Poetry of Programming

—

How (not) to teach Clojure to beginners

Attila Egri-Nagy
www.egri-nagy.hu

Akita International University, JAPAN

2017

Introduction

Background story

Course design

Running the course

How to code it?

Future

Topic

designing and deploying a full semester Clojure course

1. how (not) to teach Clojure to beginners
2. the cognitive difficulties of problem solving in general
3. trying to understand why I like the language

Overview

Introduction

Background story

Course design

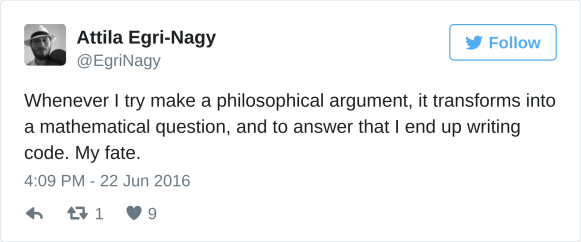
Running the course

How to code it?

Future

Introduction

Who am I?



Attila Egri-Nagy
@EgriNagy

Whenever I try make a philosophical argument, it transforms into a mathematical question, and to answer that I end up writing code. My fate.

4:09 PM - 22 Jun 2016

Software engineer disguised as a mathematician,

- working in applied computational abstract algebra,
- teaching traditional math classes.

Commodore 64 BASIC/assembly
→ Pascal, C (university)
→ JAVA (as a software engineer)
→ GAP `www.gap-system.org` (as an academic researcher)
→ Clojure (for recomputing results)

Akita International University, Japan

- a liberal arts college
- teaching in English

- offering programming courses: C# and Clojure

Background story

Rationale for programming courses

The current hype of teaching everyone coding – controversial issue.

What we can agree about:

> *Every university student (regardless of major) should have at least one programming course.*

In particular at AIU: we are working on a new major 'Digital Studies' — Liberal Arts education getting a digital update.

How did it really happen?

New job, expectation for deploying new courses.

⇓

MAT 240 Mathematics for the Digital World

⇓

Realizing how much work a new course involves, and the expectation for more courses.

⇓

New tactic: a crazy course that is likely to be rejected (either more research time, or a course that is research).

⇓

MAT245 Poetry of Programming – Puzzle-based introduction to Functional Programming

Poetry?

A summer river being crossed
how pleasing
with sandals in my hands!

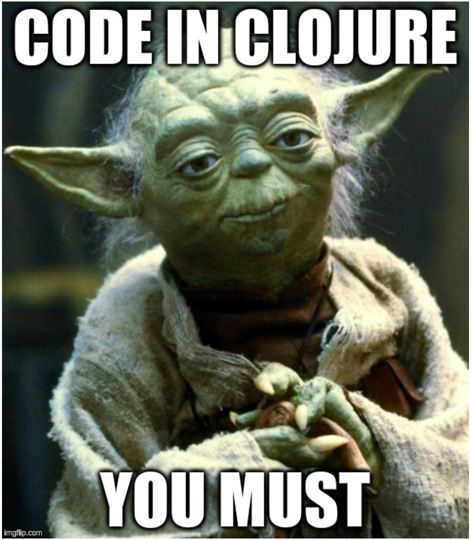
Yosa Buson (1716 – 1784)

Snowclad houses in the night (1778)

```
(fn f [a b]
  (if (zero? b)
    a
    (recur b (mod a b))))
```

Yes! In a limited sense...

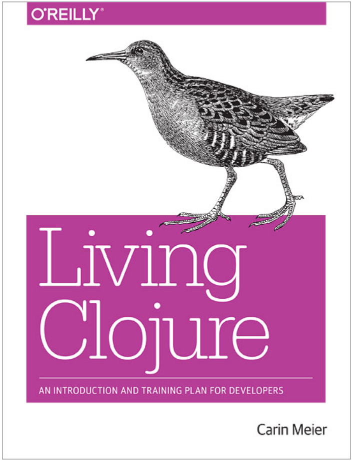
But would the students come?

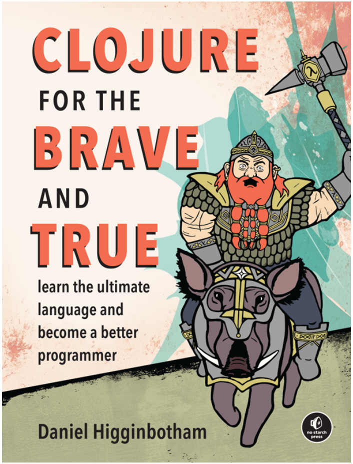


16 registered, 14 finished the course, gender ratio 3:11

Course design

Excellent beginner books on Clojure




CLOJURE FOR THE BRAVE AND TRUE
learn the ultimate language and become a better programmer
Daniel Higginbotham
no starch press

But, their target audience is different.

Teaching total beginners

- shared background knowledge
- focused material, minimized amount
- one concept at a time

Liberal Arts students

shared background knowledge

- high school algebra, real-valued functions
- learning a foreign language

reduced learning material

- no tooling, no IDE, just a REPL
- no Java interop
- the functional core of Clojure

Dependency graph of Clojure functions/programming concepts

We introduce a new concept, when all of its ‘parts’ are already known.

function definition depends on

- lists
- function calls
- `def`
- vectors

The alternative is to ask for postponed comprehension (e.g. `HelloWorld.java`).

Which path maximizes learning speed/fun?

Running the course

Starting point: function composition

Definition (copied from a College Algebra textbook)

The **composite function** $f \circ g$ of two functions f and g is defined by

$$(f \circ g)(x) := f\,(g(x))$$

"If you've had to learn this stuff anyway, why not using it beyond the math course?"

"All you need to do is this..."

$$f(x) \longrightarrow \texttt{(f x)}$$

$$f(g(x)) \longrightarrow \texttt{(f (g x))}$$

Simple functions

$f(x) = x$	`identity`
$f(x) = x + 1$	`inc`
$f(x) = x - 1$	`dec`

The game is on: what can you do with a limited set of functions?

What's the value? – a simple question:

```
((comp inc dec) 1)
```

and a little puzzle:

```
(((comp comp comp) dec dec) 1)
```

This forced me to revisit these functions.

When do we stop learning?

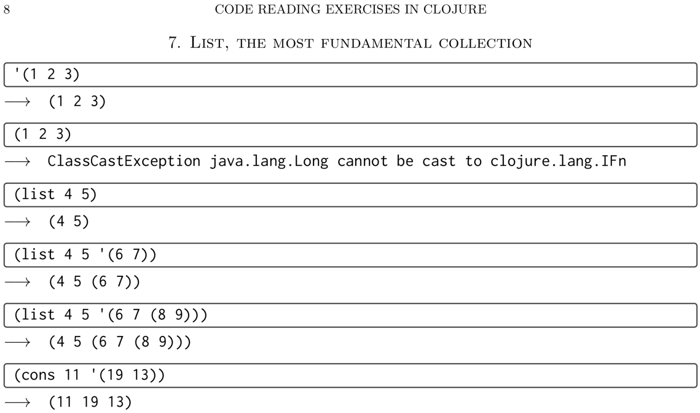
Assessment – mass produced koans

Lab work makes up 40% of final grade.

Mid term and a final exam, 30% each.

Paper based exam?

Reading exercise: evaluate a single Clojure expression.



8 CODE READING EXERCISES IN CLOJURE

7. List, the most fundamental collection

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

⟶ `(1 2 3)`

```
(1 2 3)
```

⟶ `ClassCastException java.lang.Long cannot be cast to clojure.lang.IFn`

```
(list 4 5)
```

⟶ `(4 5)`

```
(list 4 5 '(6 7))
```

⟶ `(4 5 (6 7))`

```
(list 4 5 '(6 7 (8 9)))
```

⟶ `(4 5 (6 7 (8 9)))`

```
(cons 11 '(19 13))
```

⟶ `(11 19 13)`

Function definitions are not urgent

Vectors, hash-maps and hash-sets are functions.

Just a silly exercise:

```
((comp {:b 11 :a 13 :c 17} [:c :b :a] [1 3 2]) 2)
```

and a more sensible one:

```
(zipmap (range 4)
        ["zero" "one" "two" "three"])
{0 "zero", 1 "one", 2 "two", 3 "three"}
```

Immutability – not an issue

New programmers do not need to unlearn old habits.

```
(def v [1 2 3])

(conj v 4)
[1 2 3 4]
v
[1 2 3]
```

Not surprised? then no worries!

Metaphors for `def`

- Creating long term memories.
- Attaching meaning to words.

Both hint towards not using them as mutable variables.

Metaphors as knowledge transferring cognitive tools:
Metaphors We Compute By, Alvaro Videla @ClojuTRE 2017

Defining functions - abstracting over calculations

playing in the REPL: several examples of the same calculation

```
(* 2 2)
4
(* 12 12)
144
```

finding what changes, identifying the ‘moving part’ (algebraic abstraction)

```
((fn [x] (* x x)) 2)
4
((fn [x] (* x x)) 12)
144
(map (fn [x] (* x x)) [1 2 3 4 5])
(1 4 9 16 25)
```

then giving it a name

```
(defn square
  [x]
  (* x x))
```

and using it anywhere

```
(square 2)
4
(square 12)
144
(map square [1 2 3 4 5])
(1 4 9 16 25)
```

This seemed easy…

Roadblock

Leaving the comfort zone (arithmetic functions): functions working with strings and numbers.

```
(defn rect-info
  [a b]
  (str "A rectangle with sides " a ", " b
       " has area " (* a b) "."))
```

Same in mathematics: the problem is that we are not working with familiar objects.

⇓

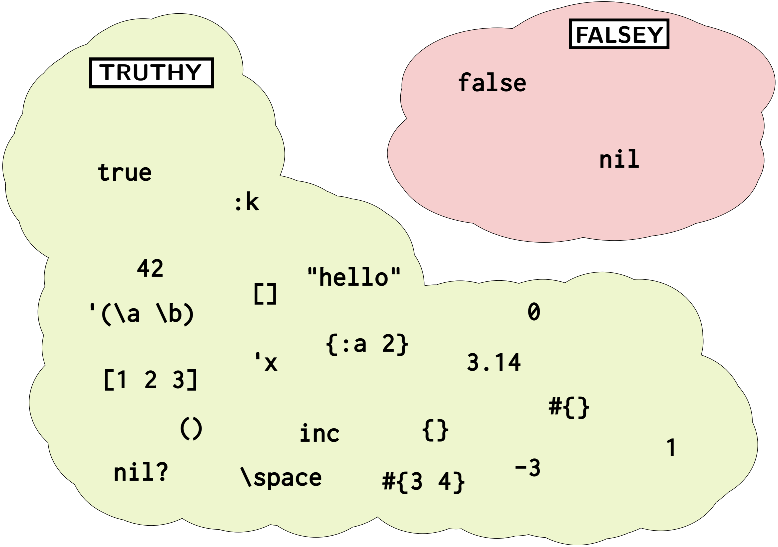
More efforts on bridging the abstract realm and the everyday experience.

(filter orange? fruits)

Separate the oranges from other fruits.

```
(filter even? [1 2 3 4 5 6])
(2 4 6)
```

what makes `filter` great



TRUTHY

`true` `:k`

`42` `"hello"` `[]` `'(\a \b)` `0`

`{:a 2}` `'x` `3.14` `[1 2 3]` `#{}`

`()` `inc` `{}` `1`

`nil?` `\space` `#{3 4}` `-3`

FALSEY

`false`

`nil`

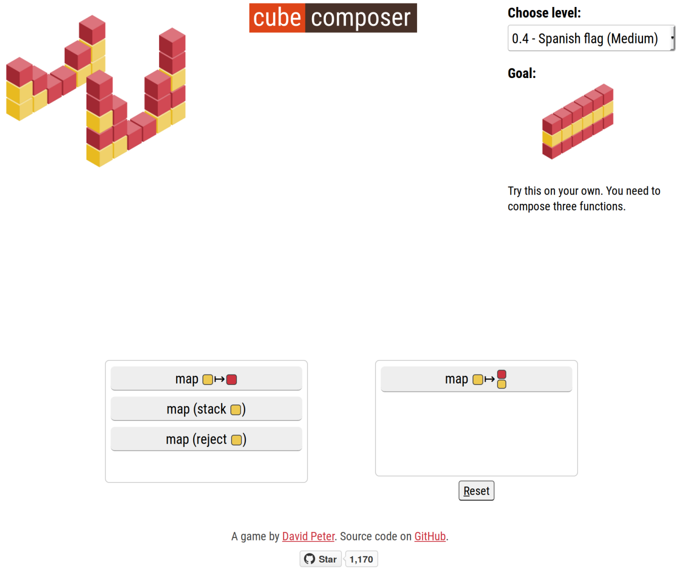
(map peel oranges)

Peel all the oranges.

```
(map square [1 2 3 4])
(1 4 9 16)
```

need to deal with a single element only, collection processing is automated

Let’s play!



cube composer

Choose level: 0.4 - Spanish flag (Medium)

Goal:

Try this on your own. You need to compose three functions.

A game by David Peter. Source code on GitHub.

https://david-peter.de/cube-composer/

(reduce cut empty-bowl fruits)

Cut all the fruits into a bowl.

```
(reduce conj [] (range 3))
[0 1 2]
```

Teaching reduce

Metaphors: on the beach a child collecting pebbles in a bucket, or just looking for the most beautiful pebble.

However the big thing is reductions – it shows the process.

```
cljs.user=> (reduce conj [:a :b] [:c :d :e])
[:a :b :c :d :e]
cljs.user=> (reductions conj [:a :b] [:c :d :e])
([:a :b] [:a :b :c] [:a :b :c :d] [:a :b :c :d :e])
cljs.user=> (reduce max [1 2 1 3 2 5 4 6 1 2])
6
cljs.user=> (reductions max [1 2 1 3 2 5 4 6 1 2])
(1 2 2 3 3 5 5 6 6 6)
cljs.user=> (reduce + [1 2 3 4])
10
cljs.user=> (reductions + [1 2 3 4])
(1 3 6 10)
```

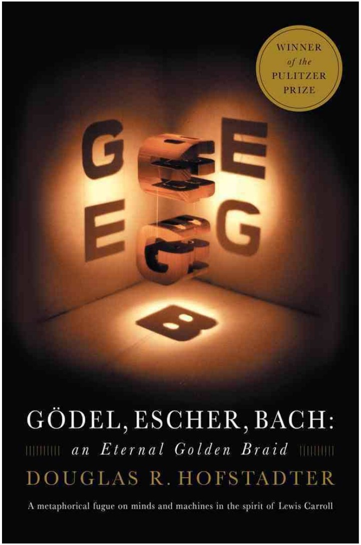
Reimplementing map: recursion vs. reduce

```
(defn MAP
  "recursive implementation of map"
  [f coll]
  (if (empty? coll)
    ()
    (cons (f (first coll))
          (MAP f (rest coll)))))
```

```
 (defn MAP2
  "implementing map by reduce"
  [f coll]
  (reduce (fn [x y]
              (conj x (f y)))
          []
          coll))
```

Which one is nicer? better? (in what sense?) easier?

blinded by recursion


WINNER of the PULITZER PRIZE
GÖDEL, ESCHER, BACH:
an Eternal Golden Braid
DOUGLAS R. HOFSTADTER
A metaphorical fugue on minds and machines in the spirit of Lewis Carroll

LEONARDO DICAPRIO
WATANABE GORDON-LEVITT COTILLARD PAGE HARDY MURPHY BERENGER AND CAINE
YOUR MIND IS THE SCENE OF THE CRIME.
INCEPTION
FROM THE DIRECTOR OF THE DARK KNIGHT
EXPERIENCE IT JULY 16 IN THEATERS AND IMAX

next time: `reduce first`

Schedule - 15 weeks, 2 classes/week, 37.5 hours

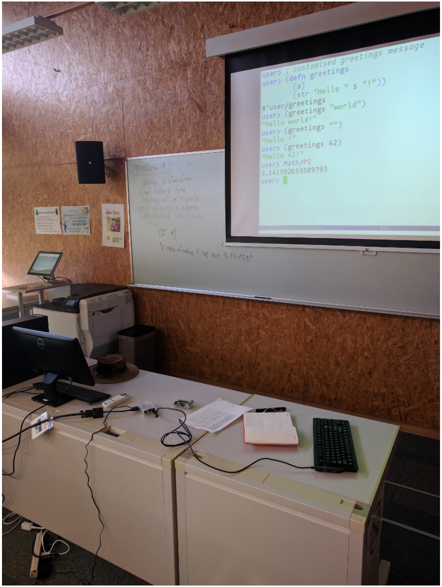
1. `identity`, `inc`, arithmetic operators, comp, predicates
2. quote, `list`, `cons`, `first`, `rest`, `last`, vectors, cons vs. `conj`, def
3. defn, `str`, map, apply, `if`
4. range, `filter`
5. laziness, `range`, `iterate`, `take`, `take-while`, Collatz conjecture
6. recursion, `fn`, reimplement count, map
7. reimplement `filter`, introducing `let`, `clojure.string`
8. hash-maps, hash-sets
9. using hash-maps, truthy and falsey
10. `concat`, `mapcat`.
11. tuples and permutations, arithmetic puzzle (brute-force)
12. `max-key`, sequential destructuring, reducing/folding
13. `reduce`, arithmetic puzzle again, point-free style
14. rand, println, for, doseq, Quil
15. Caesar-shift cipher

Class format

$\frac{1}{3}$ livecoding (instead of lecturing): Raspberry Pi, Emacs + Cider

$\frac{2}{3}$ problem solving, “patrolling”

The best thing that can happen: students helping each other.



```
user> ; customised greetings message
user> (defn greetings
        [s]
        (str "Hello " s "!"))
#'user/greetings
user> (greetings "world")
"Hello world!"
user> (greetings "")
"Hello !"
user> (greetings 42)
"Hello 42!"
user> Math/PI
3.141592653589793
user> 
```

Problem 1

Write a function that takes the radius of a circle and outputs a string describing the circle.

e.g.

(f 1)

"A circle of radius 1 has area 3.14159"

An example problem: Collatz conjecture

Iterating the function

$$\text{collatz}(x) = \begin{cases} 3x + 1 & \text{if } x \text{ is odd} \\ \frac{x}{2} & \text{if } x \text{ is even} \end{cases}$$

gives sequences that seem to always end in 1.

What number between 1 and 1000 produces the longest sequence?

```
(defn collatz [n]
  (if (even? n)
      (/ n 2)
      (inc (* n 3))))

(defn c-length [n]
  (count (take-while #(not= 1 %)
                     (iterate collatz n))))
```

Code Transformations

answering the longest run question:

```
(apply max (map c-length (range 1 1001)))
178
(filter #(= 178 (c-length %)) (range 1 1001))
(871)
```

transformed by introducing max-key

```
(apply max-key c-length (range 1 1001))
```

Difficulty: apply

`(max [1 3 2])` vs. `(apply max [1 3 2])`

Helping metaphor: container data structures are packaging.

If apples are in a bag, you can't eat them directly.

especially confusing

```
(apply + [1 2 3 4])
```

```
(reduce + [1 2 3 4]])
```

Complicated solutions

Task: Write a function that multiplies a number by 10.

Student: Here is the solution.

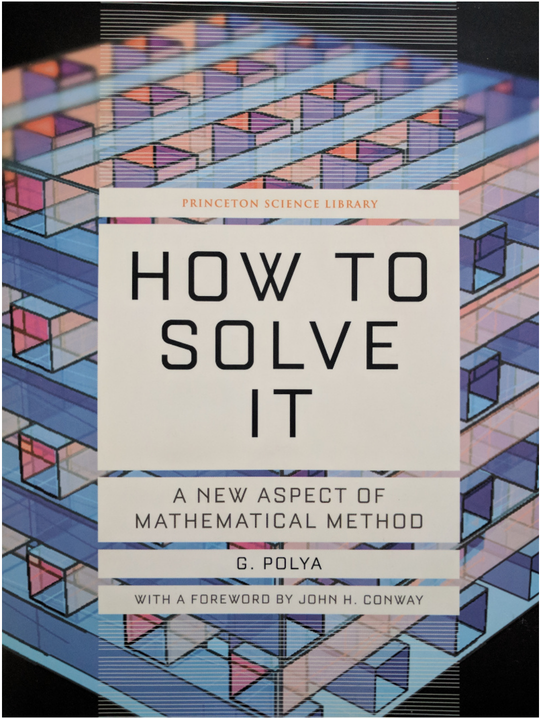
```
(defn f
   [x]
   ((fn [x] (* x 10)) x))
```

Me: This can be simplified. `(defn f [x] (* x 10))`

Student: (checks in REPL) Yes, but then I don't understand.

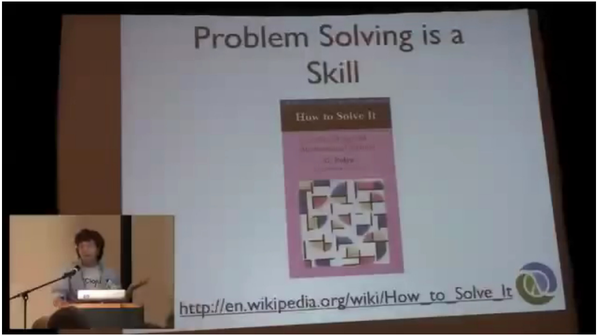
Me: (thinking hard, figuring out what to say)

How to code it?


PRINCETON SCIENCE LIBRARY
HOW TO SOLVE IT
A NEW ASPECT OF MATHEMATICAL METHOD
G. POLYA
WITH A FOREWORD BY JOHN H. CONWAY

You know this book...

...since 5 years ago you were told to buy it immediately.



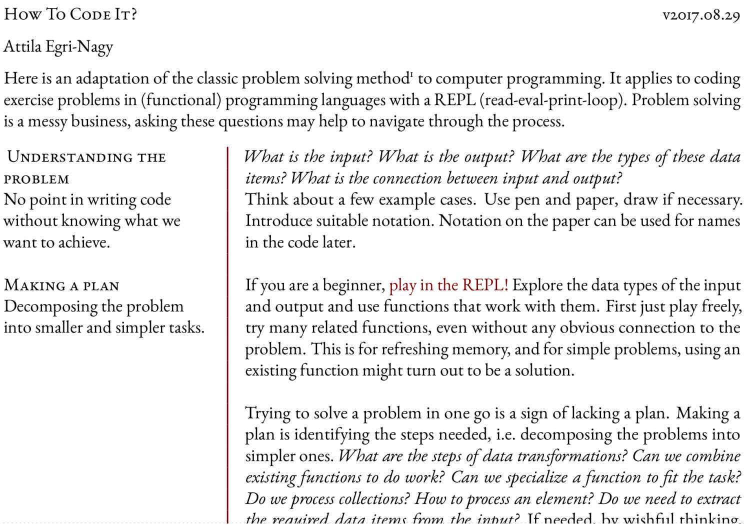
Problem Solving is a Skill

http://en.wikipedia.org/wiki/How_to_Solve_It

Rich Hickey: Hammock Driven Development, Clojure/Conj 2012

Math ⟶ Programming

Most ideas translate easily.



How To Code It? v2017.08.29

Attila Egri-Nagy

Here is an adaptation of the classic problem solving method[1] to computer programming. It applies to coding exercise problems in (functional) programming languages with a REPL (read-eval-print-loop). Problem solving is a messy business, asking these questions may help to navigate through the process.

Understanding the problem
No point in writing code without knowing what we want to achieve.

What is the input? What is the output? What are the types of these data items? What is the connection between input and output?
Think about a few example cases. Use pen and paper, draw if necessary. Introduce suitable notation. Notation on the paper can be used for names in the code later.

If you are a beginner, play in the REPL! Explore the data types of the input and output and use functions that work with them. First just play freely, try many related functions, even without any obvious connection to the problem. This is for refreshing memory, and for simple problems, using an existing function might turn out to be a solution.

Making a plan
Decomposing the problem into smaller and simpler tasks.

Trying to solve a problem in one go is a sign of lacking a plan. Making a plan is identifying the steps needed, i.e. decomposing the problems into simpler ones. *What are the steps of data transformations? Can we combine existing functions to do work? Can we specialize a function to fit the task? Do we process collections? How to process an element? Do we need to extract the required data items from the input?* If needed, by wishful thinking,

Steps of problem solving

Understanding the problem No point in writing code without knowing what we want to achieve.

Making a plan Decomposing the problem into smaller and simpler tasks. **Play in the REPL!**

Carrying out the plan Write code, test code, write code, test code...

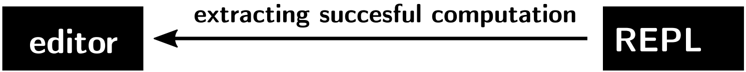
Looking back Check, reflect and learn. Rewrite for improvement.

Play in the REPL!

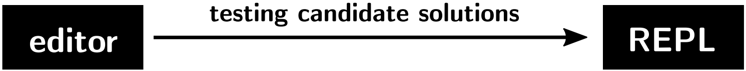
- finding pieces of a solution
- mobilizing knowledge

The REPL may not be as natural as we would like to think.

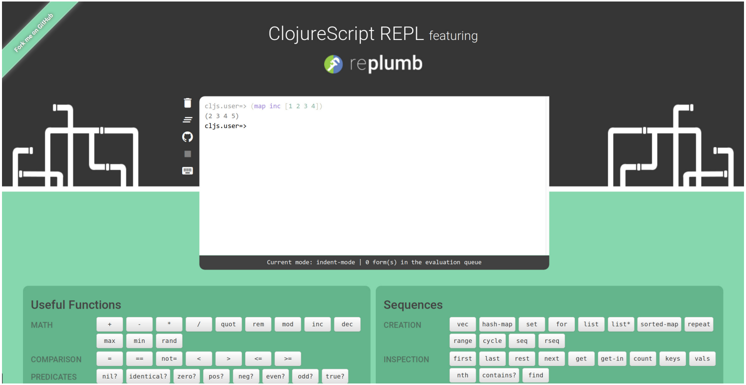
Beginner's workflow:



Professional's workflow:



First REPL: `clojurescript.io`

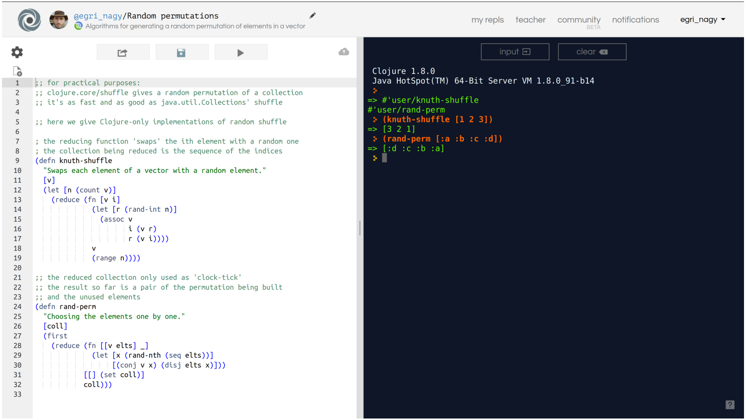


```
cljs.user=> (map inc [1 2 3 4])
(2 3 4 5)
cljs.user=>
```

Current mode: indent-mode | 0 form(s) in the evaluation queue

need for saving solutions

Editor is needed: `repl.it` interface


@egri_nagy/Random permutations
Algorithms for generating a random permutation of elements in a vector
my repls
teacher
community
BETA
notifications
egri_nagy
input
clear

```
;; for practical purposes:
;; clojure.core/shuffle gives a random permutation of a collection
;; it's as fast and as good as java.util.Collections' shuffle

;; here we give Clojure-only implementations of random shuffle

; the reducing function 'swaps' the ith element with a random one
; the collection being reduced is the sequence of the indices
(defn knuth-shuffle
  "Swaps each element of a vector with a random element."
  [v]
  (let [n (count v)]
    (reduce (fn [v i]
              (let [r (rand-int n)]
                (assoc v
                       i (v r)
                       r (v i))))
              v
              (range n))))

;; the reduced collection only used as 'clock-tick'
;; the result so far is a pair of the permutation being built
;; and the unused elements
(defn rand-perm
  "Choosing the elements one by one."
  [coll]
  (first
    (reduce (fn [[v elts] _]
              (let [x (rand-nth (seq elts))]
                   [(conj v x) (disj elts x)]))
            [[] (set coll)]
            coll)))
```

```
 Clojure 1.8.0
 Java HotSpot(TM) 64-Bit Server VM 1.8.0_91-b14
 ›
=> #'user/knuth-shuffle
#'user/rand-perm
 › (knuth-shuffle [1 2 3])
=> [3 2 1]
 › (rand-perm [:a :b :c :d])
=> [:d :c :b :a]
 ›
```

editor/buffer

console/REPL

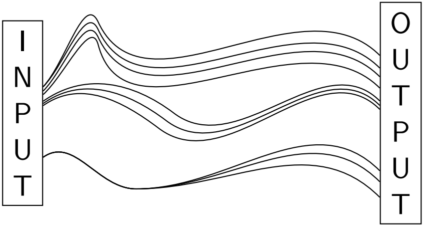
Problem: students give up the dynamism of the REPL.

Documentation

- inline documentation – geared towards professionals (maybe naturally so)
- `clojuredocs.org` – very useful, but simple and advanced examples are in no particular order
- the weird characters guide – awesome
- stackoverflow effect – it leads to solutions but not to understanding

Solution spaces

There are several different solutions for the same problem!



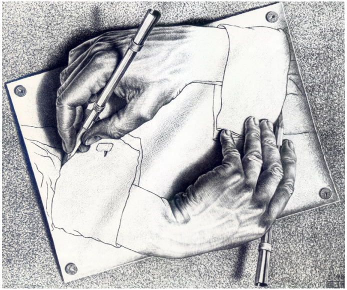
```
(defn abs [x]
  (if (>= x 0) x (- x)))
```

```
(defn abs2 [x]
  (max (- x) x))
```

Future

Improving the course

- reactive → proactive
- adding a 'summit'
  - Algebra: Euler's equation $e^{\pi i} + 1 = 0$
  - Calculus: The Fundamental Theorem of Calculus (integration is inverse derivation)
  - Lisp/Clojure: metacircular evaluator



- Tooling: NightCode/Parinfer

Summary

What is achievable in one semester?

Using `map`, `filter` and `reduce` in solving math puzzles, programming exercises.

Suggestions (for everyone):

- the learning process never finishes
  - maintain a beginner's mindset, or
  - expose yourself to beginner thinking (by teaching)
- metacognition: think about your coding
- think of a solution space, not just a single path in it

Conclusion

Your love of Clojure cannot be destroyed by deconstruction.

Poetry of Programming - course material

`https://egri-nagy.github.io/popbook/`

Conclusion

Your love of Clojure cannot be destroyed by deconstruction.

Poetry of Programming - course material

`https://egri-nagy.github.io/popbook/`

Thank You!