Toggle slide-show mode
First slide
Previous slide
Next slide
Last slide
Open presenter preview

Mastermind

Let's begin

```
(defn exact-matches
  "Given two collections, return the number of
   positions where the collections contain equal
   items."
  [c1 c2])
```

Experiment by comparing data with diff

```
(require '[clojure.data :as data])
(data/diff [:r :g :g :b] [:r :y :y :b])

user=> [[nil :g :g] [nil :y :y] [:r nil nil :b]]
```

exact-matches

```
(defn exact-matches
  "Given two collections, return the number of
   positions where the collections contain equal
   items."
  [c1 c2]
  (let [[_ _ matches] (data/diff c1 c2)]
    (count (remove nil? matches))))

(exact-matches [:r :g :g :b] [:r :y :y :b])

user=> 2
```

Frequencies

```
(def example-secret [:r :g :g :b])
(def example-guess [:y :y :y :g])

(frequencies example-guess)
(frequencies example-secret)

user=> {:y 3, :g 1}
user=> {:r 1, :g 2, :b 1}

(select-keys (frequencies example-secret) example-guess)
(select-keys (frequencies example-guess) example-secret)
(merge-with min {:g 1} {:g 2})

user=> {:g 2}
user=> {:g 1}
user=> {:g 1}
```

unordered-matches

```
(defn unordered-matches
  "Given two collections, return a map where each
   key is an item in both collections, and each
   value is the minumum number of occurrences"
  [c1 c2]
  (let [f1 (select-keys (frequencies c1) c2)
        f2 (select-keys (frequencies c2) c1)]
    (merge-with min f1 f2)))

(unordered-matches [:r :g :g :b] [:y :y :y :g])

user=> {:g 1}
```

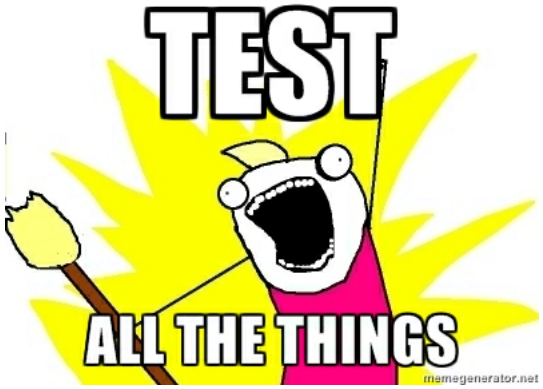
Combine to create score

```
(defn score [c1 c2]
  (let [exact (exact-matches c1 c2)
        unordered (apply +
                         (vals
                           (unordered-matches c1 c2)))]
    {:exact exact :unordered (- unordered exact)}))

(score [:r :g :g :b] [:r :y :y :g])

user=> {:exact 1, :unordered 1}
```

But WHERE ARE THE TESTS, you ask..



math.combinatorics

```
(require '[clojure.math.combinatorics :as comb])
(comb/selections [:r :g :b] 2)

user=> ((:r :r) (:r :g) (:r :b) (:g :r)
        (:g :g) (:g :b) (:b :r) (:b :g) (:b :b))
```

Generate all possible pairs of pairs

```
(-> (comb/selections [:r :g :b] 2)
    (comb/selections 2))

user=> (((:r :r) (:r :r)) ((:r :r) (:r :g))
        ((:r :r) (:r :b)) ((:r :r) (:g :r))
        ((:r :r) (:g :g)) ((:r :r) (:g :b))
        ...
        ((:b :b) (:b :g)) ((:b :b) (:b :b)))
```

And turn it into a function

```
(defn generate-turn-inputs
  "Generate all possible turn inputs for a
   clojurebreaker game with colors and n columns"
  [colors n]
  (-> (comb/selections colors n)
      (comb/selections 2)))
```

Define the domain

```
(defn score-inputs
  "Given a sequence of turn inputs, return a lazy
   sequence of maps with :secret, :guess, and
   :score."
  [inputs]
  (map
   (fn [[secret guess]]
     {:secret (seq secret)
      :guess (seq guess)
      :score (score secret guess)})
   inputs))
```

Take it for a ride

```
(->> (generate-turn-inputs [:r :g :b] 2)
     (score-inputs))

user=> ({:secret (:r :r), :guess (:r :r),
         :score {:exact 2, :unordered 0}}
        {:secret (:r :r), :guess (:r :g),
         :score {:exact 1, :unordered 0}}
        ....
        {:secret (:b :b), :guess (:b :b),
         :score {:exact 2, :unordered 0}})
```

Print out the entire domain

```
(use 'clojure.pprint)
(require '[clojure.java.io :as io])
(with-open [w (io/writer "scoring-table")]
  (binding [*out* w]
    (print-table
     (->> (generate-turn-inputs [:r :g :b :y] 4)
          (score-inputs)))))
```

You want me to do what?

```
=================================================
:secret       | :guess        | :score
=================================================
(:r :r :r :r) | (:r :r :r :r) | {:exact 4, :unordered
(:r :r :r :r) | (:r :r :r :g) | {:exact 3, :unordered
(:r :r :r :r) | (:r :r :r :b) | {:exact 3, :unordered
(:r :r :r :r) | (:r :r :r :y) | {:exact 3, :unordered
....
(:y :y :y :y) | (:y :y :y :b) | {:exact 3, :unordered
(:y :y :y :y) | (:y :y :y :y) | {:exact 4, :unordered
=================================================
```

Experiment with test.generative

```
(require '[clojure.test.generative.generators :as gen])
(gen/vec gen/boolean)
(gen/hash-map gen/byte gen/int)

user=> [false true false false true true false false
        true true false false true false true false fals
user=> {-65 -1280904780, 96 193928749, 4 835628727,
        -69 1265123962, 38 740672507, 42 -872871511}

(gen/geometric 0.02)
(gen/list gen/int 2)
(gen/list gen/int (gen/uniform 0 5))

user=> 42
user=> (278318889 -909716267)
user=> (-1880284059 -442888494 -98097322 1681870739)
```

Introduce randomness

```
(defn random-secret []
  (gen/vec #(gen/one-of :r :g :b :y) 4))

(random-secret)

user=> [:b :y :b :r]
```

Define the system constraints/contracts

```
(defn matches
  [score]
  (+ (:exact score) (:unordered score)))

(defn scoring-is-symmetric
  [secret guess sc]
  (= sc (score guess secret)))

(defn scoring-is-bounded-by-number-of-pegs
  [secret guess score]
  (<= 0 (matches score) (count secret)))

(defn reordering-the-guess-does-not-change-matches
  [secret guess sc]
  (= #{(matches sc)}
     (into #{}
           (map
            #(matches (score secret %))
            (comb/permutations guess)))))
```

Try out our contracts with sample data

```
(def secret [:r :g :g :b])
(def guess [:r :b :b :y])

(scoring-is-symmetric secret guess (score secret guess))

(scoring-is-bounded-by-number-of-pegs
 secret guess (score secret guess))

(reordering-the-guess-does-not-change-matches
  secret guess (score secret guess))


user=> true
user=> true
user=> true
```

Create a test.generative test

```
(use '[clojure.test.generative :only (defspec) :as test])

(defspec score-invariants
  score
  [^{:tag `random-secret} secret
   ^{:tag `random-secret} guess]
  (assert (scoring-is-symmetric secret guess %))
  (assert (scoring-is-bounded-by-number-of-pegs secret gu
  (assert (reordering-the-guess-does-not-change-matches s
```

Run the test

```
(test/test-vars #'user/score-invariants)

user=> {:iterations 1747, :msec 10004,
        :var #'user/score-invariants, :seed 42}
       {:iterations 1748, :msec 10002,
        :var #'user/score-invariants, :seed 46}
       {:iterations 1733, :msec 10002,
        :var #'user/score-invariants, :seed 43}
       {:iterations 1745, :msec 10001,
        :var #'user/score-invariants, :seed 49}
       {:iterations 1734, :msec 10004,
        :var #'user/score-invariants, :seed 45}
       {:iterations 1762, :msec 10009,
        :var #'user/score-invariants, :seed 44}
       {:iterations 1743, :msec 10008,
        :var #'user/score-invariants, :seed 47}
       {:iterations 1749, :msec 10009,
        :var #'user/score-invariants, :seed 48}
       :run-complete
```

What happens when a test fails?

```
(defn scoring-is-bounded-by-number-of-pegs
  [secret guess score]
  (>= 0 (matches score) (count secret)))

(test/test-vars #'user/score-invariants)

user=> {:form (#'user/score-invariants [:g :g :g :r]
                                       [:r :r :r :y]),
 :iteration 0, :seed 44,
 :error "Assert failed: (scoring-is-bounded-by-number-of-
                           secret guess %)",
 :exception #<AssertionError java.lang.AssertionError:
   Assert failed: (scoring-is-bounded-by-number-of-pegs
                     secret guess %)>}
```

Paste :form in to the REPL to examine your problem

```
(#'user/score-invariants [:g :y :b :r] [:r :r :y :y])

user => AssertionError Assert failed:
  (scoring-is-bounded-by-number-of-pegs
    secret guess %)
  user/score-invariants (NO_SOURCE_FILE:145)
```

Practical cases

```
(defspec integer-commutative-laws
  (partial map identity)
  [^{:tag `integer} a ^{:tag `integer} b]
  (if (longable? (+' a b))
    (assert (= (+ a b) (+ b a)
               (+' a b) (+' b a)
               (unchecked-add a b) (unchecked-add b a)))
    (assert (= (+' a b) (+' b a))))
  (if (longable? (*' a b))
    (assert (= (* a b) (* b a)
               (*' a b) (*' b a)
               (unchecked-multiply a b) (unchecked-multi
    (assert (= (*' a b) (*' b a)))))
```

Another

```
(defspec integer-associative-laws
  (partial map identity)
  [^{:tag `integer} a ^{:tag `integer} b ^{:tag `integer}
  (if (every? longable? [(+' a b) (+' b c) (+' a b c)])
    (assert (= (+ (+ a b) c) (+ a (+ b c))
               (+' (+' a b) c) (+' a (+' b c))
               (unchecked-add (unchecked-add a b) c)
               (unchecked-add a (unchecked-add b c))))
    (assert (= (+' (+' a b) c) (+' a (+' b c))
               (+ (+ (bigint a) b) c) (+ a (+ (bigint b)
  (if (every? longable? [(*' a b) (*' b c) (*' a b c)])
    (assert (= (* (* a b) c) (* a (* b c))
               (*' (*' a b) c) (*' a (*' b c))
               (unchecked-multiply (unchecked-multiply a
               (unchecked-multiply a (unchecked-multiply
    (assert (= (*' (*' a b) c) (*' a (*' b c))
               (* (* (bigint a) b) c) (* a (* (bigint b)
```

And of course

```
(defspec integer-distributive-laws
  (partial map identity)
  [^{:tag `integer} a ^{:tag `integer} b ^{:tag `integer]
  (if (every? longable? [(*' a (+' b c)) (+' (*' a b) (*'
                         (*' a b) (*' a c) (+' b c)])
    (assert (= (* a (+ b c)) (+ (* a b) (* a c))
               (*' a (+' b c)) (+' (*' a b) (*' a c))
               (unchecked-multiply a (+' b c))
               (+' (unchecked-multiply a b) (unchecked-mu
    (assert (= (*' a (+' b c)) (+' (*' a b) (*' a c))
               (* a (+ (bigint b) c)) (+ (* (bigint a) b]
```

References

- This talk github.com/abedra/the-generative-generation
- Mastermind en.wikipedia.org/wiki/Mastermind_(board_game)
- Test Generative github.com/clojure/test.generative
- Programming Clojure, 2nd Edition pragprog.com/book/shcloj2/programming-clojure
- Haskell's Quick Check www.haskell.org/haskellwiki/Introduction_to_QuickCheck
- John Hughes on Quick Check (Erlang) www.erlang.org/euc/03/proceedings/1430John.pdf
- ICheck (Ioke) github.com/olabini/ioke/blob/master/lib/ioke/icheck.ik
- Org HTML Slideshow (ClojureScript) github.com/relevance/org-html-slideshow

Questions?