Clojure

Setup

- https://goo.gl/5A3lIL
- Install Java
- lein http://leiningen.org
- IntelliJ IDEA https://www.jetbrains.com/idea/download/
- Cursive https://cursiveclojure.com/userguide/

REPL

- `$ lein repl`
- `(+ 1 2)`
- `(doc +)`
- `*1, *2`

LISP

```
(operator arg1 arg2 …)
```

```
(+ 1 2 3) ⇔ plus(1, 2, 3)
```

```
(println "Hello, World")
```

Start Cursive REPL

- Run > Edit Configurations > + > Clojure REPL > Local > set Name > OK

- Run > Run… > REPL

Cursive REPL

- Tools > REPL
  - Send form before caret to REPL (⇧⌘B)
  - Send top form to REPL (⇧⌘P)
- Preferences… > Keymap > …

Paredit (Structural Editing)

- Edit > Structural Editing
    - Slurp Forwards (⇧⌘K)
    - Barf Forwards (⇧⌘J)
    - Wrap with () (⇧⌘9)
    - Splice sexp (⌥S)
- Preferences… > Keymap > …

Function

```
(defn name [argument] content)
```

```
(defn hello [name]
  (str "Hello, " name))
```

Multi-arity

```
(defn hello
  ([] "Hello")
  ([name] (str "Hello, " name))
  ([name1 name2]
   (str "Hello, " name1
        " and " name2)))
```

Variable arguments

```
(defn print-numbers [num1 & rest]
  (println num1)
  (println rest)
  (println (first rest)))
```

Anonymous function

```
(fn [n] (* n 2))
```

```
(def x "X-men")
```

```
(def mul3
  (fn [n] (* n 3)))
```

Short hand

```
#(println % %1 %2 %&)
```

```
(fn [a b & c]
  (println a a b c))
```

_ for don’t care

```
(defn print-second [_ a]
  (println a))
```

let

```
(let [a "Apple"
      b "Bird"
      c (str b " eats " a)]
  (println a)
  (println b)
  (println c))
```

- local scope
- =~ private variable

apply

- Turn list of arguments to single value arguments

```
(defn plus [& args]
  (apply + args))
```

if

```
(if check
    true
    false)
```

Truthiness

- false is false
- nil is false
- others, true
  - “” is true

do

- Wrap expression
- E.g. use in if

```
(do (+ 1 2)
    (* 2 3))
```

```
(defn ifs [x]
  (if x
    (do (println "True")
        100)
    (do (println "False")
        (println ":("))))
```

Collections

- Vector
- List
- Set
- Map

Vector

```
[\a \b \c]
```

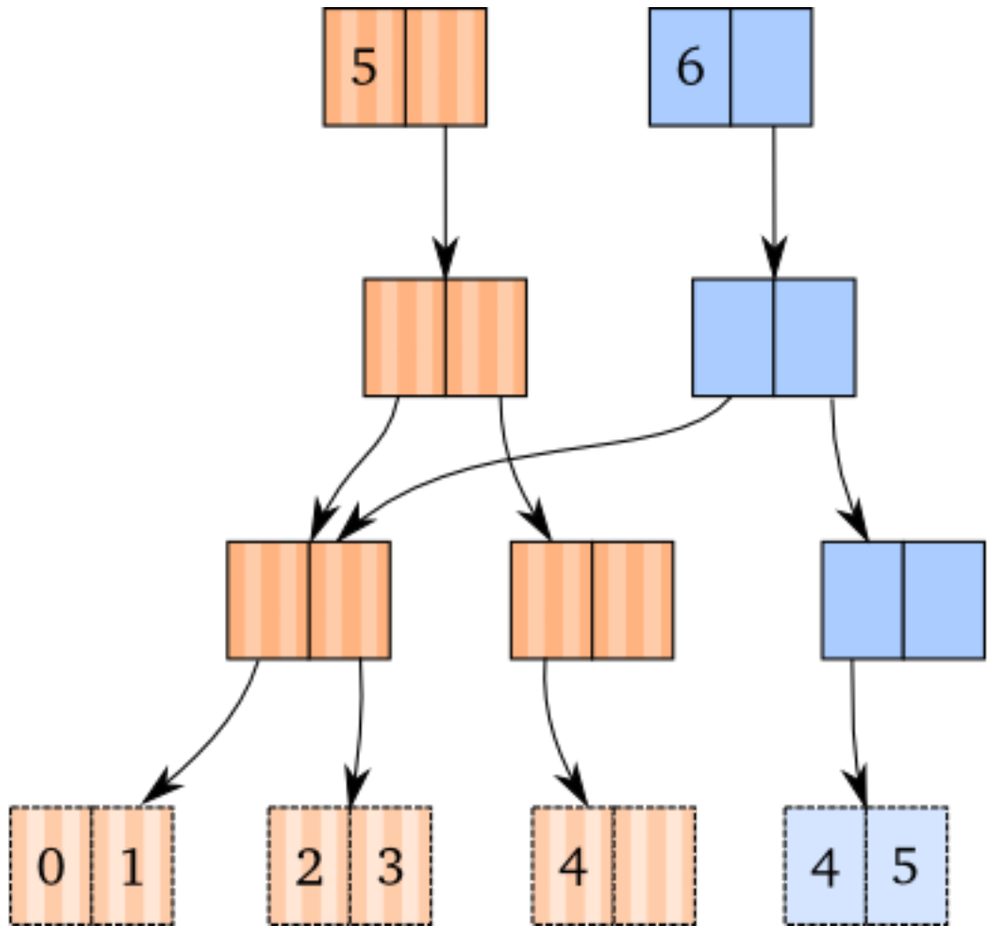
```
(get [\a \b \c] 1)  ([\a \b \c] 1)
```

```
(conj [1 2 3] 4 5 6)
```

```
(assoc [1 2 3] 0 4)
```

Persistent data structure

```
(conj [0 1 2 3 4] 5)
```



http://hypirion.com/musings/understanding-persistent-vector-pt-1

Persistent data structure

- Easy for concurrency
- Fast (enough) access
- Faster / use less memory than fully clone
- No more “who modify it!”
- No more defensive copy
- Easy undo

List

- traditional Lisp data structure
- Clojure code itself

```
'(1 2 3)
```

- Operation on Head
  - stack-like (peek/pop)
- I don’t use it often

conj

- Add in the most natural way
  - at the end for vector
  - at the beginning for list
  - whatever for set

Map

```
{:a 1 :b 2}
```

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

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

Mix

```
(get-in {:a {:b {:c 1}}} [:a :b :c])

(update-in {:a {:b {:c {:d 5}}}} [:a :b :c :d] inc)
```

```
(update-in {:a {:b {:c {:d 5}}}} [:a :b :c :d] * 3)

(update-in {:a {:b [{:z 1} {:y {:x 10}}]}}
           [:a :b 1 :y :x]
           #(if (odd? %) (inc %) (dec %)))

(assoc-in [1 {:a [2 {:b 3}]}] [1 :a] 4)
```

Set

```
#{1 2}
```

```
(conj #{1 2} 2)
```

```
(#{1 2} 2)
```

```
(#{1 2} 3)
```

Some collection functions

```
(count #{1 2 3})
(some {:a 1 :b 2} [:a :b])
(every? pos? [1 2 3])
```

Sequence

- logical list wrapping each collection type for sharing functions

```
(seq [1 2 3])


(map inc [0 1 2])
(map inc '(0 1 2))
(map inc #{1 2 3})
```

Lazy Seq

```
(range 5)
(take 3 (range 5))
(take 5 (iterate inc 10))
```

Some useful functions

- http://clojure.org/cheatsheet

Vector destructuring

```
(defn v-dest [[a [b _ c] :as z]]
    [a b c z])
```

```
(let [[a [b _ c] :as z] nv]
  [z c b a])
```

Map destructuring

```
(defn m-dest
  [{a :x b :y :keys [c d] :as z}]
  [a b c d z])
```

```
(let [{a :x b :y
       :keys [c d] :as z} nm]
  [z d c b a])
```

Thread first (->)

```
(subvec (assoc (conj (conj [1 2 3] 4) 5) 1 1.5) 1 3)
```

```
(-> [1 2 3]
    (conj 4)
    (conj 5)
    (assoc 1 1.5)
    (subvec 1 3))
```

Thread last (->>)

```
(frequencies (filter odd? (map
inc (take 10 (repeatedly #(rand-
int 11))))))
```

```
(->> (repeatedly #(rand-int 11))
     (take 10)
     (map inc)
     (filter odd?)
     (frequencies))
```

Threading Macro

```
(macroexpand-1
  '(->> [1 2 3]
        (map #(* 3))
        (remove odd?)))

(remove odd? (map (fn* [] (* 3))
[1 2 3]))
```

Java Interop

```
(Math/sqrt 9)
```

```
(def ja (ArrayList.))
(.add ja 1)
```

future

```
(future (Thread/sleep 2000)
        (println "Done!"))
```

delay

```
(delay (println "Expensive")
       "Return value”)
```

deref

```
(deref de)
```

```
@de
```

promise

```
(def pm (promise))
```

```
(deliver pm "value")
```

```
@pm
```

atom

```
(def at (atom 10))

(reset! at 11)

(swap! at * 5)

@at
```

agent

```
(def ag (agent [5 9]))
```

```
(send ag slow-conj 15)
```

```
@ag
```

pmap

Recursion

```
(defn repeat-inc [c n]
  (if (= c n)
    "Success"
    (repeat-inc (inc c) n)))
```

recur

```
(defn repeat-inc-recur [c n]
  (if (= c n)
    "Success"
    (recur (inc c) n)))
```

loop/recur

```
(defn repeat-inc-loop [n]
  (loop [c 0]
    (if (= c n)
      "Success"
      (recur (inc c)))))
```

require

```
(require '[clojure.string :as string])
```

```
(require '[clojure.string :refer [upper-case]])
```

```
(ns sprint3r.core
  (:require [clojure.string :as cs]))
```

What else?

Compile to JAR

```
(ns sprint3r.main
  (:gen-class))

(defn -main [args]
  (println "Hello," args))
```

```
; project.clj
:aot [sprint3r.main]
:main sprint3r.main
```

```
$ lein uberjar
$ java -jar target/…
```

Web

```
; Modify project.clj
:dependencies [[org.clojure/clojure "1.6.0"]
               [ring/ring-core "1.4.0-RC2"]
               [ring/ring-jetty-adapter "1.4.0-RC2"]]
:plugins [[lein-ring "0.8.11"]]
:ring {:handler sprint3r.web/handler}
```

```
(ns sprint3r.web)

(defn handler [request]
  {:status 200
   :headers {"Content-Type" "text/html"}
   :body "Hello World"})
```

```
$ lein ring server
$ lein ring uberjar
```

Desktop App

- seesaw
- src/sprint3r/ui.clj

Adding language feature with macro

```
(macroexpand-1
  '(when (> 2 1) 5 6))
```

```
(if (> 2 1) (do 5 6))
```

core.test

- test/sprint3r/core_test.clj
- Runnable in REPL
  - Run tests in current NS in REPL (⇧⌘T)
  - Run test under caret in REPL (^⇧T)
  - Re-run last test action in REPL (⌘T)

ClojureScript

- http://himera.herokuapp.com/index.html
- Om influent React change

```
(js/alert "Hi")
(.log js/console [1 2 3])
(.log js/console (pr-str [1 2 3]))
```

Polymorphic & Record

- http://www.braveclojure.com/multimethods-records-protocols/

core.async

- Borrow from Go-style CSP
- http://www.braveclojure.com/core-async/
- Make it more functional with transducer http://elbenshira.com/blog/understanding-transducers/#transducers-in-core.async

Credits

- Clojure for the brave and true http://www.braveclojure.com
- Clojure By Example http://kimh.github.io/clojure-by-example/
- Living Clojure http://shop.oreilly.com/product/0636920034292.do
- Cognitect Clojure Lab https://github.com/cognitect/clojure-lab
- 4clojure https://www.4clojure.com