

# Running with Scissors: Live Coding with Data

@stuarthalloway

# About Me

30 years programming

Languages: 8086 Assembly, C, C++, Clojure, Java, JavaScript, Ruby, Obj-C, Perl, Python, Smalltalk

Roles: developer, founder, manager, operations, owner, stakeholder, support, tester, trainer

Developer: Clojure, ClojureScript, Datomic

# About Clojure

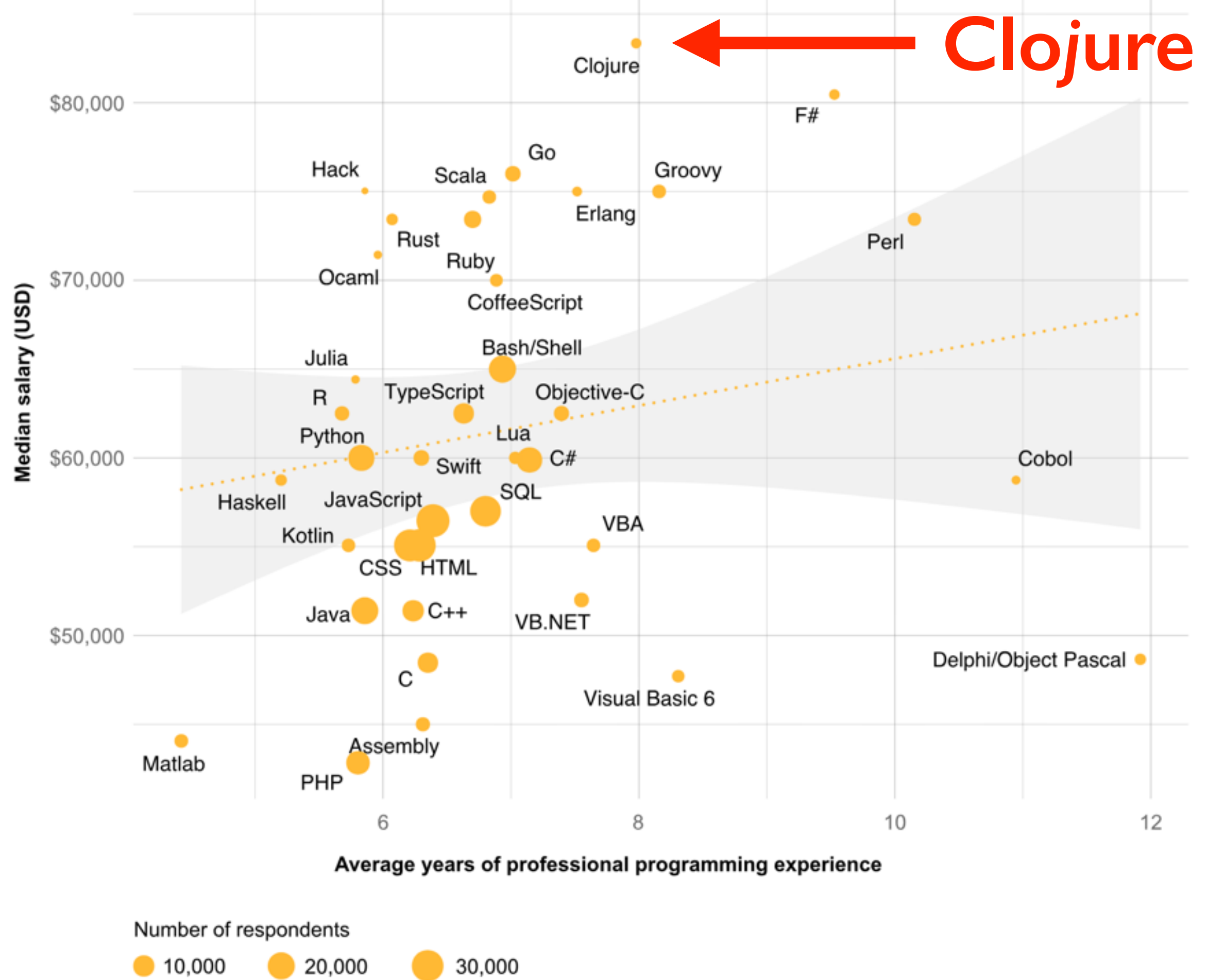
Dynamic

Functional

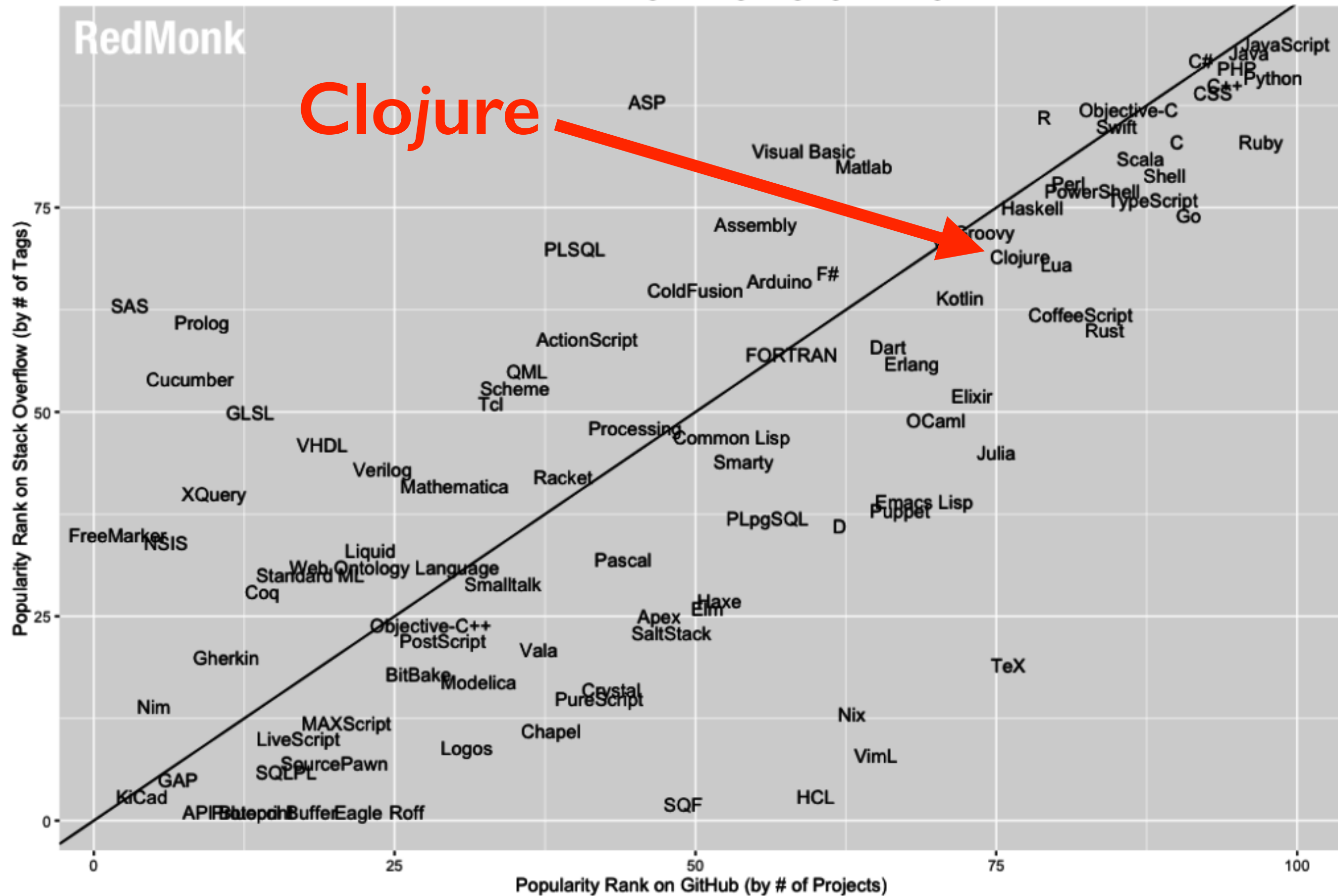
Hosted

Lisp





## RedMonk Q318 Programming Language Rankings



# Rationale

Dynamic

Functional

Hosted

Lisp



# Running with Scissors: Live Coding with Data

(a Clojure dev workflow)

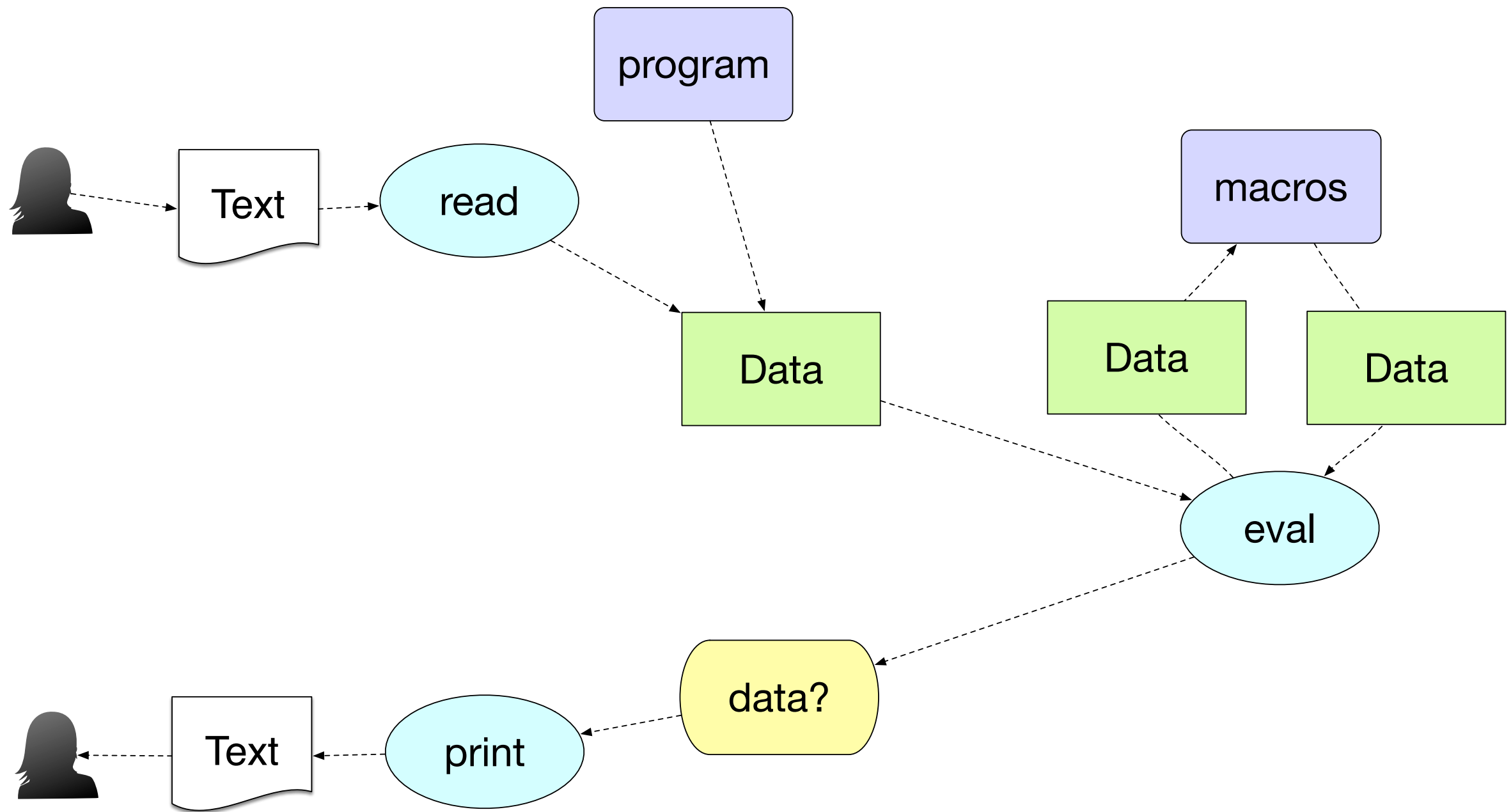
# Running

Work inside your running program

REPL: Read, Eval, Print, Loop

Fast feedback loop





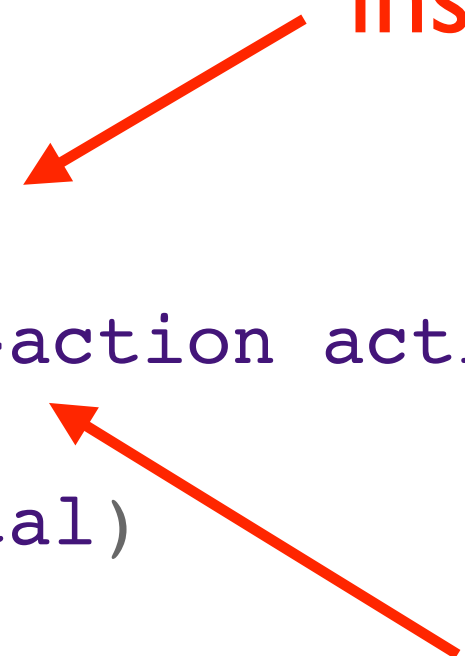
# A La Carte Read

reducing over  
financial transactions

```
1 (let [db (d/db conn)
2       ent (d/entity db ftx)]
3   (println (d/touch ent))
4   (flush)
5   (let [action (edn/read in)
6         result (perform-cat-action action conn ftx)]
7     (case result
8       :quit (reduced total)
9       :next (inc total)
10      :skip total
11      :again (recur))))
```

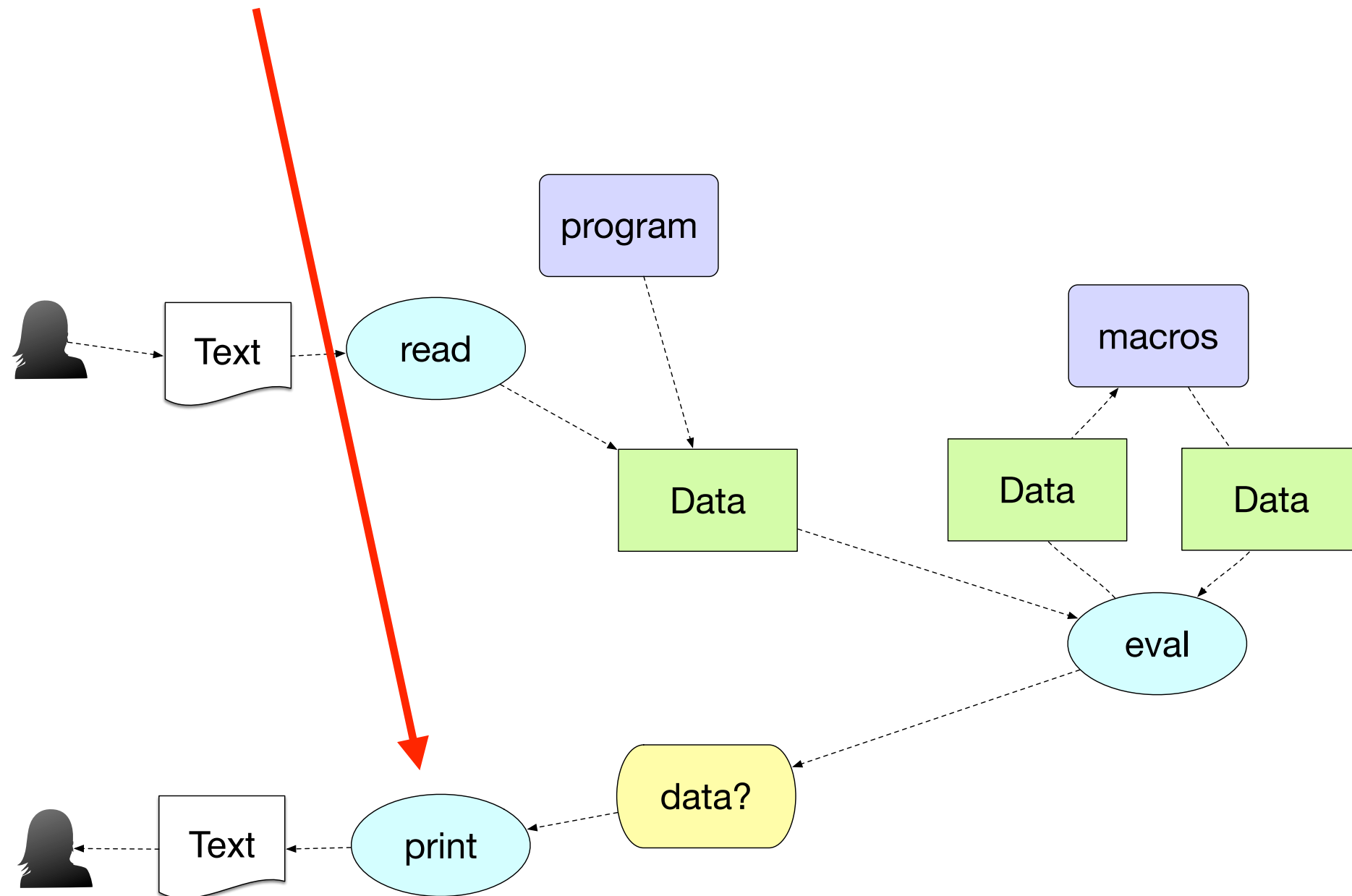
read  
instruction

polymorphic  
processing

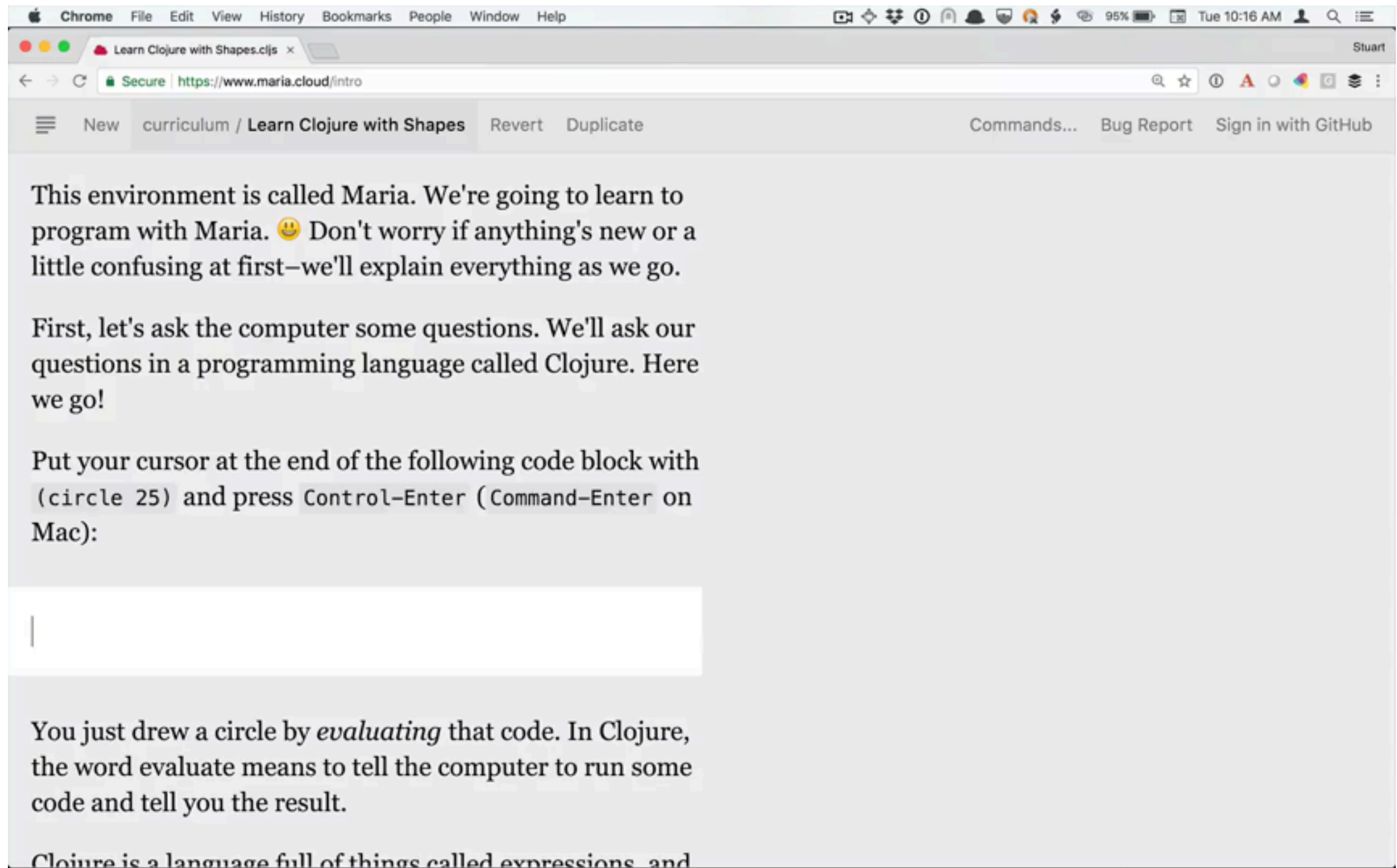


# Custom Print

```
1 (require '[clojure.pprint :as pp]  
2         '[clojure.main :as main])  
3 (clojure.main/repl :print pp/pprint)
```



# Custom Error Printing



The screenshot shows a web browser window with the URL <https://www.maria.cloud/intro>. The browser's address bar and tabs are visible at the top. The main content area of the browser displays a tutorial for learning Clojure in the Maria Cloud environment. The tutorial text is as follows:

This environment is called Maria. We're going to learn to program with Maria. 😊 Don't worry if anything's new or a little confusing at first—we'll explain everything as we go.

First, let's ask the computer some questions. We'll ask our questions in a programming language called Clojure. Here we go!

Put your cursor at the end of the following code block with `(circle 25)` and press `Control-Enter` (`Command-Enter` on Mac):

```
|
```

You just drew a circle by *evaluating* that code. In Clojure, the word evaluate means to tell the computer to run some code and tell you the result.

Clojure is a language full of things called expressions and

# Common REPL Concerns

I already have a shell...

Typing into a REPL sucks...

Somebody made spaghetti code at a REPL once...

# “Just a Shell” is Not Enough

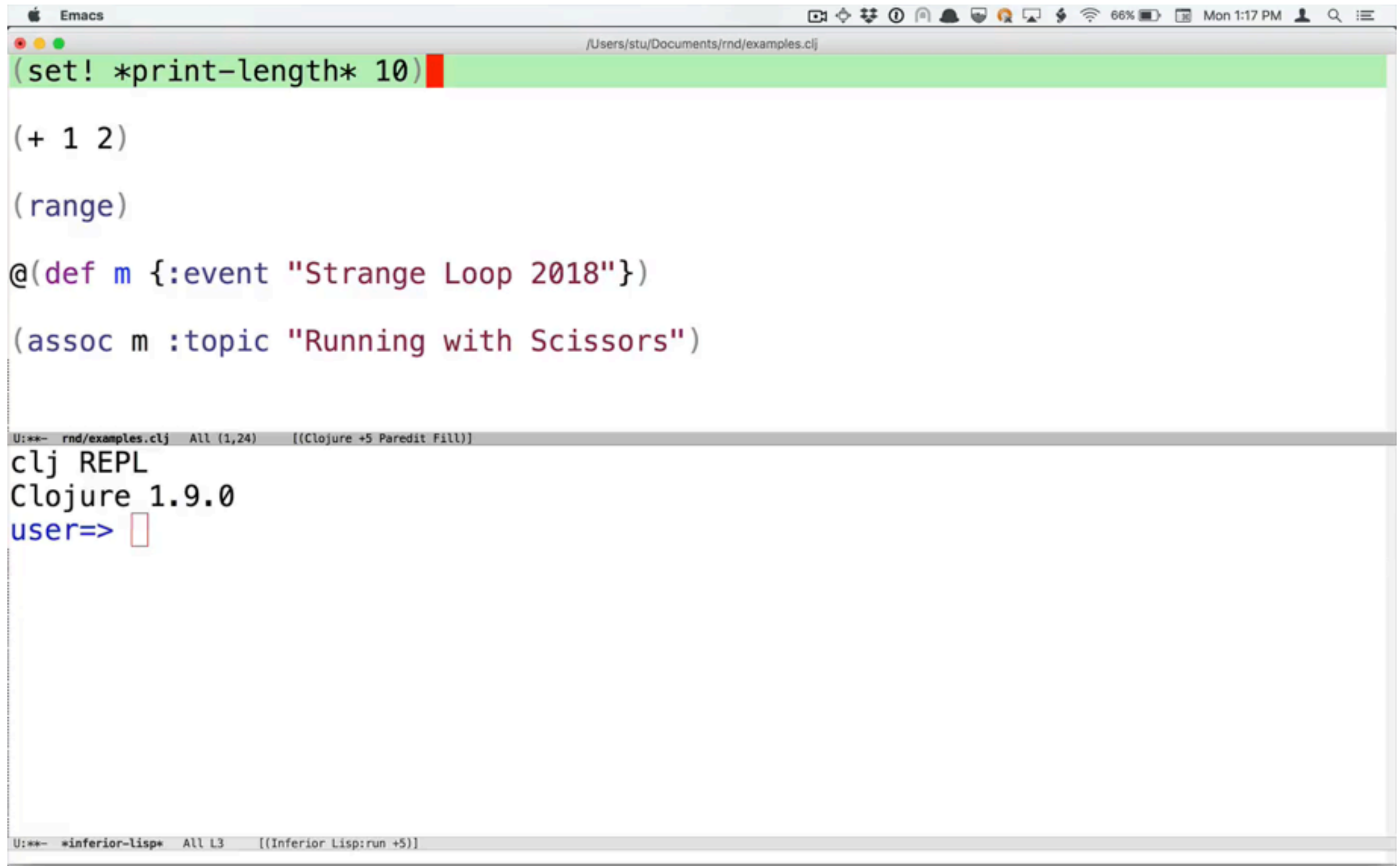
	REPL	“Sidecar” Shell
program semantics	sequential evaluation of forms	files, modules, projects, etc.
interactive semantics	sequential evaluation of forms	special, maybe not like programs
text -> data	read	literals? parser?
text -> code	read	parser to AST? eval text? object to text? toString?
execute	eval data	
data -> text	print	

# Sidecar Shells: JShell

“The JShell state includes an *evolving code and execution state*. To facilitate rapid investigation and coding, statements and expressions *need not occur within a method*, and variables and method *need not occur within a class*.”

— <http://openjdk.java.net/jeps/222>

# REPL is Not About Text Entry



```
Emacs /Users/stu/Documents/rnd/examples.clj
(set! *print-length* 10)

(+ 1 2)

(range)

@(def m {:event "Strange Loop 2018"})

(assoc m :topic "Running with Scissors")

U:*- rnd/examples.clj All (1,24) [(Clojure +5 Paredit Fill)]
clj REPL
Clojure 1.9.0
user=>

U:*- *inferior-lisp* All L3 [(Inferior Lisp:run +5)]
```



# Spaghetti Code?

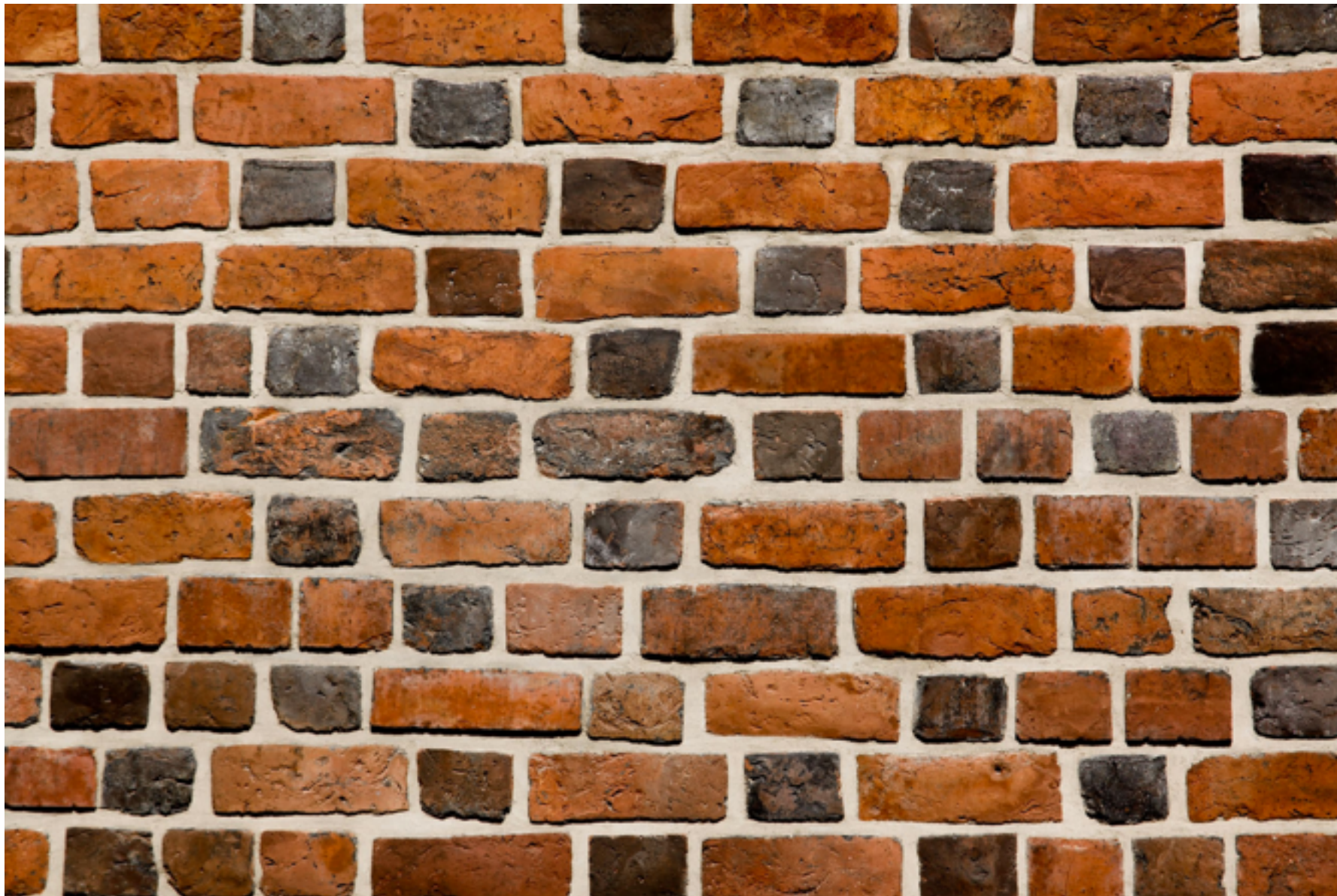
REPL + imperative = faster spaghetti





# Functional Code

REPL + Functional = faster bricks



# ...with Scissors

Don't run your entire program!

Focus: cut your code and data

down to match the job at hand

# Task-Specific Dev

Start with example data

maybe generate data for exploration

Interactively test some fns

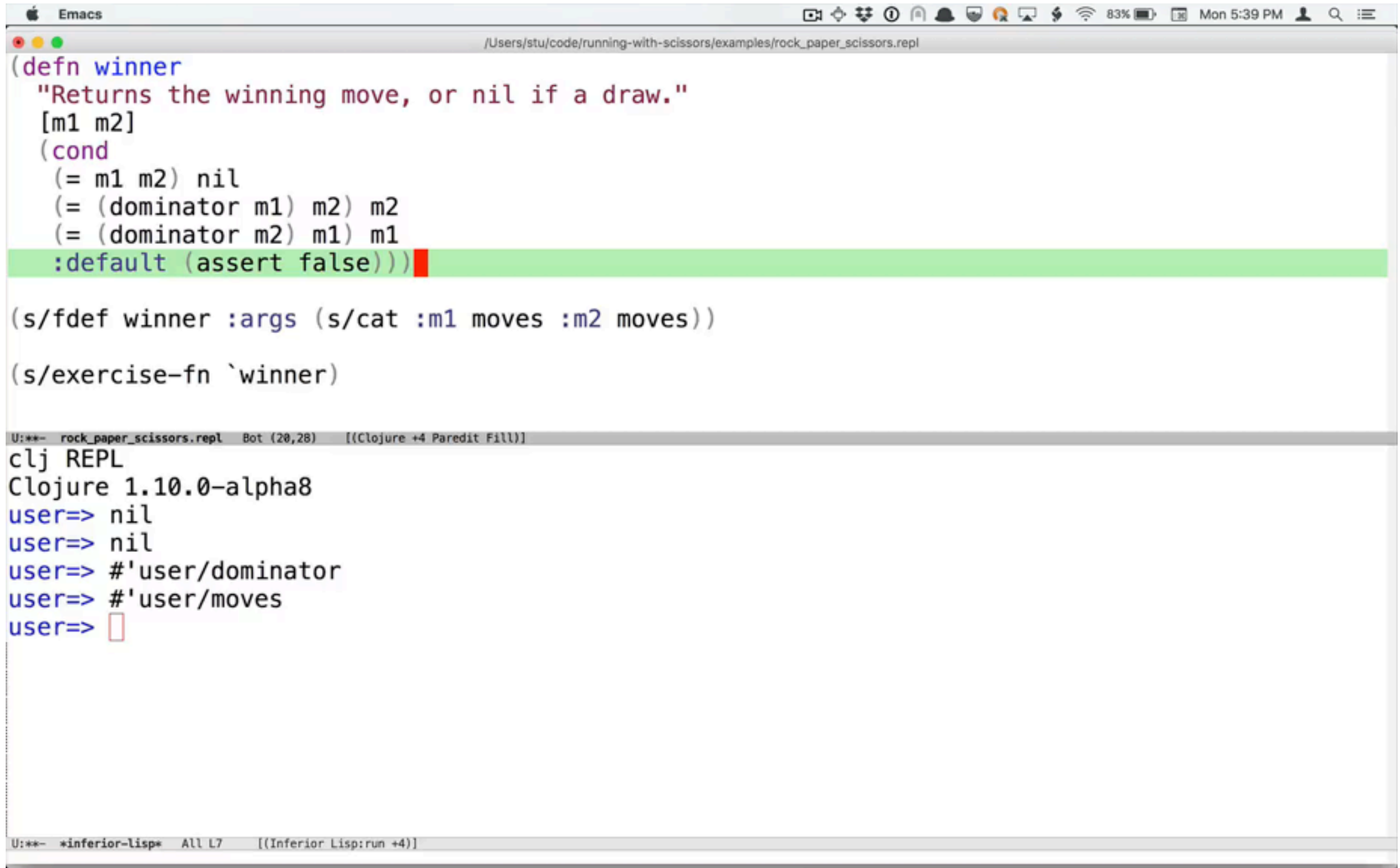
Load what you need (namespaces, vars, etc.)

Custom UI

# Example Data

```
1 (def dominator
2   "Map from a move to its dominator."
3   {:rock :paper
4    :scissors :rock
5    :paper :scissors})
6
7 (def moves
8   "The set of legal moves."
9   (into #{} (keys dominator)))
```

# Generating Data



The image shows a screenshot of an Emacs editor window. The title bar at the top indicates the application is Emacs, and the window title is `/Users/stu/code/running-with-scissors/examples/rock_paper_scissors.repl`. The main editing area contains the following Clojure code:

```
(defn winner
  "Returns the winning move, or nil if a draw."
  [m1 m2]
  (cond
    (= m1 m2) nil
    (= (dominator m1) m2) m2
    (= (dominator m2) m1) m1
    :default (assert false)))

(s/def winner :args (s/cat :m1 moves :m2 moves))

(s/exercise-fn `winner)
```

The line `:default (assert false))` is highlighted in green. Below the code, the REPL session is visible, showing the following interactions:

```
U:*** rock_paper_scissors.repl Bot (20,28) [(Clojure +4 Paredit Fill)]
clj REPL
Clojure 1.10.0-alpha8
user=> nil
user=> nil
user=> #'user/dominator
user=> #'user/moves
user=> []
```

At the bottom of the window, the status bar shows `U:*** *inferior-lisp* All L7 [(Inferior Lisp:run +4)]`.

# Load What You Need

code

```
1 (require '[clojure.data.csv :as csv]
2          '[clojure.java.io :as io]
3          '[clojure.pprint :as pp]
4          '[clojure.spec.alpha :as s]
5          '[datomic.api :as d]
6          '[pfinance.repl :refer :all])
7
8 (set! *print-length* 50)
9 (def uri "datomic:dev://localhost:4334/pfinance")
10 (def conn (d/connect uri))
11 (def db (d/db conn))
```

state

# Custom UI

Don't pound your head against a wall of text

pretty-print (and print-table) it

inspect it

make a spreadsheet

make HTML

make a picture



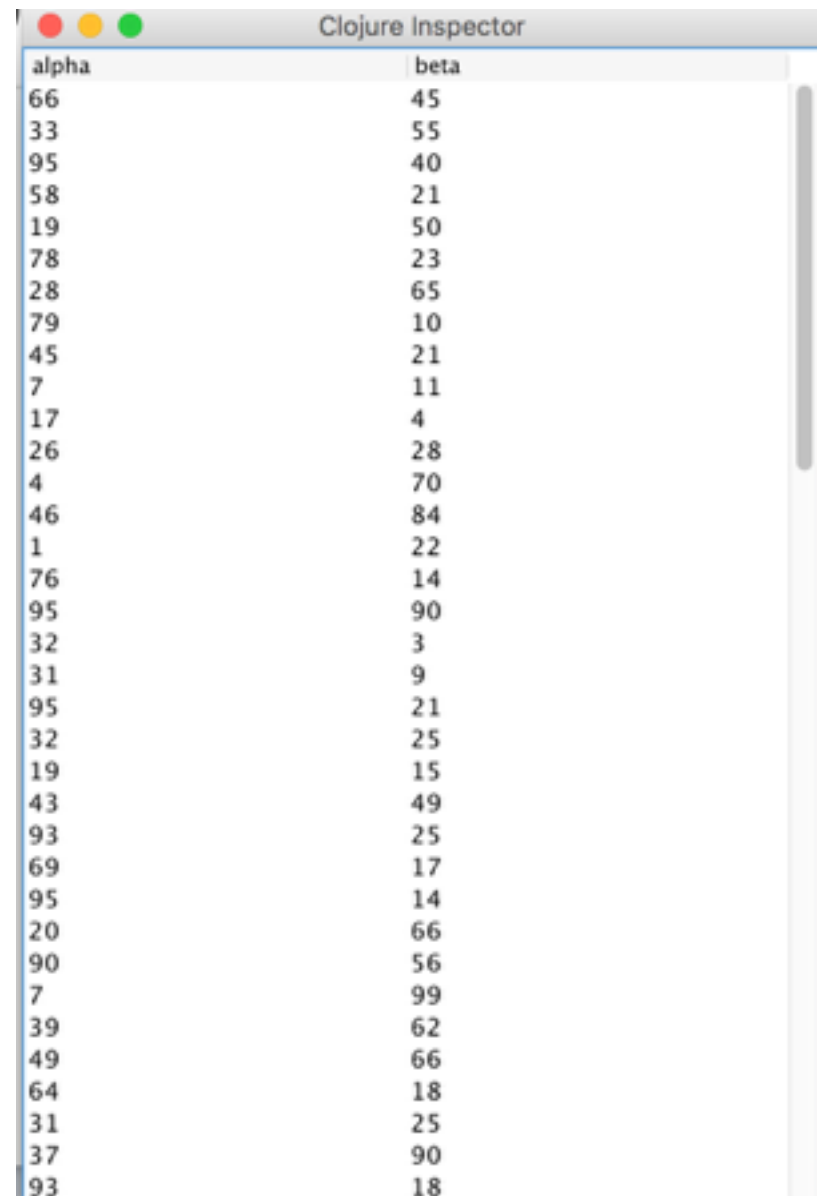
# No

```
@(def data (repeatedly 100 (fn [] {:alpha (rand-int 100)
                                     :beta (rand-int 100)})))
```

```
=> ({:alpha 58, :beta 45} {:alpha 45, :beta 83} {:alpha 64, :beta 71} {:alpha 0, :beta 30}
{:alpha 8, :beta 24} {:alpha 50, :beta 49} {:alpha 73, :beta 63} {:alpha 1, :beta 3}
{:alpha 30, :beta 34} {:alpha 79, :beta 62} {:alpha 64, :beta 29} {:alpha 72, :beta 74}
{:alpha 28, :beta 15} {:alpha 64, :beta 53} {:alpha 74, :beta 38} {:alpha 93, :beta 26}
{:alpha 9, :beta 89} {:alpha 48, :beta 29} {:alpha 64, :beta 51} {:alpha 35, :beta 15}
{:alpha 0, :beta 79} {:alpha 74, :beta 91} {:alpha 17, :beta 99} {:alpha 2, :beta 14}
{:alpha 66, :beta 70} {:alpha 75, :beta 69} {:alpha 40, :beta 70} {:alpha 29, :beta 82}
{:alpha 85, :beta 94} {:alpha 2, :beta 68} {:alpha 2, :beta 28} {:alpha 30, :beta 34}
{:alpha 57, :beta 48} {:alpha 57, :beta 87} {:alpha 44, :beta 38} {:alpha 29, :beta 14}
{:alpha 55, :beta 88} {:alpha 2, :beta 59} {:alpha 28, :beta 5} {:alpha 17, :beta 4}
{:alpha 44, :beta 35} {:alpha 79, :beta 8} {:alpha 18, :beta 36} {:alpha 7, :beta 7}
{:alpha 32, :beta 5} {:alpha 56, :beta 34} {:alpha 12, :beta 73} {:alpha 88, :beta 98}
{:alpha 20, :beta 41} {:alpha 72, :beta 73} {:alpha 72, :beta 75} {:alpha 5, :beta 29}
{:alpha 68, :beta 9} {:alpha 60, :beta 89} {:alpha 4, :beta 27} {:alpha 11, :beta 28}
{:alpha 4, :beta 91} {:alpha 68, :beta 86} {:alpha 2, :beta 23} {:alpha 62, :beta 38}
{:alpha 19, :beta 81} {:alpha 9, :beta 67} {:alpha 56, :beta 43} {:alpha 59, :beta 69}
{:alpha 52, :beta 68} {:alpha 99, :beta 60} {:alpha 76, :beta 11} {:alpha 55, :beta 73}
{:alpha 48, :beta 64} {:alpha 72, :beta 95} {:alpha 95, :beta 23} {:alpha 89, :beta 65}
{:alpha 28, :beta 14} {:alpha 91, :beta 10} {:alpha 49, :beta 68} {:alpha 95, :beta 43}
{:alpha 31, :beta 90} {:alpha 74, :beta 99} {:alpha 44, :beta 86} {:alpha 17, :beta 77}
{:alpha 93, :beta 56} {:alpha 11, :beta 50} {:alpha 58, :beta 38} {:alpha 27, :beta 88}
{:alpha 92, :beta 62} {:alpha 56, :beta 33} {:alpha 48, :beta 73} {:alpha 14, :beta 14}
{:alpha 56, :beta 97} {:alpha 97, :beta 28} {:alpha 41, :beta 30} {:alpha 99, :beta 47}
{:alpha 30, :beta 18} {:alpha 53, :beta 99} {:alpha 51, :beta 1} {:alpha 96, :beta 95}
{:alpha 40, :beta 23} {:alpha 46, :beta 74} {:alpha 33, :beta 43} {:alpha 27, :beta 84})
```

# Inspect It

```
(require '[clojure.inspector :as ins])  
(ins/inspect-table data)
```



The screenshot shows a window titled "Clojure Inspector" with a table of data. The table has two columns, "alpha" and "beta", and 32 rows of data. The data is as follows:

alpha	beta
66	45
33	55
95	40
58	21
19	50
78	23
28	65
79	10
45	21
7	11
17	4
26	28
4	70
46	84
1	22
76	14
95	90
32	3
31	9
95	21
32	25
19	15
43	49
93	25
69	17
95	14
20	66
90	56
7	99
39	62
49	66
64	18
31	25
37	90
93	18

# Spreadsheet It

```
(with-open [w (io/writer "temp.csv")]
  (csv/write-csv
    w
    (map->csv data [:alpha :beta])))
(sh/sh "open" "temp.csv")
```

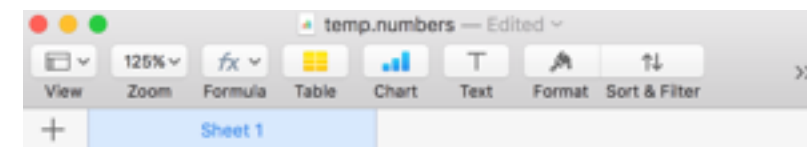
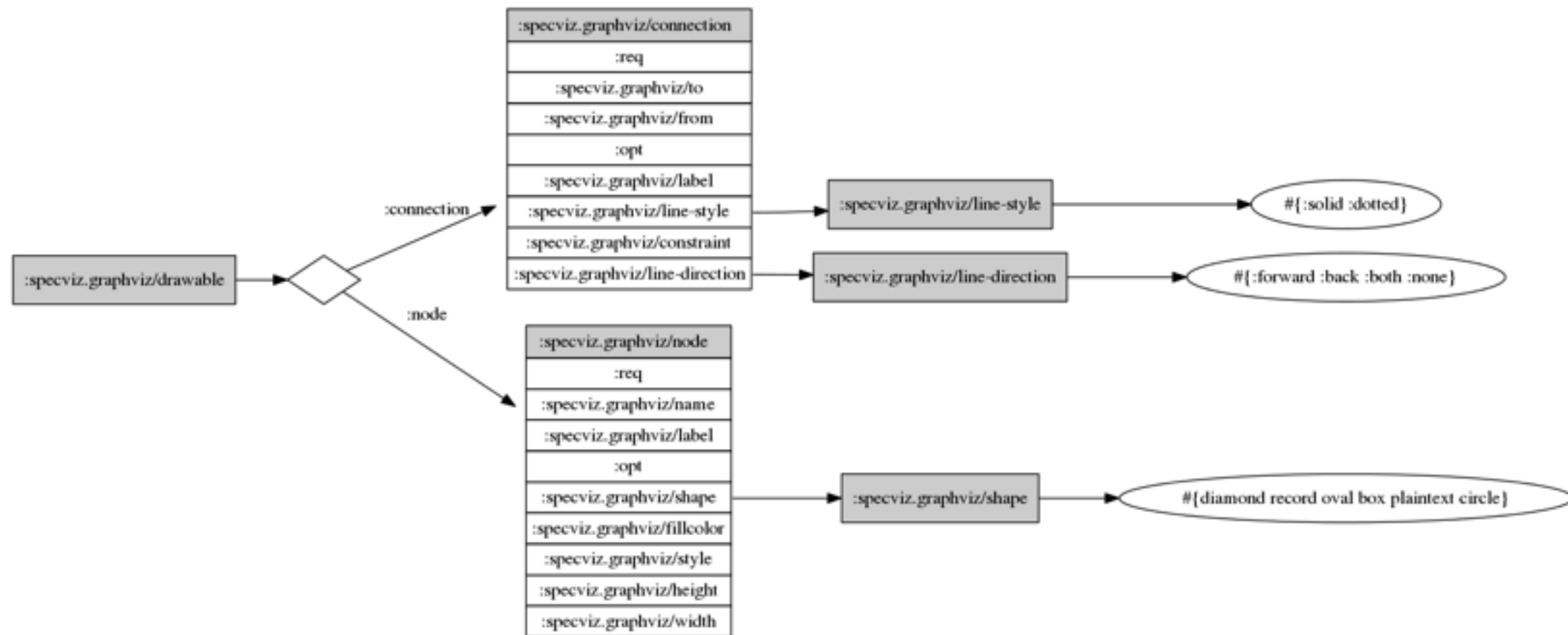


Table 1

:alpha	:beta
58	45
45	83
64	71
0	30
8	24
50	49
73	63
.	.

# Picture It

Specviz: generate Graphviz from Clojure spec



# Too Much Work?

“With a REPL first I need to do a load of import statements. Then populate my objects. Then get started with the actual debugging. I personally don't find it anywhere near as efficient as an IDE that has been setup.”

— <https://news.ycombinator.com/item?id=16315552>

“do” block, anyone?

— me

# Rich Comment Blocks

At end of a .clj file

Sample data

Sample invocations

*Durable history of the dev process*

# Rich Comments

```
(comment
(do
  (refer 'set)
  (def xs #{{:a 11 :b 1 :c 1 :d 4}
             {:a 2 :b 12 :c 2 :d 6}
             {:a 3 :b 3 :c 3 :d 8 :f 42}})

  (def ys #{{:a 11 :b 11 :c 11 :e 5}
             {:a 12 :b 11 :c 12 :e 3}
             {:a 3 :b 3 :c 3 :e 7 }}))

(join xs ys)
(join xs (rename ys {:b :yb :c :yc}) {:a :a}))

(union #{:a :b :c} #{:c :d :e })
(difference #{:a :b :c} #{:c :d :e})
(intersection #{:a :b :c} #{:c :d :e})

(index ys [:b])
)
```

setup



expedition  
log



# What About Tests?

Automate (re)running of REPL interactions

Instead of testing specialness, lean on language for  
lifecycle / reuse / scope / state / validation



# Transcriptor

```
1 ;; my-test.repl
2 (require '[cognitect.transcriptor :refer (check!)] )
3
4 ;; check exact match
5 (+ 1 2)
6 (check! #{3})
7
8 ;; check predicate (or any spec!)
9 (+ 1 1)
10 (check! even?)
```

```
1 ;; my-suite.repl
2 (require '[cognitect.transcriptor :as xr :refer (check!)] )
3 (xr/run "my-test.repl")
```

# Sets: Scissors-Ready Data

Just do it

Strongly-named keys

```
{:github/id "stuarthalloway"  
 :github/location "Chapel Hill, NC"}
```

```
{:name/last "Halloway"  
 :name/first "Stuart"}
```

```
{:twitter/id "stuarthalloway"  
 :twitter/joined #inst "2008-03"}
```

# Slots

Enumerate prior to use

Specify names, or types, or both

**Sorry, No Scissors!** — separate API per struct

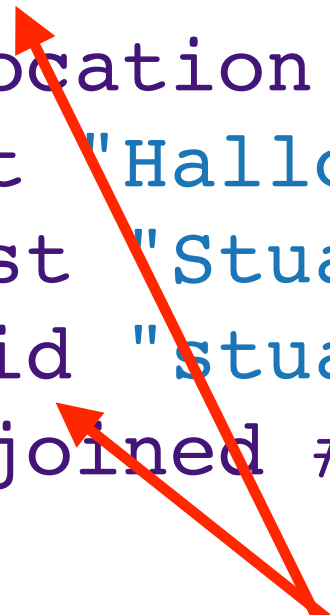
```
struct Person {  
    String last;  
    String first;  
};
```

```
struct GithubAccount {  
    String id;  
    Date joined;  
};
```

```
struct TwitterAccount {  
    String id;  
    Date joined;  
};
```

# Ad hoc Merge

```
(def about-me  
  (merge  
    {:github/id "stuarthalloway"  
     :github/location "Chapel Hill, NC"}  
    {:name/last "Halloway"  
     :name/first "Stuart"}  
    {:twitter/id "stuarthalloway"  
     :twitter/joined #inst "2008-03"}))
```



multiple ids ok, no collision  
thanks to namespace names

# Ad Hoc Enumeration

```
(defn keys-named
  "Given map m, return all the keys whose name
  component is n."
  [m n]
  (filter #(= (name %) "id") (keys m)))

(keys-named about-me "id")
=> (:github/id :twitter/id)
```

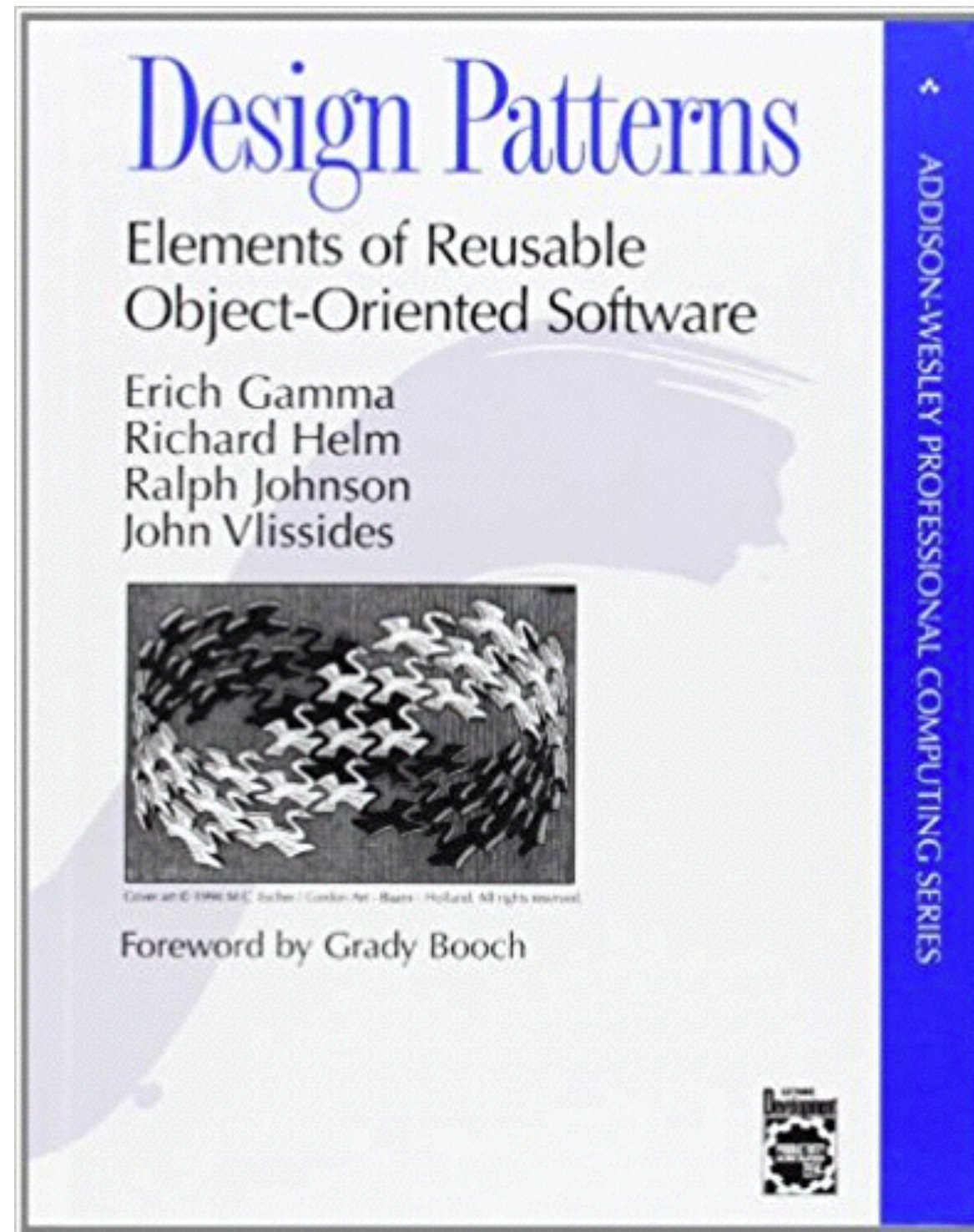
# Perfectly Good Fact, or Broken Struct?

```
(select-keys about-me [:github/location])  
=> {:github/location "Chapel Hill, NC"}
```

# Slots vs. Sets

	Slots	Sets
worldview	closed	open
genericity	no	it's just maps
ad hoc usage	no	carve away!
scaling dev effort	combinatorial	linear

# Celebrate Needless Effort





# Live Coding

Your running program is ***tangible***

query it

transform it

***program it***

# Query the Program

<b>Clojure API</b>	<b>args</b>	<b>returns</b>	<b>count</b>
apropos	stringy	symbols	N
find-doc	stringy	docstrings	N
doc	symbol	docstring	1
source	symbol	source string	1
all-ns	-	namespaces	all
ns-publics	namespacey	map sym->var	all
imports	namespacey	classes	all

# Transform the Program

finish experiments / undo mistakes

without ever leaving your running program

Clojure API	args	change
in-ns	sym	*ns*
def	symbol, init?	new var root
ns-unmap	ns, sym	remove symbol
ns-unalias	ns, sym	remove alias
remove-ns	sym	remove ns

# Codeveloping Two Libs

Load both libs from the REPL

Jump in and study data

including ad-hoc programs

Make changes one def form at a time

Leave test setup in a comment

# Live Coding vs. Reloading

“workflow, reloaded” operates one level higher

files and namespaces, tools help keep track

app state must adhere to certain idioms

live coding is targeted surgery

more precise

developer keeps track

both have utility

# What About GUI Debuggers?

“Off the rack” experience

improved visibility

limited/special transformations

no/weak programmability

Not scissors

# Live Data: Clojure spec

a la carte specificity without sacrificing generality

you get to keep your scissors

dynamic leverage

anytime, anywhere, up to you

fantastic for brownfield development

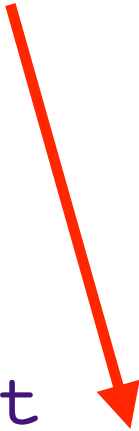
# spec as Exploration Tool

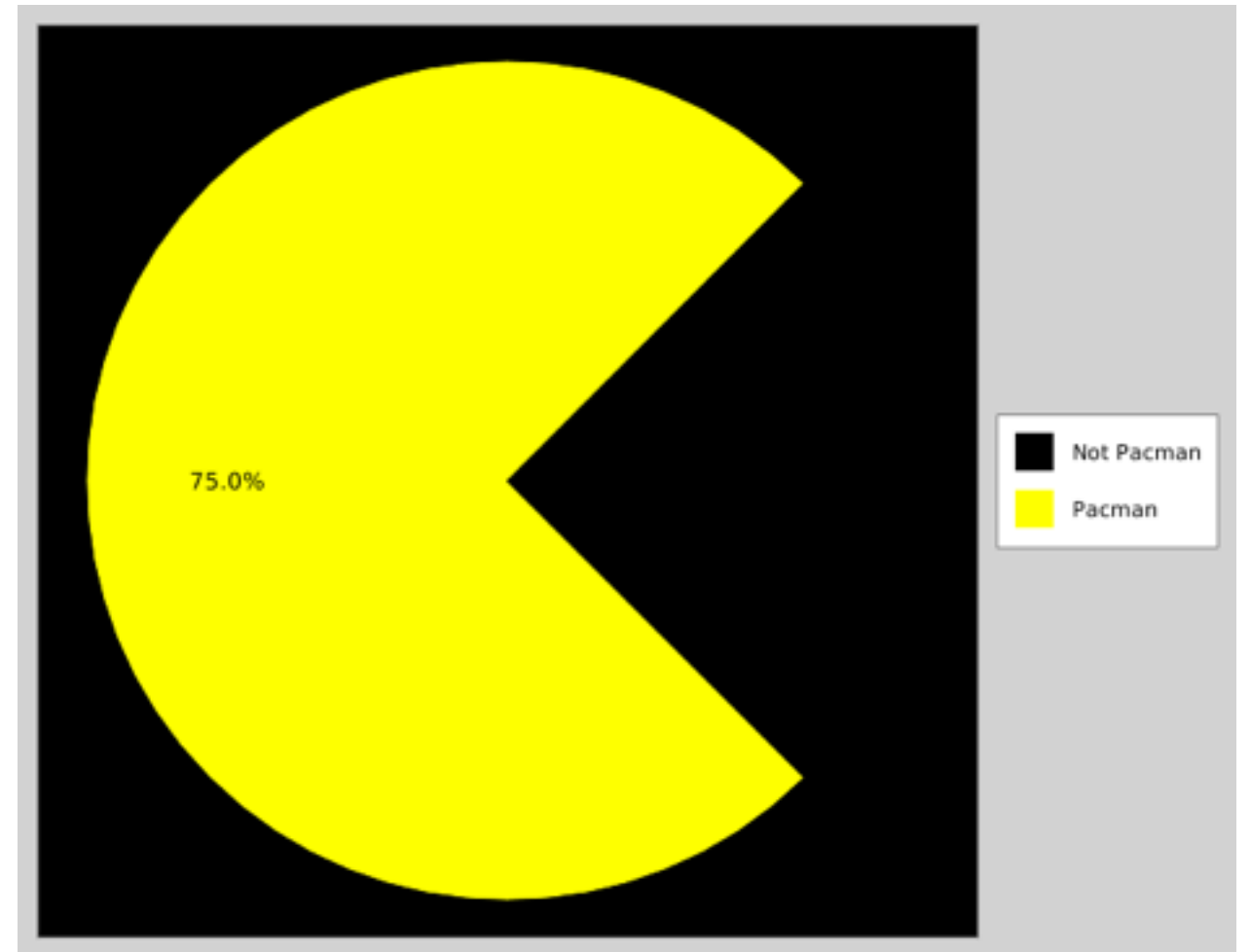
What	How
what are the building blocks?	declarative structure
what invariants hold?	arbitrary predicates
how do I check?	validation
what went wrong?	explanation
what went right?	conformance
docs please	autodoc
examples please	sample generator
am I using this right?	instrumentation
is my code correct?	generative testing
can I recombine pieces like this?	assertion



# clj-xchart

note scissor-ready  
generic data

  
`(c/pie-chart  
 [ [ "Not Pacman" 1/4 ]  
 [ "Pacman" 3/4 ] ]  
 { :start-angle 225.0  
 :plot { :background-color :black }  
 :series [ { :color :black } { :color :yellow } ] } )`



# From Basic Predicates

```
(s/def ::chartable-number (s/and number? finite?))  
(s/def ::x (s/every ::chartable-number :min-count 1))  
(s/def ::y (s/every ::chartable-number :min-count 1))
```

# To Testable Types

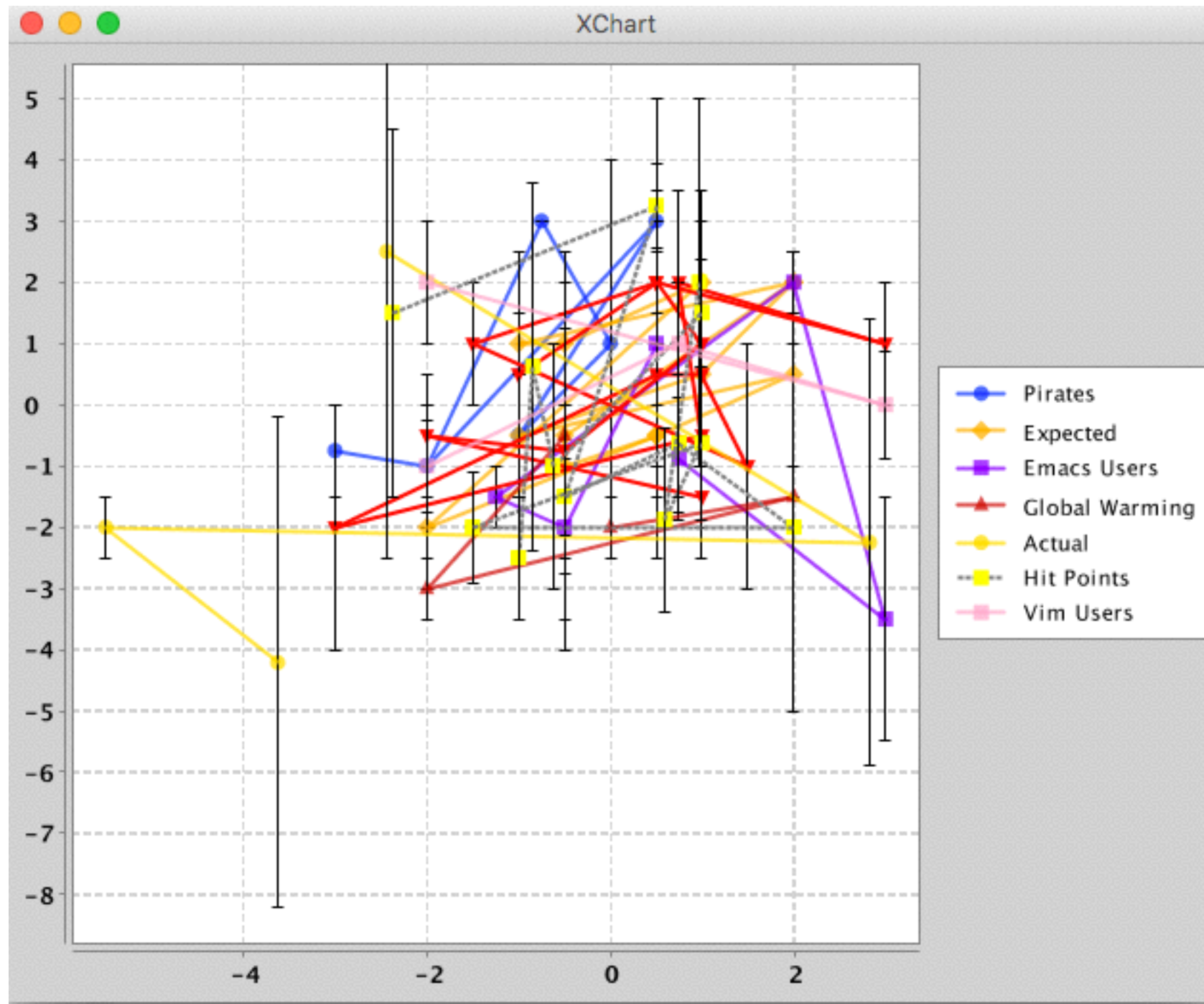
```
(defn axis-counts-match?  
  [{:keys [x y error-bars bubble] :as args}]  
  (= (count (second x))  
     (count y)  
     (count (or error-bars y))  
     (count (or bubble y)))))  
  
(defmethod data-compatible-with-render-style? :area  
  [series]  
  (ordered? (second (:x series)))))
```

# Exercising Data

```
(s/exercise ::series/line-width)  
=> ([2.0 2.0] [23 23] [16 16] [1.0 1.0]  
     [0.5 0.5] [0.9375 0.9375] [74 74] [1 1] [1 1] [39 39])
```

```
(s/exercise ::series/series-name 5 generators)  
=> ([ "Grommets" "Grommets" ] [ "Emacs Users" "Emacs Users" ]  
     [ "Grommets" "Grommets" ] [ "Vim Users" "Vim Users" ]  
     [ "Expected" "Expected" ])
```

# Exercising Code



# Instrumentation

```
(xchart/xy-chart {"bad-chart"  
                  {:x [3 2 1] :y [4 5 7]  
                   :style {:render-style :area}}})
```

=> ExceptionInfo Call to #'com.hypirion.clj-xchart/xy-chart  
did not conform to spec:

```
val: {:x [:numbers [3 2 1]], :y [4 5 7],  
      :style {:render-style :area}}
```

fails spec: :com.hypirion.clj-xchart.specs.series.xy/series-elem

at: [:args :series 1] predicate: data-compatible-with-render-style?

# Reflections

spec-ing (can be) interactive

specs need not be complete or exact

specs work for you, not you for specs

add support where you need/want it

get your exercise!

generative testing found a bug *in the JVM!*

# Running

work inside your program, from a REPL

# with Scissors

precision cut your code and data down to size

# Live coding

against a tangible runtime

# Live data

explore and extend programs with spec



# Running with Scissors: Live Coding with Data

@stuarthalloway