St. Louis Clojure

Clojure Schemata and Generators

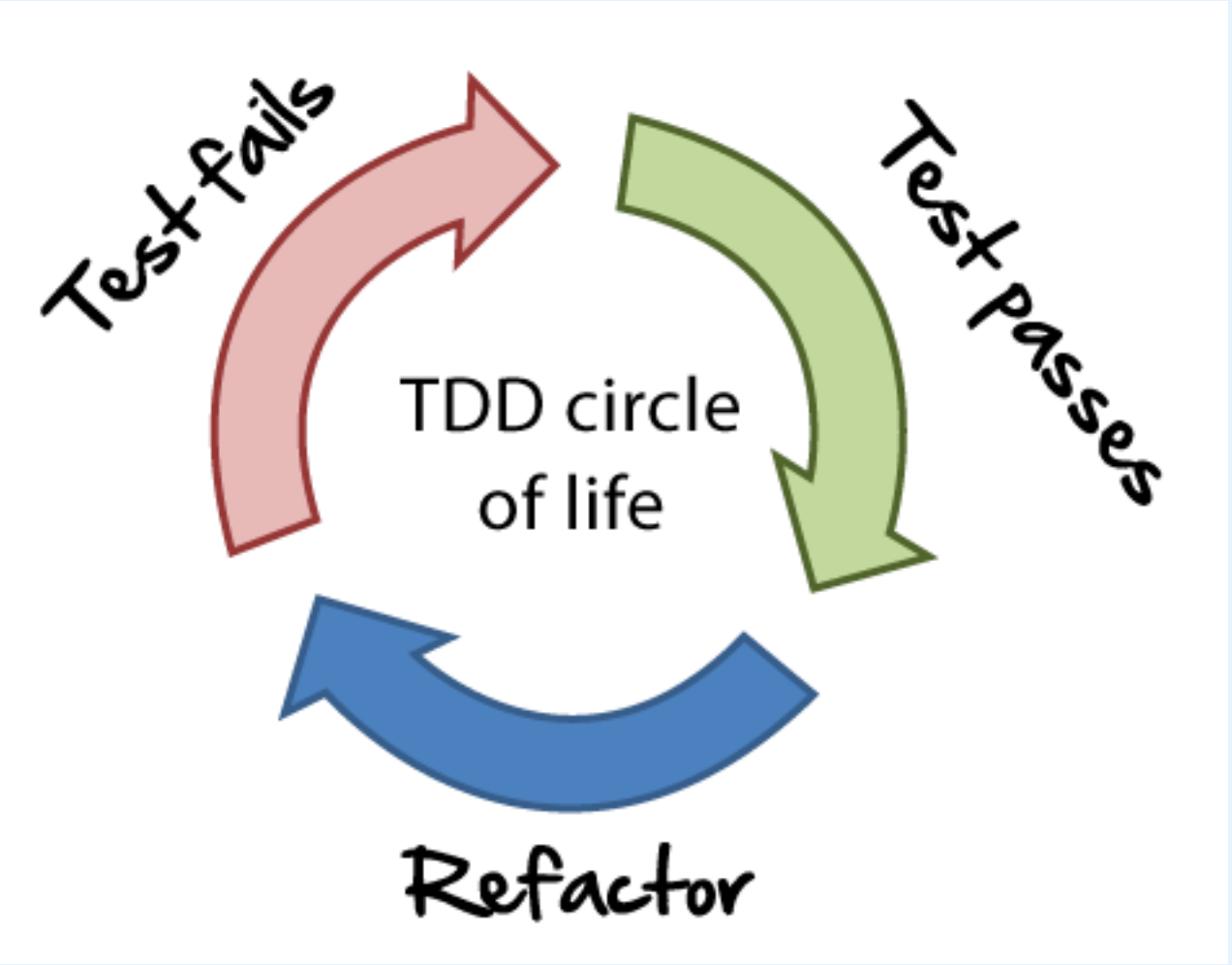
Christopher Mark Gore

cgore.com

Tuesday, September 29, AD 2015

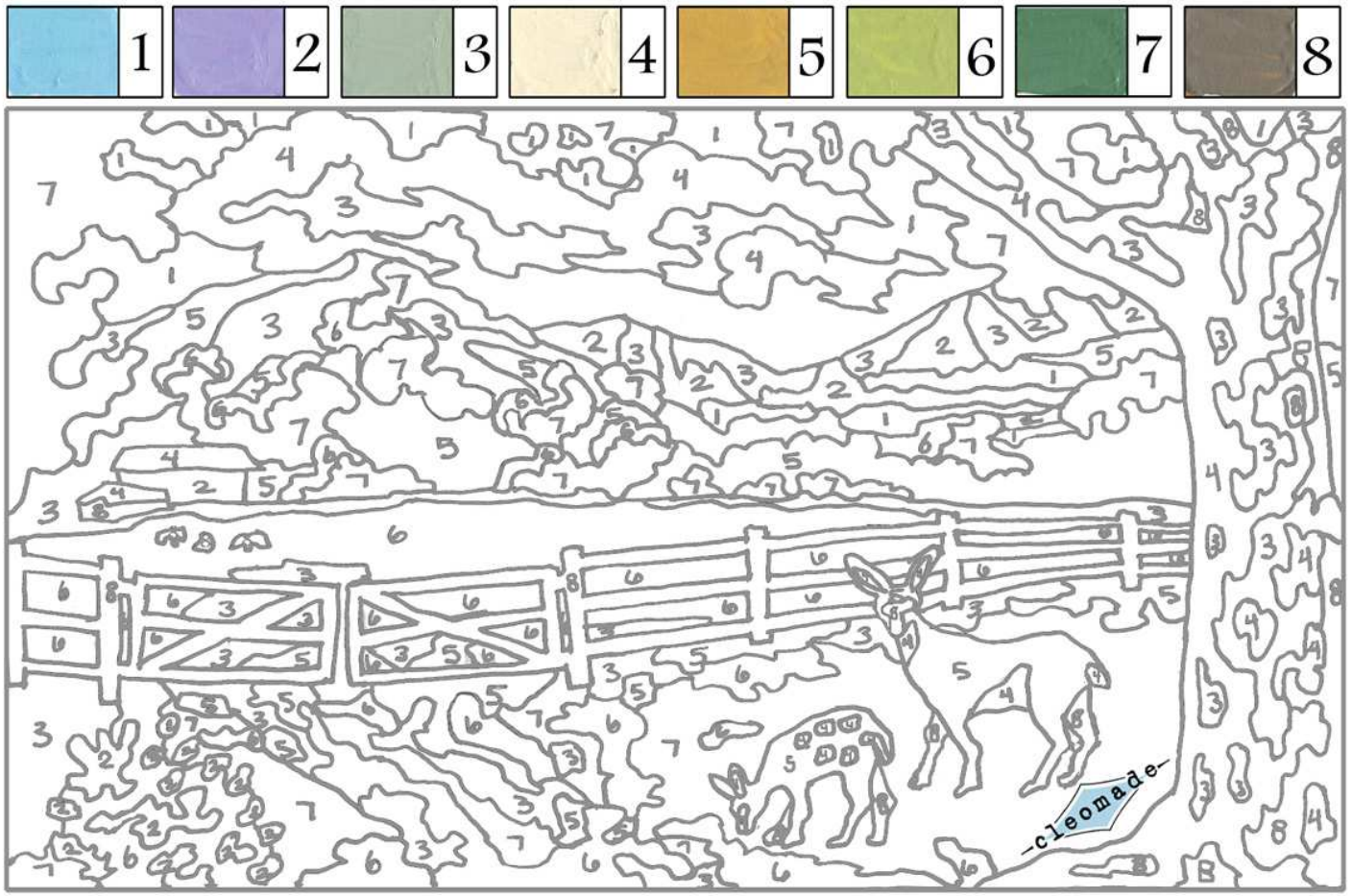
We write Clojure at The Climate Corporation, and we're hiring! Come work with us!

We don't need to look for new ways to test our code, TDD works great!



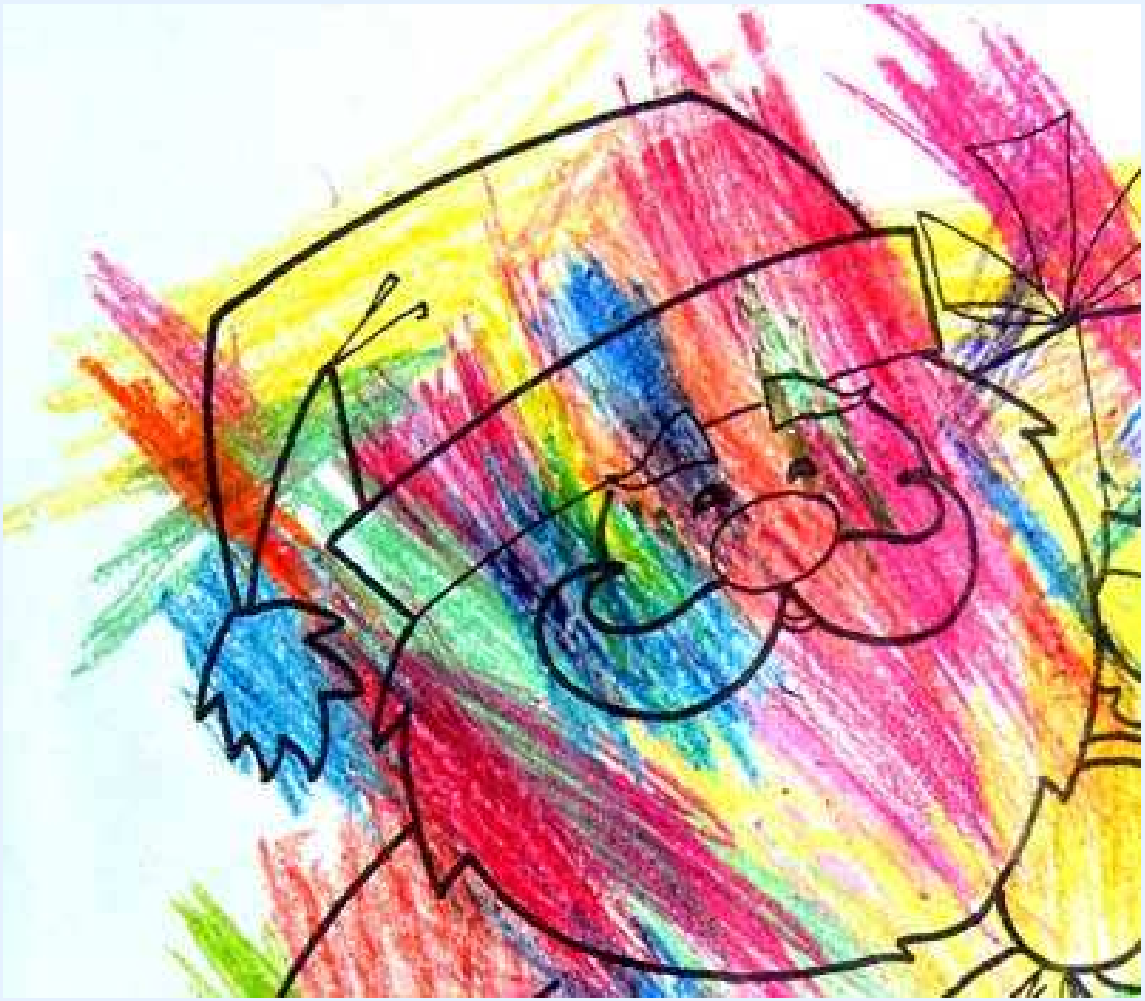
We don't need to look for new ways to test our code; *TDD works great???*

Schemata sketch out the boundary lines our code shouldn't cross.



Generators are nice little chaos monkeys for our code.

Used together, we can catch when our *"work of art"* might need some more time.



You'll need to add some stuff to your `project.clj` to get schemata and generators.

```
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   )
```

Schemata are like types, but only as strict as you want them to be at that specific moment.

```
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)
```

Generators make random examples according to your definition.

```
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)
```

Schema `validate` raises an error; schema `check` returns an error description.

```
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"))
```

We define properties we expect to always hold, and assert those properties over a large set of generated inputs.

```
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]}}
```

We discover the *properties really are* of our system, 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}
```

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

Our schema must accept *all* instances to be valid. Therefore 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)
```

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

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

Names are great!
Two for everything? That sucks.

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!

We can generate booleans with `gen/boolean`.

```
1  (gen/sample gen/boolean)
2  ;; => (false
3  ;;      true
4  ;;      false
5  ;;      true
6  ;;      true
7  ;;      false
8  ;;      false
9  ;;      false
10 ;;      false
11 ;;      true)
```

We can generate a single `java.lang.Byte` with `gen/byte`.

```
1  (gen/sample gen/byte)
2  ;; => (-88
3  ;;      101
4  ;;      101
5  ;;      104
6  ;;       24
7  ;;      -37
8  ;;      -36
9  ;;        9
10 ;;       20
11 ;;      107)
```

We can generate a `byte-array` with `gen/bytes`.

```
1 (gen/sample gen/bytes)
2 ;; => (#<byte[] [B@2efce23e>
3 ;;      ...)
```

There's a lot of existing generators for numbers of various sorts.

`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.

There's a lot of existing generators for 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" ""
;;     ", Ou" "6kT]<" "}`!" "5ZH=v75")
```

`gen/string-alphanumeric` Alphanumeric strings.

You can generate a vector that's in a specific order with `gen/tuple`.

```
(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"])
```

You can generate a vector of all the same sort of thing with `gen/vector`.

```
1  (gen/sample (gen/vector gen/int))
2  ;; => ([]
3  ;;      []
4  ;;      []
5  ;;      [1]
6  ;;      []
7  ;;      [0 5 -1 0 5]
8  ;;      [5 5 3]
9  ;;      [-6 3 -1]
10 ;;      [7 -7 -6 2 0 -5 3 -7]
11 ;;      [-9 -4 4 -6 -5 0])
```

We can randomly permute an existing sequence in various ways with `gen/shuffle`.

```
1  (gen/sample (gen/shuffle [1 2 3]))
2  ;; => ([1 2 3]
3  ;;      [3 1 2]
4  ;;      [1 2 3]
5  ;;      [2 3 1]
6  ;;      [1 3 2]
7  ;;      [2 1 3]
8  ;;      [2 1 3]
9  ;;      [2 1 3]
10 ;;      [1 3 2]
11 ;;      [3 1 2])
```

We can generate maps with both the keys and the values being randomly generated via gen/map.

```
1  (gen/sample
2    (gen/map (gen/elements [:bibbidi :bobbidi :boo])
3             gen/int))
4  ;; => ({}
5  ;;      {:bobbidi -1}
6  ;;      {}
7  ;;      {:bobbidi -2, :bibbidi -3}
8  ;;      {:boo 2, :bobbidi -2}
9  ;;      {:boo -2, :bobbidi -1}
10 ;;      {:boo -1, :bibbidi -5}
11 ;;      {}
12 ;;      {}
13 ;;      {:bibbidi 2})
```

You'll use `gen/hash-map` a lot, it generates a hash map with specific keys, like you might use for the input or result from a typical function.

```
(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}
;;     ...)
```

Use `gen/elements` to randomly pick (without exhaustion) from a collection.

```
1  (gen/sample
2    (gen/elements [:spades :diamonds :hearts :clubs]))
3  ;; => (:diamonds
4  ;;      :clubs
5  ;;      :spades
6  ;;      :clubs
7  ;;      :hearts
8  ;;      :spades
9  ;;      :spades
10 ;;      :hearts
11 ;;      :hearts
12 ;;      :hearts)
```

With `gen/fmap` you can use any Clojure function you want.

```
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
7  ;;      0.0
8  ;;      0.8433531349313175
9  ;;     -0.5407298249526976
10 ;;      0.7282154724842486
11 ;;     -0.5111220285736056
12 ;;     -1.6998294599186186
13 ;;      1.4744104363479704
14 ;;      2.094621081981671
15 ;;     -1.7704991357273019)
```

We also have `gen/bind`, which is sort of like `gen/fmap`, but for when you need a bound generated value.

```
(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|>"}
;;     ...)
```

Sometimes you really don't want generated empty values, so use `gen/not-empty` then.

```
(gen/sample (gen/vector gen/int))
;; => ([] ; <--- DO NOT WANT!  GO AWAY!
;;      [-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])
```

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?