# Clojure Functional Lisp on the JVM

## CONCEPTS

Code is data: everything is (function arg ...). Collections are immutable, so assoc and conj return new values.

**Source form** (+ 1 2)
↓ read
**Data structure** list of symbols
↓ compile
**JVM bytecode** on the fly
↓ run
**Value** 3

## COMMAND LINE

**clj**
Start a REPL with line editing (rlwrap)

**clojure**
Run without rlwrap, for scripts and pipes

**clojure -M -m** *<ns>*
Run the -main function of a namespace

**clojure -M** *<file.clj>*
Run a script file

**clojure -M:***<alias>*
Run with a deps.edn alias

**clojure -X** *<ns>*/*<fn>* **:k v**
Call a function with a map of args

## REPL

**(doc** *<name>***)**
Show docs for a name

**(source** *<name>***)**
Print the source of a function

***1 / *2 / *e**
Last results / last exception

**(require '[***<ns>* **:as** *<a>***] :reload)**
Load a namespace, re-reading its file

## BASICS

**(def x 1)**
Define a var in the namespace

**(defn f "doc" [a & more] ...)**
Function with docstring and rest args

## BASICS (cont.)

**(fn [x] ...) / #(* % 2)**
Anonymous function, short form

**(let [a 1 b (+ a 1)] ...)**
Local bindings, sequential

**(if** *<t>* *<a>* *<b>***) / (when** *<t>* **...)**
Conditional / one-armed with body

**(cond** *<t1>* *<a>* **:else** *<b>***)**
Multi-way conditional

**(loop [i 0] (recur (inc i)))**
Loop with explicit recur

**(doseq [x xs] ...)**
Loop for side effects

**(for [x xs :when (odd? x)] ...)**
Lazy list comprehension

**(str "a" 1 :b)**
Concatenate as strings

**(println ...) / (prn ...)**
Print for humans / as readable data

**(try ... (catch Exception e ...) (finally ...))**
Handle errors

**(nil? x) / (some? x)**
Nil tests

**(= a b) / (not= a b)**
Value equality

## COLLECTIONS

**[1 2 3] / '(1 2 3)**
Vector / list literal

**{:a 1 :b 2}**
Map literal

**#{1 2 3}**
Set literal

**(conj coll x)**
Add: end of vector, front of list

**(assoc m :k v)**
Add or replace a key (or vector index)

**(dissoc m :k)**
Remove a key

**(get m :k** *<default>***) / (:k m)**
Look up a key

## COLLECTIONS (cont.)

**(get-in m [:a :b])**
Look up a nested path

**(assoc-in m [:a :b] v)**
Set a nested path

**(update m :k inc)**
Apply a function to a value

**(update-in m [:a :b] f)**
Apply a function at a nested path

**(merge m1 m2)**
Combine maps, later wins

**(keys m) / (vals m)**
Keys / values of a map

## SEQUENCES

**(map f xs)**
Apply f to each item, lazily

**(filter pred xs) / (remove pred xs)**
Keep / drop matching items

**(reduce f init xs)**
Fold into one value

## FUNCTIONS & MACROS

**(→ x (f a) (g b))**
Thread x as first argument

**(→» xs (map f) (filter p))**
Thread as last argument

**(fn [{:keys [a b]}] ...)**
Destructure a map argument

**(let [[a b & r] xs] ...)**
Destructure a sequence

## STATE & CONCURRENCY

**(swap! a f args)**
Atomically update with f

## NAMESPACES & JAVA

**(ns** *<name>* **(:require [***<lib>* **:as** *<a>***]))**
Declare a namespace and its deps

**(require '[clojure.string :as str])**
Load a library with an alias