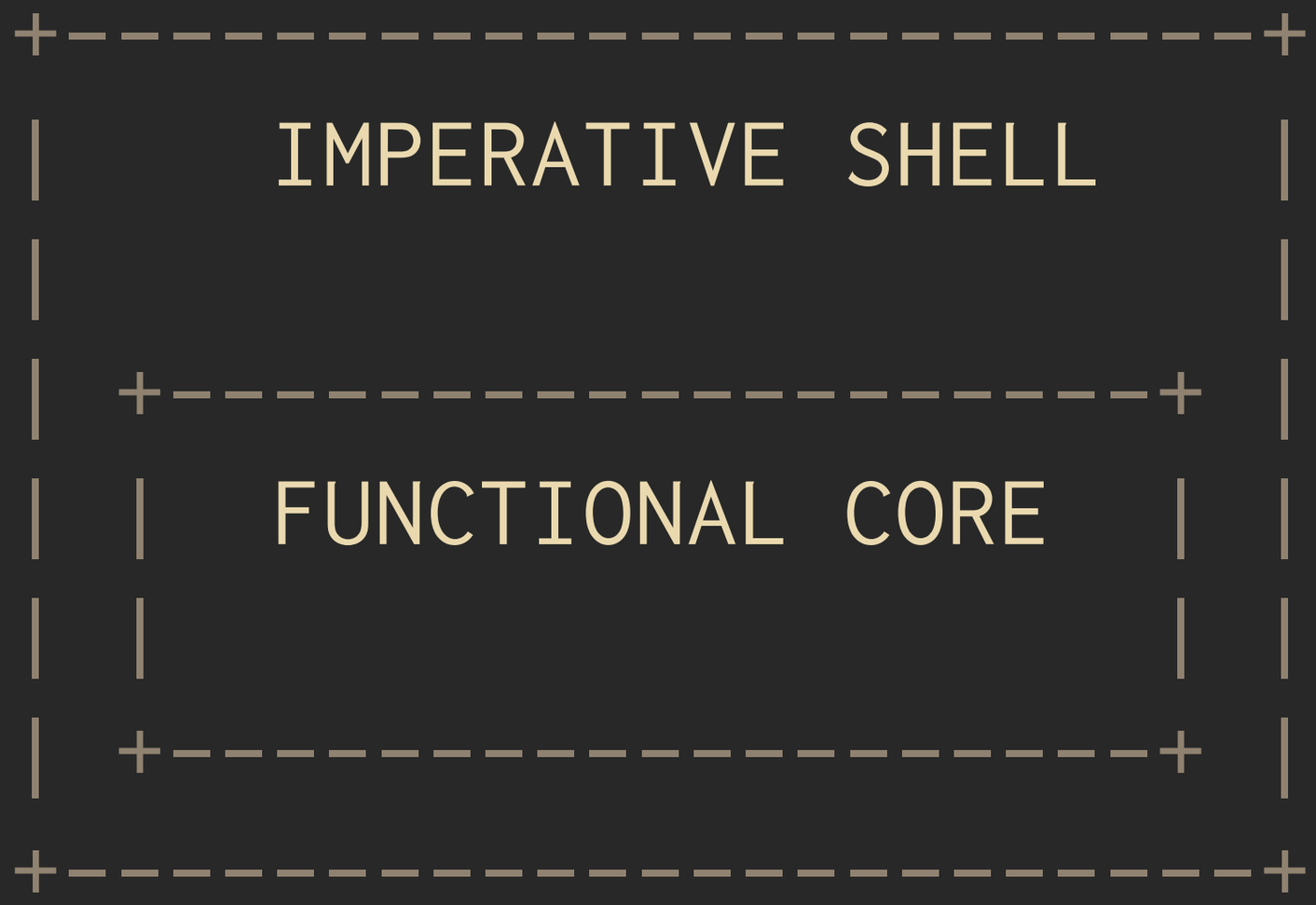
FUNCTIONAL PROGRAMMING WITH CLOJURE

FUNCTIONAL PROGRAMMING

1. SIDE EFFECT FREE
2. IMMUTABILITY
3. FIRST CLASS FUNCTIONS
4. FUNCTION BASED CONTROL FLOW

[1]


IMPERATIVE SHELL
FUNCTIONAL CORE

CLOJURE

1. LISP DIALECT
2. JVM & JS TARGETS
3. DYNAMIC
4. HOMOICONICITY
5. IMMUTABILITY
6. PLATFORM INTEROP
7. SMALL POWERFUL CORE
8. CONCURRENCY

SYNTAXING?

==========

$$1 + 2$$

==========

==========

```
(+ 1 2)
```

==========

==========

```
add(1, 2)
```

==========

==========

```
(add 1, 2)
```

==========

==========

```
(add 1 2)
```

==========

==========

```
1 / 2 * 3
```

==========

==========

```
(/ 1 (* 2 3))
```

==========

==========

CLOJURE BY EXAMPLE

==========

==========

INSTALLING LEININGEN

==========

```
➜ brew install leiningen
```

```
➜ curl -Lo lein http://bit.ly/1m8fHx2
➜ chmod a+x lein
➜ ./lein
```

==========

THE REPL

```
➜ lein repl
nREPL server started on port ...
Clojure 1.6.0

user=>
```

==========

THE REPL

==========

```
user=> (doc map)
;  ------------------
;
user=> (find-doc "fold")
;  ------------------
```

==========

DATA TYPES

```
==========================================
                DATA TYPES
==========================================

(class 1)                   java.lang.Long
(class "Hello")             java.lang.String
(class 1.0)                 java.lang.Double
(class \H)                  java.lang.Character
(class true)                java.lang.Boolean

==========================================
```

```
==========================================
                DATA TYPES
==========================================

(class nil)                 nil
(class (fn [] 1))           clojure.lang.IFn
(class 5/3)                 clojure.lang.Ratio
(class :test)               clojure.lang.Keyword
(class 'a)                  clojure.lang.Symbol

==========================================
```

DATA TYPES

```
(class '(1 2))          clojure.lang.List*
(class [1 2])           clojure.lang.Vector*
(class #{1 2})          clojure.lang.Set*
(class {:a 1 :b 2})     clojure.lang.Map*
```

```
==========================================
              POP QUIZ HOT SHOT
==========================================

(= __ true)                  (= __ (= "a" :a 'a))
(= __ (= 2 2/1))             (= __ (= 2 2/1))
(= :a (keyword __))          (not= __ false)
(= __ (== 2.0 2))            (= false (not __))

==========================================
                                      [3]
```

LISTS, VECTORS & SETS

LISTS, VECTORS & SETS

```
(list 1 2 3 2 1)       '(1 2 3 2 1)
(vector 1 2 3 2 1)      [1 2 3 2 1]
(hash-set 1 2 3 2 1)   #{1 2 3}
```

```
=======================================
        LISTS, VECTORS & SETS
=======================================

(= __ (count '(42)))

(= __ (conj [1 2] 3))

(= __ (cons 1 [2 3]))

(= __ (first [1 2 3]))

(= __ (last [1 2 3]))

=======================================
```

```
==========================================
          LISTS, VECTORS & SETS
==========================================

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

(= __ (nth [1 2 3] 2))

(= __ (peek [1 2 3]))

(= __ (pop [1 2 3]))

(= __ (rest []))

==========================================
```

MAPS

MAPS

```
(hash-map {:a 1})      { :a 1 }
(hash-map {:a 1 :b})   ERROR!
```

```
==========================================
                  MAPS
==========================================

(= __ (get {:b 2} :b))

(= __ ({:a 1} :a))

(= __ (:a {:a 1}))

(= __ (:b {:a 1}))

(= __ (:b {:a 1} 2))

==========================================
```

```
==========================================
                  MAPS
==========================================

(= __ (count {:a 1}))

(= __ (:b {:a 1} 2))

(= __ (assoc {:a 1} :b 2))

(= __ (dissoc {:a 1 :b 2} :a))

(= __ (dissoc {:a 1 :b 2} :a :b))

==========================================
```

==========

MAPS

==========

```
(= __ (contains? {:a nil :b nil} :b))
(= __ (keys {:a 1 :b 2}))
(= __ (vals {:a 1 :b 2}))
```

==========

FUNCTIONS

==========

FUNCTIONS

==========

```clojure
(def sq (fn [a] (* a a)))
(defn sq [a] (* a a))
(def sq #(* % %))
```

==========

```
==========================================
                FUNCTIONS
==========================================

(= __ ((fn [n] (* 5 n)) 2))

(= __ (#(* 15 %) 4))

(= __ (#(+ %1 %2 %3) 4 5 6))

(= __ (#(* 15 %2) 1 2))

(= 9 (((fn [] ___)) 4 5))

==========================================
```

CONDITIONALS

```
==========================================
                CONDITIONALS
==========================================


(= __ (if (false? (= 4 5))
          :a :b))
(= __ (if (> 4 3) []))


==========================================
```

==========

CONDITIONALS

==========

```
(let [x 5]
  (= :your-road
     (cond (= x __) :road-not-taken
           (= x __) :another-not-taken
           :else __)))
```

==========

```
==========================================
                CONDITIONALS
==========================================

(let [choice 5]
  (= :your-road (case choice
                  __ :road-not-taken
                  __ :your-road
                  :another-not-taken)))

==========================================
```

LOOPING

LOOPING

```
(= __ (loop [v 1]
        (if-not (> v 5)
          (recur (inc v))
          v))
```

HIGHER ORDER FUNCTIONS

```
==========================================
          HIGHER ORDER FUNCTIONS
==========================================


(= [__ __ __] (map #(* 4 %) [1 2 3]))
(= __ (filter nil? [:a :b nil :c :d]))
(= __ (reduce * [1 2 3 4]))


==========================================
```

LAZY SEQUENCES

```
==========================================
                LAZY SEQUENCES
==========================================

(= __ (range 1 5))

(= __ (range 5))

(= [0 1 2 3 4 5] (take __ (range 100)))

(= __ (take 20 (iterate inc 0)))

(= [:a :a :a :a :a :a] (repeat __ __))

==========================================
```

USEFUL MACROS

```
==========================================
                USEFUL MACROS
==========================================

(= __ (-> "a b c d"
          .toUpperCase
          (.replace "A" "X")
          (.split " ")
          first))

==========================================
```

USEFUL MACROS

```
(= __ (->> (range)
           (filter even?)
           (take 10)
           (reduce +)))
```

```
==========================================
                USEFUL MACROS
==========================================

(= __ (try
        (/ 1 0)
        true
      (catch Exception e
        false)))

==========================================
```

ATOMS

```
==========================================
                  ATOMS
==========================================
(let [my-atom (atom 1)]
  (= __ @my-atom)
  (swap! my-atom inc)
  (= __ @my-atom)
  (reset! my-atom 4)
  (= __ @my-atom))
==========================================
```

(REST CLOJURE)

1. JAVA INTEROP
2. MACROS
3. DESTRUCTURING
4. RECORDS
5. PROTOCOLS
6. COMPREHENSION
7. TRANSDUCERS
8. CLOJURESCRIPT

==========

IT'S DANGEROUS TO GO ALONE…

==========

1. CLOJURE GRIMOIRE
2. WEIRD & WONDERFUL CHARACTERS OF CLOJURE
3. CLOJURE DOCS
4. CLOJURE FOR THE BRAVE AND TRUE

==========

==========

DEAD TREE EDITION

==========

1. JOY OF CLOJURE 2ND EDITION
2. PROGRAMMING CLOJURE
3. FUNCTIONAL THINKING
4. STRUCTURE & INTERPRETATION OF COMPUTER PROGRAMS

==========

==========

ATTRIBUTION

==========

[1]: http://bit.ly/learning-clojure

[2]: http://bit.ly/destroy-all-software

[3]: http://bit.ly/clojure-koans

[4]: http://bit.ly/clojure-gist

==========