



St. Louis Clojure

Clojure Schemata and Generators

Christopher Mark Gore

cgore.com

Tuesday, September 29, AD 2015

**We Write Clojure at The Climate Corporation,
And We're Hiring!**



Add Stuff to Your `project.clj`

```
1 (defproject your-project "1.2.3"  
2   :description "It's like totally awesome and stuff"  
3   ;; ...  
4   :dependencies [;;...  
5                   [prismatic/schema "0.4.3"]  
6                   [org.clojure/test.check "0.7.0"]  
7                   ;; ...  
8                 ]  
9   ;; ...  
10  )
```

Prismatic Schema

Schemata^a are sort of like types, but only as strict as you want them to be at that specific moment, so no type hell.

```
1 (ns schema-stuff
2   (:require [schema.core :as s]))
3
4 (s/validate s/Num 42)
5 (s/validate s/Str "howza")
6 (s/validate s/Keyword :hey)
```

^aThe plural of *schema* is *schemata*, not *schemas*.

Clojure `test.check` and Generators

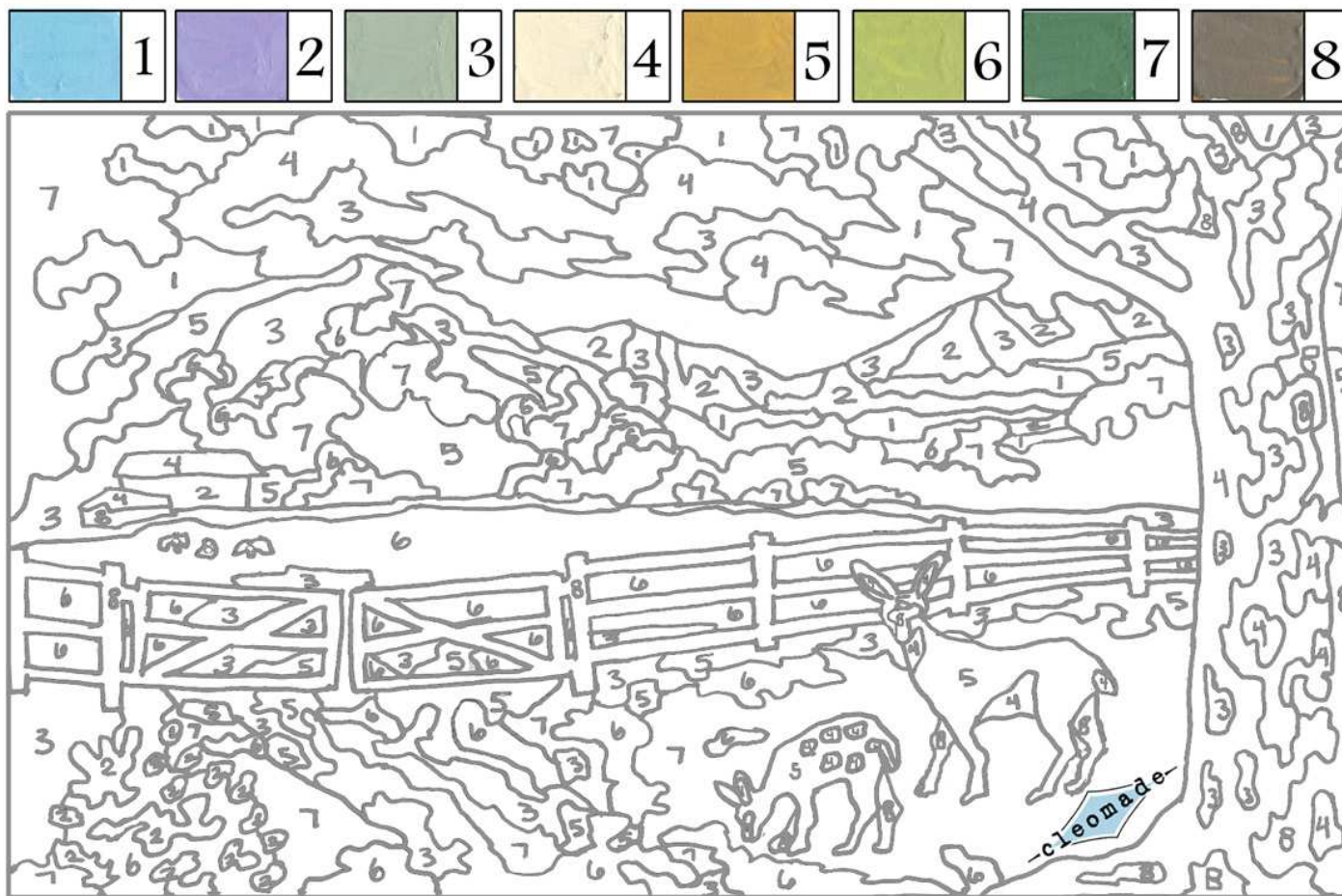
Generators make random examples according to a definition. It's a great way to make test data without brittle hand-rolled examples.

```
1 (ns gen-stuff
2   (:require [clojure.test.check :as tc]
3              [clojure.test.check.generators :as gen]
4              [clojure.test.check.properties :as prop]
5
6   (gen/sample gen/int)
7   ;; => (0 1 -1 0 -1 4 4 2 7 1)
8   (gen/sample gen/int 20)
9   ;; => (0 1 1 0 2 -4 0 5 -7 -8 4 5 3 11 -9 -4 6 -5 -3 0))
```

Schemata + Generators = Awesome!

- Schemata to validate function input
 - Definitely in tests.
 - Maybe even in production.
- Generators to fuzz the function in tests.
- Feed the generators into the schemata.
 - Check the generator against the schema.
 - Check the schema with the generator.

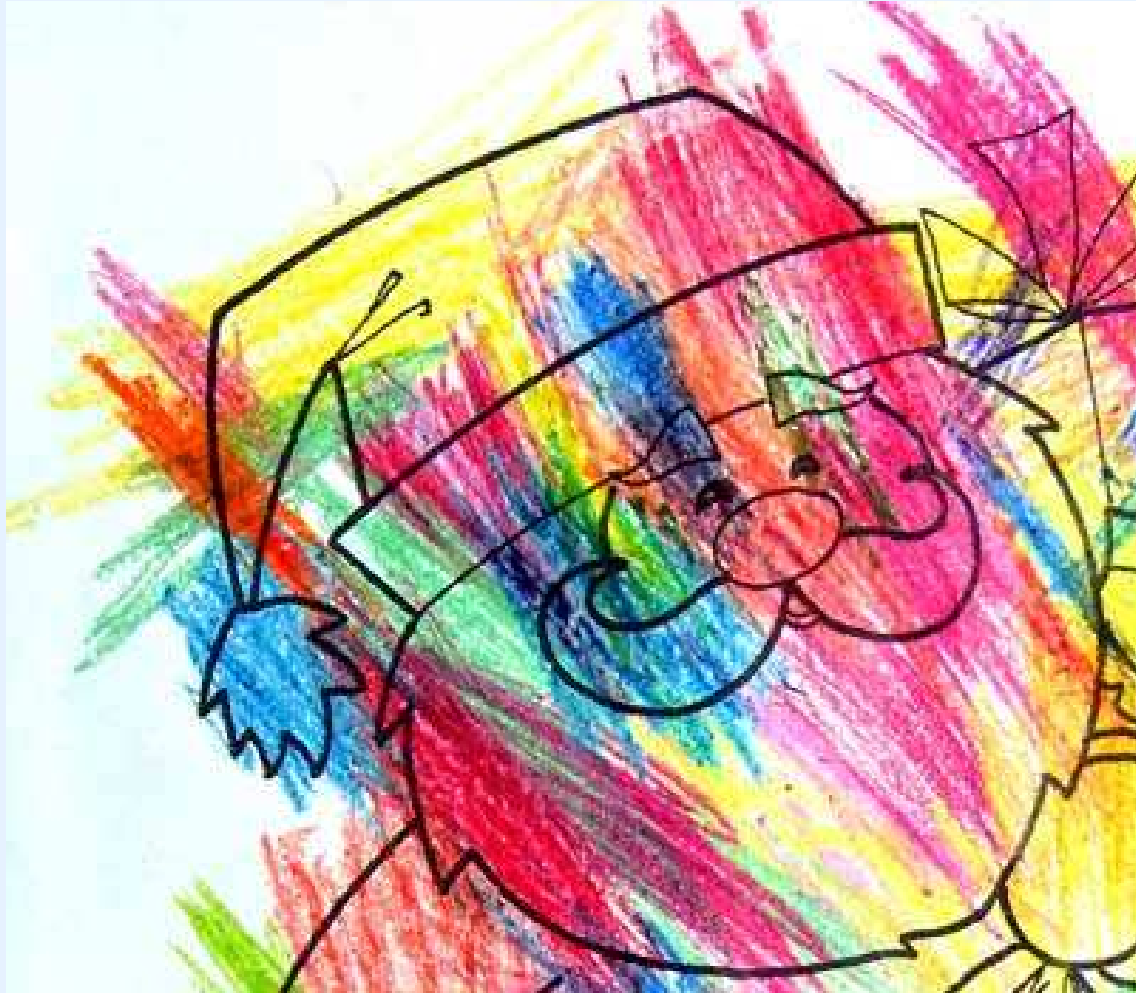
Schemata are These



Generators are These



Used Together, We Catch When Our Code
Does This



Schema: validate **versus** check

The two most important functions for schema checks are `validate` and `check`, the only real difference being that `validate` raises an error and `check` does not.

```
1 ;; A schema for integers.
2 (s/validate s/Int 42) ; passes, returns 42
3 (try (s/validate s/Int "nope") ; fails, throws an error.
4     (catch Exception e))
5
6 (s/check s/Int 42) ; => nil
7 (s/check s/Int "nope") ; => (not (integer? "nope"))
```

Test Check Properties

We define properties we expect to always hold, and assert those properties.

```
1 (def prop-addition-increments
2   (prop/for-all [a gen/int
3                  b gen/int]
4                  (>= (+ a b) a))) ; This is always true.
5 ;; Check 100 times
6 (tc/quick-check 100 prop-addition-increments)
7 ;; FAIL! We forgot negatives!
8 {:result false, :seed 1434746134125, :failing-size 2,
9  :num-tests 3, :fail [1 -2],
10 :shrunk {:total-nodes-visited 4, :depth 2, :result false,
11          :smallest [0 -1]}}
```

Test Check Properties

We discover the *real properties* of our system this way, not just what we *think* they are.

$$[a + b \geq a] \forall a, b \in \mathbb{N} = \mathbb{Z} \cap [0, \infty)$$

```
1 ;; We meant for natural numbers [0,...)a
2 (def prop-addition-increments-for-nat
3   (prop/for-all [a gen/nat
4                  b gen/nat]
5                 (>= (+ a b) a))) ; This is REALLY true
6 ;; Check 100 times
7 (tc/quick-check 100 prop-addition-increments-for-nat)
8 ;; => {:result true, :num-tests 100, :seed 1434746600412}
```

^aSomebody with a Ph.D. in mathematics might have told you that 0 isn't a natural number: they are wrong.

Our Schemata are Our Properties

Our schema must accept *all* instances, if not, it's not a valid schema, therefore we can state that the schema is a property *for all* of our generated examples.

```
1 (def Person {:name s/Str
2              :age s/Int}) ; We'll make s/Nat in a bit.
3 (def person (gen/hash-map :name gen/string
4                           :age gen/nat))
5 (def prop-person-generates-Person
6   (prop/for-all [p person]
7                 (s/validate Person p)))
8 (tc/quick-check 100 prop-person-generates-Person)
```

Integrating `test.check` and `clojure.test`

There is a `defspec` macro to parallel `deftest` at `clojure.test.check.clojure-test/defspec`.

```
1 (defspec first-element-is-min-after-sorting
2     100 ; the number of iterations
3     (prop/for-all [v (gen/not-empty
4                       (gen/vector gen/int))])
5     (= (apply min v)
6        (first (sort v)))))
```

Names are Great, Two Are Better!

What is it	Schema	Generator
strings	s/Str	gen/string
real numbers	s/Num	<i>missing</i>
$\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$	s/Int	gen/int
$\mathbb{N} = \{0, 1, 2, \dots\}$	<i>missing</i>	gen/nat OR gen/pos-int
$\mathbb{Z}^+ = \mathbb{N} \setminus \{0\}$	<i>missing</i>	gen/s-pos-int
\mathbb{Z}^-	<i>missing</i>	gen/s-neg-int
$\mathbb{Z}^- \cup \{0\}$	<i>missing</i>	gen/neg-int

That's a Lot of Missing Things!



Existing Generators: Bits and Bytes

gen/boolean Either true or false.

```
1      (gen/sample gen/boolean)
2      ;; => (false true false true true
3      ;;      false false false false true)
```

gen/byte A single java.lang.Byte.

```
1      (gen/sample gen/byte)
2      ;; => (-88 101 101 104 24 -37 -36 9 20 107)
```

gen/bytes A byte-array.

```
1      (gen/sample gen/bytes)
2      ;; => (#<byte[] [B@2efce23e> ...)
```

Existing Generators: Numbers

gen/int Any integer.

```
1      (gen/sample gen/int)
2      ;; => (0 -1 1 2 3 -2 -3 3 5 -2)
```

gen/choose Numbers in the specified range.

```
1      (gen/sample (gen/choose 18 45))
2      ;; => (24 34 33 37 27 29 29 32 44 18)
```

gen/nat Natural numbers: positive integers and 0.

gen/pos-int **and** gen/neg-int Positive-only or negative-only integers, allowing for 0.

gen/s-pos-int **and** gen/s-neg-int Positive-only or negative-only integers, **not** allowing for 0.

Existing Generators: Characters and Strings

`gen/char` Any character.

`gen/char-ascii` ASCII-only characters.

`gen/char-alphanumeric` Alphanumeric characters, a–z, A–Z and 0–9.

`gen/char-alpha` Alpha-only characters, a–z and A–Z.

`gen/string` Any string, *including weird characters*.

`gen/string-ascii` ASCII-only strings.

```
1      (gen/sample gen/string-ascii)
2      ;; => (" " " " "qc" "I-k" "F" " "
3      ;;      ", Ou" "6kT]<" "}" `!" "5ZH=v75")
```

`gen/string-alphanumeric` Alphanumeric strings.

Existing Generators: Collections

`gen/tuple` A vector in a specific order.

```
1      (gen/sample (gen/tuple gen/int gen/string-ascii))
2      ;; => ([0 "" ] [0 "" ] [-2 "KY"] [0 "J\\"] [4 "n"]
3      ;;      [-2 "\\"] [-2 ";%HJCM"] [-2 "QwD"]
4      ;;      [1 "]KLY|P"] [6 "g"])
```

`gen/vector` A vector of generated things.

```
1      (gen/sample (gen/vector gen/int))
2      ;; => ([] [] [] [1] [] [0 5 -1 0 5] [5 5 3]
3      ;;      [-6 3 -1] [7 -7 -6 2 0 -5 3 -7]
4      ;;      [-9 -4 4 -6 -5 0])
```

`gen/list` A list of generated things, instead of a vector.

Existing Generators: Collections

`gen/shuffle` Randomly permute a sequence.

```
1      (gen/sample (gen/shuffle [1 2 3]))
2      ;; => ([1 2 3] [3 1 2] [1 2 3] [2 3 1] [1 3 2]
3      ;;      [2 1 3] [2 1 3] [2 1 3] [1 3 2] [3 1 2])
```

`gen/map` Generate maps with both the key and value being generated.

```
1      (gen/sample
2      (gen/map (gen/elements [:bibbidi :bobbidi :boo])
3                gen/int))
4      ;; => ({:bobbidi -1} {})
5      ;;      {:bobbidi -2, :bibbidi -3}
6      ;;      {:boo 2, :bobbidi -2}
7      ;;      {:boo -2, :bobbidi -1}
8      ;;      {:boo -1, :bibbidi -5} {} {} {:bibbidi 2})
```

Existing Generators: Collections

gen/hash-map You'll use this a lot.

```
1      (gen/sample
2        (gen/hash-map :bibbidi gen/int
3                      :bobbidi gen/string-ascii
4                      :boo (gen/return 4077)))
5      ;; => ({:boo 4077, :bobbidi "", :bibbidi 0}
6      ;;      {:boo 4077, :bobbidi "H", :bibbidi 0}
7      ;;      {:boo 4077, :bobbidi "", :bibbidi 0}
8      ;;      {:boo 4077, :bobbidi "8B", :bibbidi -3}
9      ;;      {:boo 4077, :bobbidi "OY", :bibbidi 1}
10     ;;      {:boo 4077, :bobbidi "a&)", :bibbidi -5}
11     ;;      {:boo 4077, :bobbidi "", :bibbidi 0} ...)
```


Modifying Existing Generators

`gen/not-empty`

`gen/no-shrink`

`gen/such-that`

`gen/sized`

`gen/resize`

Making New Generators

gen/return Always the same thing.

```
1      (gen/sample (gen/return 42))
2      ;; => (42 42 42 42 42 42 42 42 42 42)
```

gen/one-of *Either this or that.*

```
1      (gen/sample (gen/one-of [gen/int gen/string-ascii]))
2      ;; => (" " "$" 1 2 "tk" "H"]=7" 1 0 -4 -6)
```

gen/frequency Same as gen/one-of, but with set probabilities.

```
1      (gen/sample (gen/frequency [[7 gen/int]
2                                   [3 gen/string-ascii]]))
3      ;; => (" " 1 1 ".P!" 0 3 -5 4 6 -3)
```

gen/bind

Using `gen/elements`

Randomly pick (without exhaustion) from a collection.

```
1 (gen/sample
2   (gen/elements [:spades :diamonds :hearts :clubs]))
3 ;; => (:diamonds :clubs :spades :clubs :hearts
4 ;;      :spades :spades :hearts :hearts :hearts)
```

Making New Generators With `gen/fmap`

If nothing makes sense to generate your stuff, there's always `gen/fmap`, and then you can use any function you want. Huzzah, Clojure!

```
1 (def even-and-positive (gen/fmap #(* 2 %) gen/pos-int))
2 (gen/sample even-and-positive 20)
3 ;; => (0 0 2 0 8 6 4 12 4 18 10 0 8 2 16 16 6 4 10 4)
4 (def gen-double (gen/fmap rand gen/int))
5 (gen/sample gen-double)
6 ;; => (0.0                                0.0
7 ;;      0.8433531349313175 -0.5407298249526976
8 ;;      0.7282154724842486 -0.5111220285736056
9 ;;      -1.6998294599186186  1.4744104363479704
10 ;;      2.094621081981671   -1.7704991357273019)
```

Questions?