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`

```
(defproject your-project "1.2.3"
  :description "It's␣like␣totally␣awesome␣and␣stuff"
  ;; ...
  :dependencies [;;...
                 [prismatic/schema "0.4.3"]
                 [org.clojure/test.check "0.7.0"]
                 ;; ...
                 ]
  ;; ...
  )
```

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.

```clojure
(ns schema-stuff
  (:require [schema.core :as s]))

(s/validate s/Num 42)
(s/validate s/Str "howza")
(s/validate s/Keyword :hey)
```

[a] The 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.

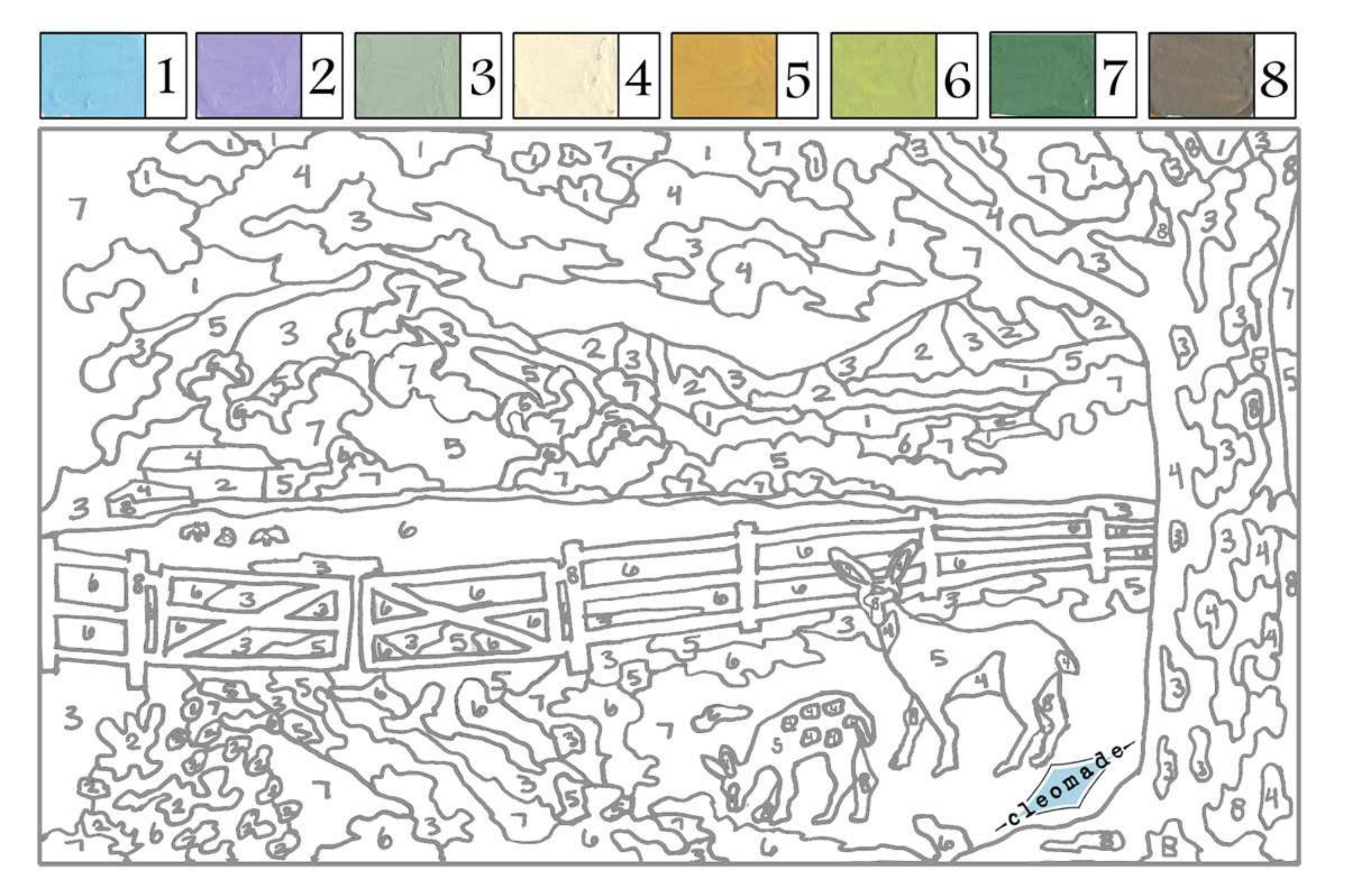
```
(ns gen-stuff
  (:require [clojure.test.check :as tc]
            [clojure.test.check.generators :as gen]
            [clojure.test.check.properties :as prop]

(gen/sample gen/int)
;; => (0 1 -1 0 -1 4 4 2 7 1)
(gen/sample gen/int 20)
;; => (0 1 1 0 2 -4 0 5 -7 -8 4 5 3 11 -9 -4 6 -5 -3 0)
```

Schemata + Generators = Awesome!

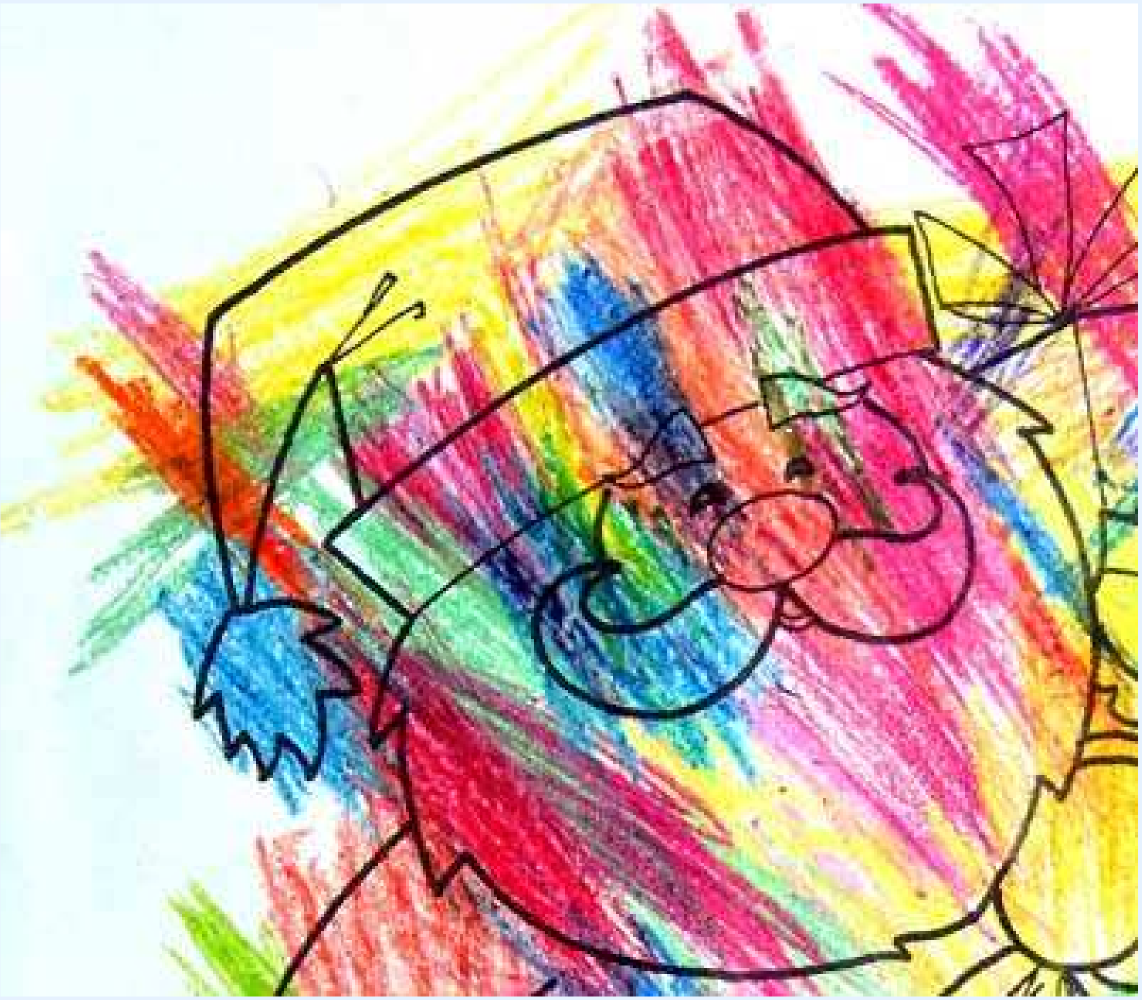
- Schemata to validate function input
  - Definitely in tests.
  - Maybe even in production.
- Generators to fuzz the function in tests.
- Feed the generators into the schemata.
  - Check the generator against the schema.
  - Check the schema with the generator.

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.

```clojure
;; A schema for integers.
(s/validate s/Int 42) ; passes, returns 42
(try (s/validate s/Int "nope") ; fails, throws an error.
     (catch Exception e))

(s/check s/Int 42) ; => nil
(s/check s/Int "nope") ; => (not (integer? "nope"))
```

Test Check Properties

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

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

Test Check Properties

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

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

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

[a] Somebody 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.

```
(def Person {:name s/Str
             :age s/Int}) ; We'll make s/Nat in a bit.
(def person (gen/hash-map :name gen/string
                          :age gen/nat))
(def prop-person-generates-Person
  (prop/for-all [p person]
                (s/validate Person p)))
(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`.

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

Names are Great, Two Are Better!

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

That’s a Lot of Missing Things!

Existing Generators: Bits and Bytes

`gen/boolean` Either `true` or `false`.

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

`gen/byte` A single `java.lang.Byte`.

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

`gen/bytes` A `byte-array`.

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

Existing Generators: Numbers

`gen/int` Any integer.

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

`gen/choose` Numbers in the specified range.

```
(gen/sample (gen/choose 18 45))
;; => (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` Postive-only or negative-only integers, allowing for 0.

`gen/s-pos-int` **and** `gen/s-neg-int` Postive-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.

```
(gen/sample gen/string-ascii)
;; => ("" "" "qc" "I-k" "F" ""
;;      ", 0u" "6kT]<" "}`!" "5ZH=v75")
```

`gen/string-alphanumeric` Alphanumeric strings.

Existing Generators: Collections

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

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

`gen/vector` A vector of generated things.

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

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

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

Existing Generators: Collections

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

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

Using `gen/elements`

Randomly pick (without exhaustion) from a collection.

```
(gen/sample
  (gen/elements [:spades :diamonds :hearts :clubs]))
;; => (:diamonds :clubs :spades :clubs :hearts
;;      :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!

```
(def even-and-positive (gen/fmap #(* 2 %) gen/pos-int))
(gen/sample even-and-positive 20)
;; => (0 0 2 0 8 6 4 12 4 18 10 0 8 2 16 16 6 4 10 4)
(def gen-double (gen/fmap rand gen/int))
(gen/sample gen-double)
;; => (0.0                0.0
;;     0.8433531349313175 -0.5407298249526976
;;     0.728215472484 2486 -0.5111220285736056
;;    -1.6998294599186186  1.474410436347970 4
;;     2.094621081981671  -1.7704991357273019)
```

Making New Generators With `gen/bind`

The `gen/bind` is sort of like `gen/fmap`: it takes in a generator and a function, but feeds the realized generated things into the function to make a new generator. It's basically for when you want a let block.

```
(gen/sample
  (gen/bind (gen/not-empty gen/string-ascii)
            #(gen/hash-map :str (gen/return %)
                           :key (gen/return (keyword %)))))
;; => ({:key :2,   :str "2"}
;;      {:key :+,   :str "+"}
;;      {:key :Nm,  :str "Nm"}
;;      {:key :Z/>, :str "Z/>"}
;;      ...)
```

Modifying Existing Generators

`gen/not-empty` Empty collections are sometimes irritating.

```
(gen/sample (gen/vector gen/int))
;; => ([] ; DO NOT WANT
;;      [-1] [0] [3 -3] [-1 0 2 0] [4]
;;      [5 -5] [2 1 -6 -1 2] [-1 -7 -7]
;;      [-5 6 -1 -4])
(gen/sample (gen/not-empty (gen/vector gen/int)))
;; => ([-1] [-2 2] [2 -2] [2] [2 0] [3 -6]
;;      [1 -5 0 4] [5 -2 -2 -3 -3 -4 5]
;;      [-2 7 2 -3] [7 6 -4])
```

`gen/no-shrink` I've never used this: it's weird? Shrinking is weird in general.

Modifying Existing Generators

`gen/such-that` Add a simple requirement to an existing generator, rejecting things that don't pass the predicate.

```
(gen/sample (gen/such-that #(< 3 %) gen/int))
;; => (4 4 9 6 4 5 6 6 4 9)
```

`gen/sized` Make a generator that is dependent on a *size* concept of some sort.

```
(gen/sample (gen/sized #(gen/choose 0 %)))
;; => (0 0 2 0 1 5 4 5 0 4)
```

`gen/resize` Change the size.

```
(gen/sample
  (gen/resize 5 (gen/sized #(gen/choose 0 %))))
;; => (5 5 5 3 1 4 1 1 5 4)
```

Making New Generators

`gen/return` Always the same thing.

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

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

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

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

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

Questions?