containing one symbol and 3 primitive types
- optimised for manipulations at the head of the list
- conj inserts at the beginning

Collections

Vectors

- [1 2 3 4]
- contiguous memory layout
- constant complexity random access
- conj inserts at the end

Collections

Maps

- Collection of key-value pairs
- {:a 1 :b 2}
- Commas are considered whitespace and can be used to organise the pairs
    - {:a 1, :b 2}

Collections

Sets

- Sets are zero or more forms enclosed in braces and prefixed by #
    - #{:a :b :c}

Lexical Scoping

- let — isolate vars to a specific scope
    - much like scoping variables in other programming languages

Flow Control

- if, if-not
- when, when-not
- do — evaluate more than one expression and return the last
- iteration — loop/recur

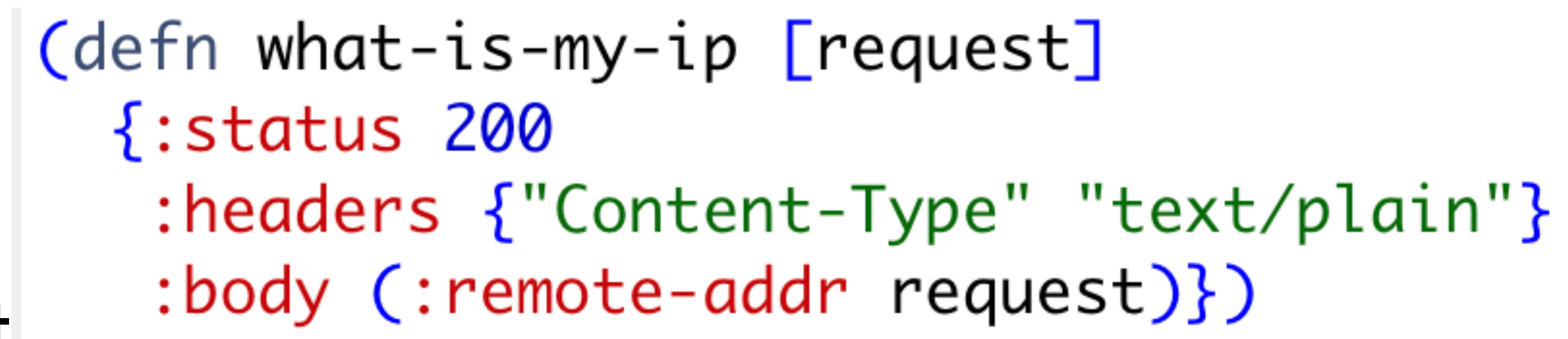
Threading macros

- ->
  - Thread as first argument
- ->>
  - Thread as second argument

Ring

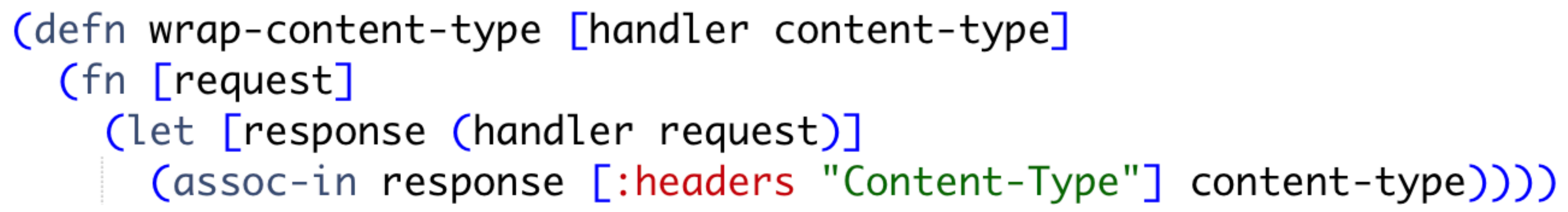
The foundation for web apps in Clojure

- Luminus and other frameworks are based on Ring
- Ring is based on simple abstractions
  - Handler



```clojure
(defn what-is-my-ip [request]
  {:status 200
   :headers {"Content-Type" "text/plain"}
   :body (:remote-addr request)})
```

  - Request
  - Response
  - Middleware

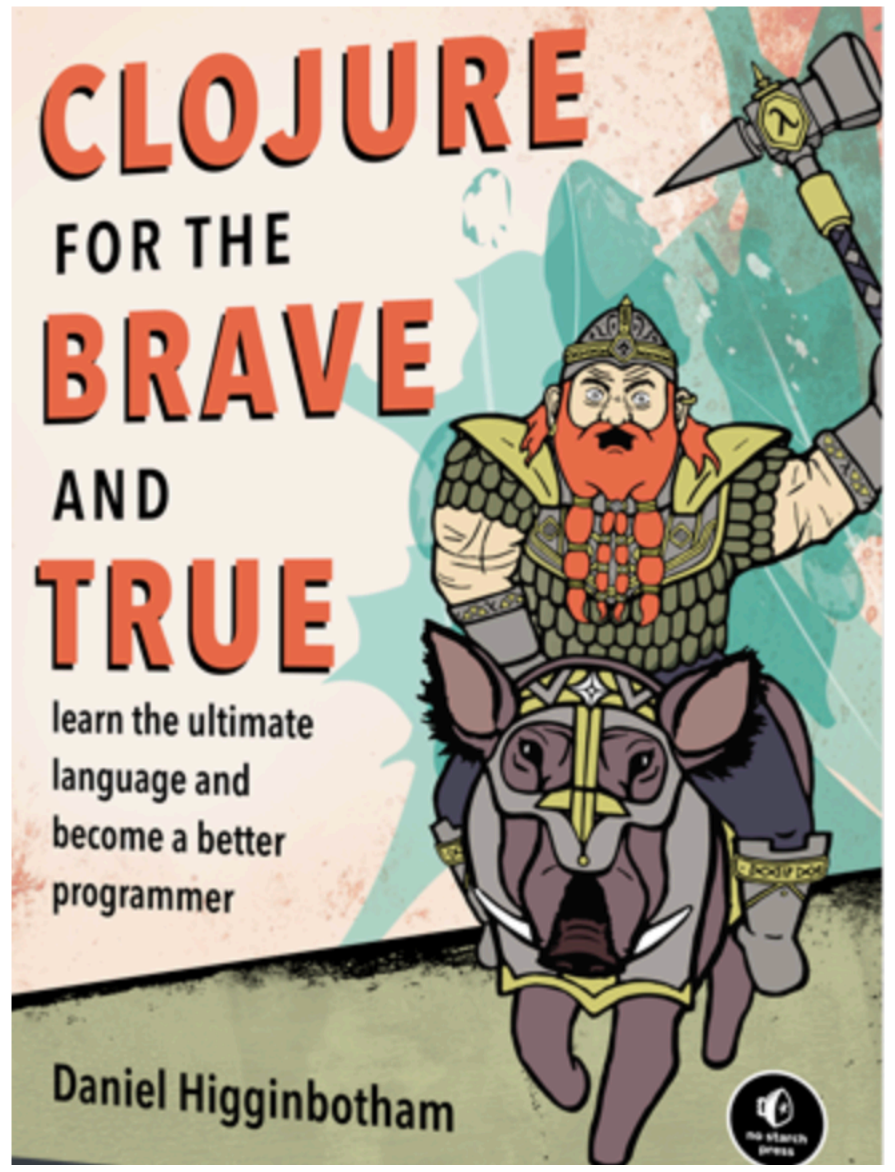


```clojure
(defn wrap-content-type [handler content-type]
  (fn [request]
    (let [response (handler request)]
      (assoc-in response [:headers "Content-Type"] content-type))))
```

Luminus demo

References

- https://clojure.org/index
- https://clojure.org/api/cheatsheet
- https://www.braveclojure.com
- https://leiningen.org
- https://luminusweb.com
- REPL samples can be found here:
  - https://github.com/ccosmin/clojure-brujug-samples



Shameless plug

- www.snowlinesoftware.com
  - Buy some Mac apps ;)

Watermark My PDF
Utilities

Image Shrink - Email Ready
Photo & Video

Ma Musique - the music player
Music

  - www.okclipboard.com
  - www.podcastregister.com