Property Based Testing:
Let the Computer Make Tests for You

Peter Schuck
@spinningtopsofdoom / @bendyworks

Unit Testing pain points

- Linear growth with code size
- Snowflake test inputs
- Zombie tests

Test Check

Property Test Library for Clojure

https://github.com/clojure/test.check

Source code for all examples is at

https://github.com/spinningtopsofdoom/property-test-madison-clojure

Overview from spaaaaace of Property based testing

Typical Unit Tests

```
(= (+ 3 7) 10)
(= (sort ["c" "a" "b"]) ["a" "b" "c"])
```

Property Based Tests

```
(= (+ a b) (+ b a))
(= (sort items) (sort (sort items)))
```

Properties are tested via generated inputs

```
(gen/vector gen/int 2)
(gen/vector gen/string)
```

Let’s go through a strawman example

Latinify function

Transforms the word "cat" to "Felinus"

```
(= (latinify "cat") "Felinus")
```

Basic unit tests for latinify

```
(= (latinify "no felines here") "no felines here")
(= (latinify "cat") "Felinus")
(= (latinify "Where is the cat?") "Where is the Felinus?")
```

Totally valid function

```
(def latinify [plain-string]
  (condp #(= plain-string %)
   "cat" "Felinus"
   "Where is the cat?" "Where is the Felinus?"
   plain-string)
```

Lets add more tests

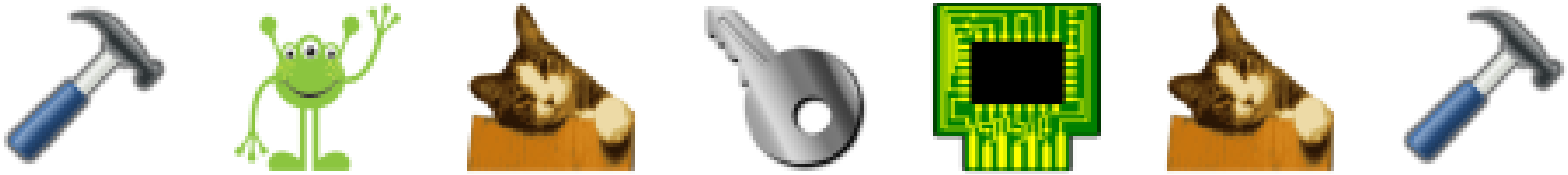
```
(= (latinify "cat with a hat") "Felinus with a hat")
(= (latinify "fat cat") "fat Felinus")
```

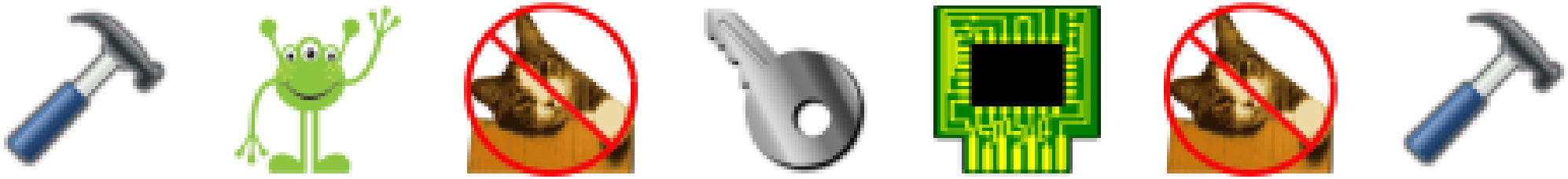
Still a valid function

```
(def latinify [plain-string]
  (condp #(= plain-string %)
   "cat" "Felinus"
   "Where is the cat?" "Where is the Felinus?"
   "fat cat" "fat Felinus"
   "cat with a hat" "Felinus with a hat"
   plain-string)
```

Now lets try property based testing

The output string should contain no "cat"s

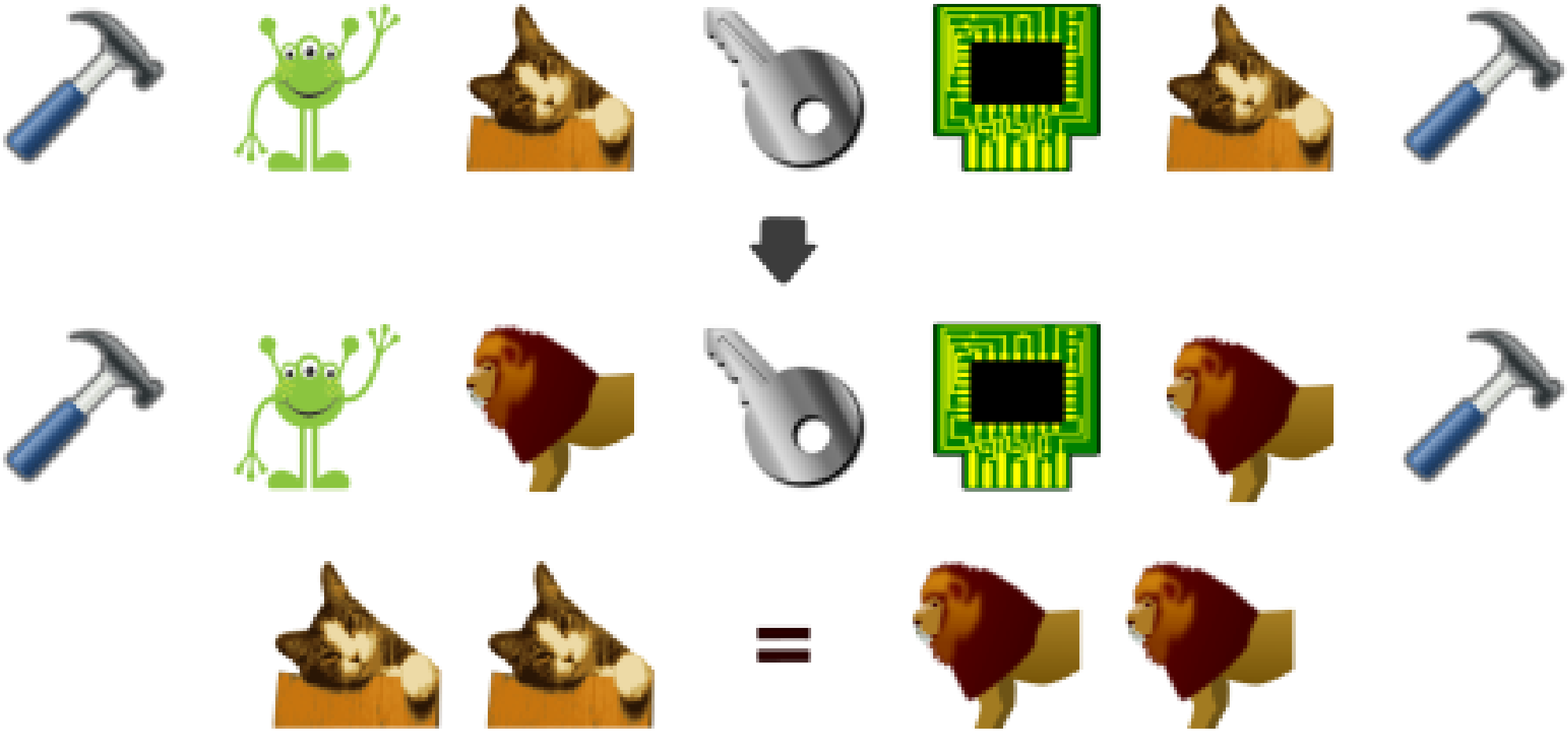




Our function is even easier to fake

```
(def latinify [plain-string]
  "No Felinus here :p")
```

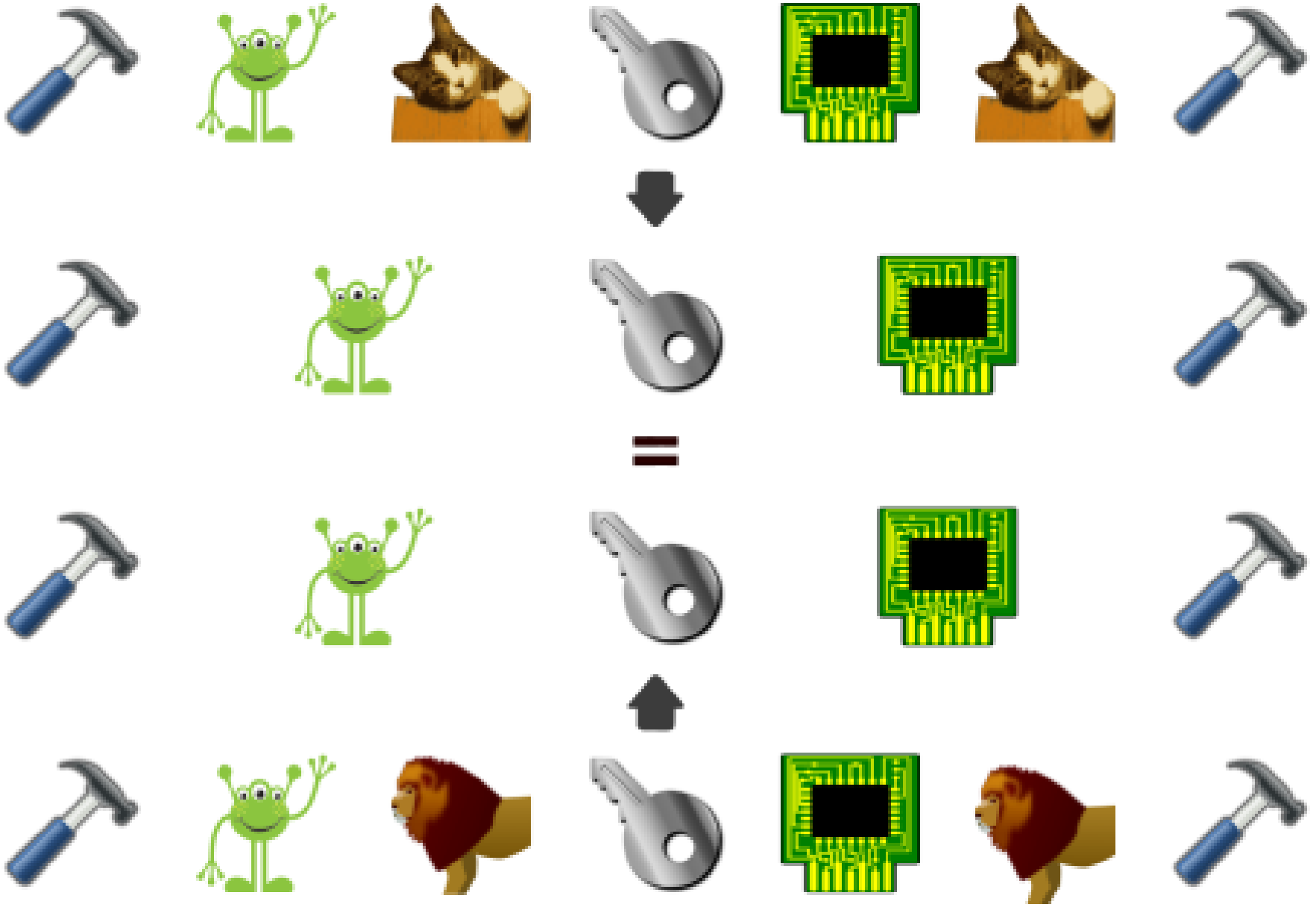
The output string has as many "Felinus" as the input has "cat"s



Closer but still wrong

```
(defn latinify [plain]
  (string/replace plain #"\bcat\b" "Felinus behind us"))
```

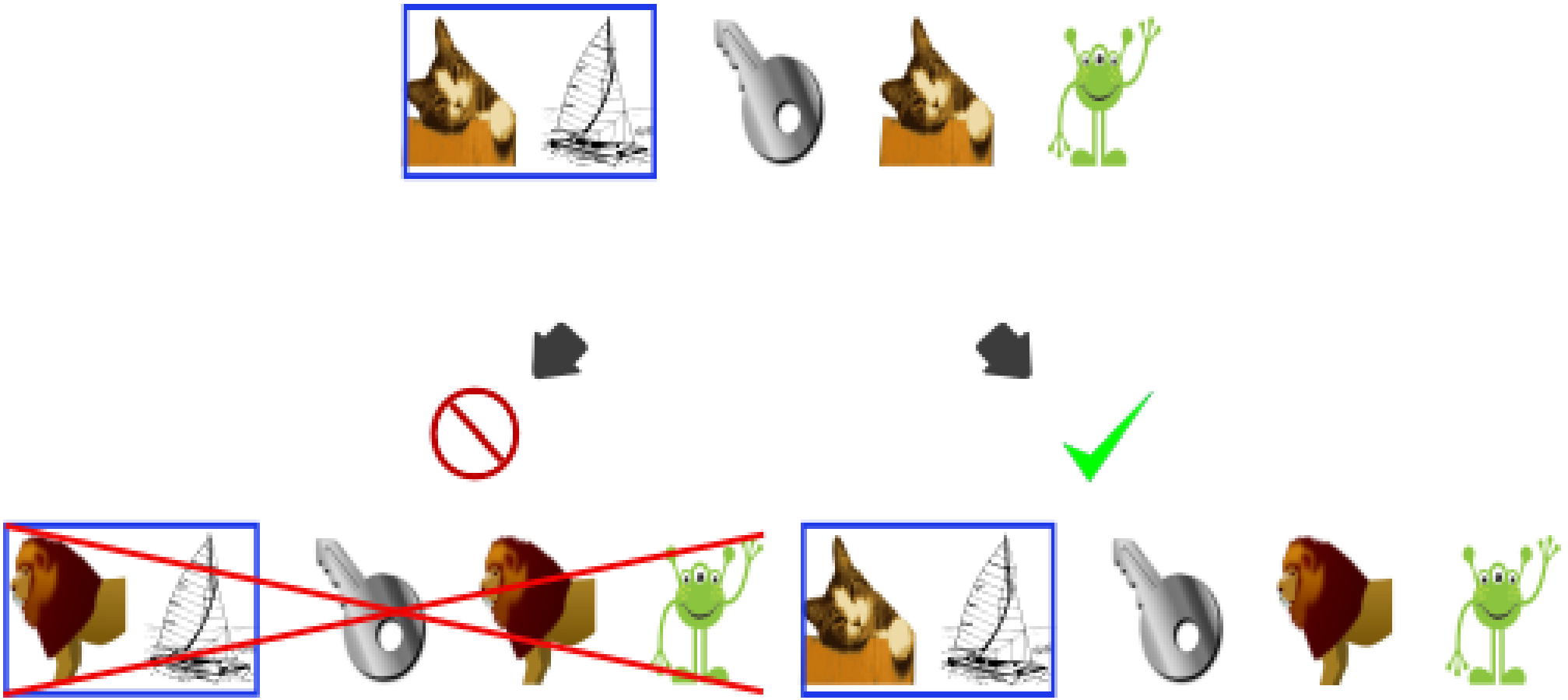
Removing 'Felinus' and 'cat' should get the same string



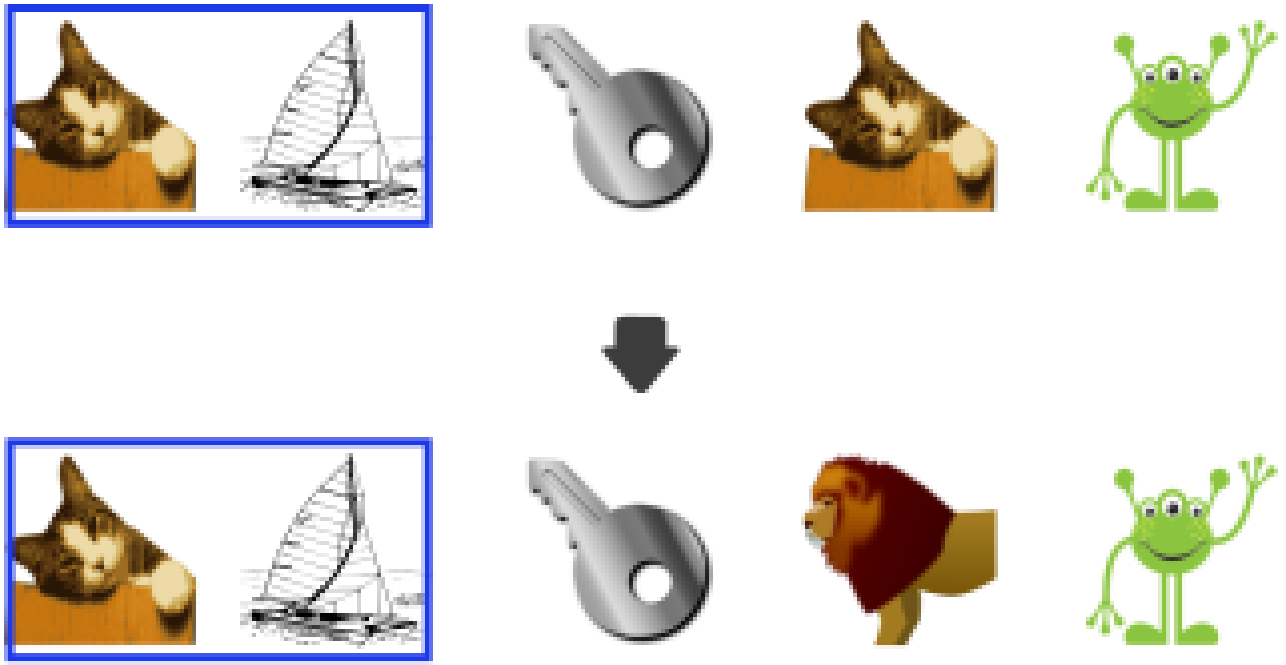
Finally forced to create a reasonable function

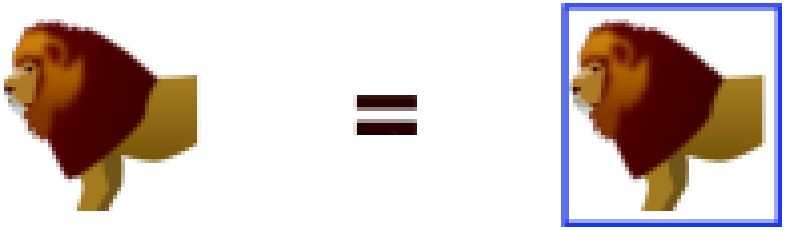
```
(defn latinify [plain]
  (string/replace plain #"cat" "Felinus"))
```

What about "Hepcat on a catamaran"?



Number of "Felinus" words matches the number of "Felinus" strings





Final function only replaces "cat" when it is a word

```
(defn latinify [plain]
  (string/replace plain #"\bcat\b" "Felinus"))
```

Finding Properties to test

1) Write some unit tests
2) Construct bad function / method that passes unit tests
3) Find property to fail function / method
4) Refactor bad function / method to slip by property test
5) Repeat 3 – 4 until satisfied with covage

test.check Generators

Features and examples

Generators provide random inputs for tests

- Composable, you can build a bigger generator from smaller ones
- Repeatable, a failing input can be repeated
- Shrinkable, The killer feature of property test libraries

What is shrinking?

Find the misspelled word

Deductibles are typically used to deter the large number of claims that a consumer can be reasonibly expected to bear the cost of. By restricting its coverage to events that are significant enough to incur large costs, the insurance firm expects to pay out slightly smaller amounts much less frequently, incurring much higher savings.

Deductibles are typically used to deter the large number of claims that a consumer can be reasonibly expected to bear the cost of.

that a consumer can be reasonibly expected to bear the cost of.

that a consumer can be reasonibly

Shrinking takes a failing input and finds the smallest portion of it that fails the test

reasonibly

Eris composite generator example

Collection of 10 temperature readings

Each reading looks like

```
{:scale 'F' :degrees 50}
```

Basic Setup

```
(def scale (gen/elements [:C :F :K]))
(def degrees (gen/choose 0 100))
(def reading (gen/hash-map :scale scale :degrees degrees))
(def measurements (gen/vector reading 10))
```

More realistic scale distribution

60% F - 30% C - 10% K

frequency: Changes the probability that generator will be chosen from an array

```clojure
(def fahrenhiet (gen/return :F))
(def celsius (gen/return :C))
(def kelvin (gen/return :K))
(def dist-scale
  (gen/frequency [[6 fahrenhiet] [3 celsius] [1 kelvin]]))
(def dist-reading
  (gen/hash-map :scale dist-scale :degrees degrees))
(def dist-measurements (gen/vector dist-reading 10))
```

Temperatures should match scale

100 C or 0 K = :(

bind: Generator => Output => Function => Generator

```clojure
(def scale-degrees
  {:F (gen/choose 0 100)
   :C (gen/choose 0 32)
   :K (gen/choose 280 310)})
(def realistic-reading
  (gen/bind dist-scale
    #(gen/hash-map
       :scale (gen/return %)
       :degrees (% scale-degrees))))
```

More accurate readings

85 degrees => 85.94 degrees

fmap: Generator => Output => Function => Modified Output

```clojure
(def accurate-reading
  (gen/fmap
    (fn [[precision reading]]
      (update reading :degrees #(+ % precision)))
    (gen/tuple
      (gen/fmap
        #(/ % 100.0)
        (gen/choose 0 100))
      realistic-reading)))
(def accurate-measurements (gen/vector accurate-reading 10))
```

Have at least one Kelvin Reading

[C, C, C, C, C, F, F, F, F, F] ☒

[C, C, C, C, C, K, F, F, F, F] ☑

such-that: Generator => Invalid Output => Check => Invalid Output => Check => Valid Output

```clojure
(def one-k-measurements
  (gen/such-that
    (fn [measurements] (some #(= :K (% :scale)) measurements))
    accurate-measurements))
```

Generators

- Force mapping out problem domain
- No more copy pasta of hand rolled inputs
- Easily create sample data for new functionality

So far we've tested functions and created generators.

How can you property test state changes?

Friendship tracker with methods

- `(add-friendship! "Bob" "Alice")`
  - Adds a new friendship between 'Bob' and 'Alice'
- `(remove-friendship! "Bob" "Alice")`
  - Removes the friendship between 'Bob' and 'Alice'
- `(friends "Alice" "Bob")`
  - Checks if 'Alice' and 'Bob' have a friendship
- `(getFriends "Alice")`
  - Get all of 'Alice's friends
- `(getPeople)`
  - Gets the people in our database ('Alice' and 'Bob')

Typical Usage

```
(def friends create-friendship-db)
(add-friendship! "Alice" "Bob")
(add-friendship! "Alice" "Carol")
(add-friendship! "Dan" "Bob")
(remove-friendship! "Alice" "Bob")
```

We can represent this in data

```clojure
(def operations
  [[:add-friendship "Alice" "Bob"]
   [:add-friendship "Alice" "Carol"]
   [:add-friendship "Dan" "Bob"]
   [:remove-friendship "Alice" "Bob"]])
```

Similar to a todo list

The actions and information on the list are not the actual actions

Todos

- Pick up milk on the way home
- Pet the nice cat on daily constitutional
- Prove that P ≠ NP

No self friending

Symmetrical Friendship

⇐

⇒

Friend friendship operation generator

```
(def operation
  (gen/tuple
    (gen/elements [:add-friendship :remove-friendship])
    (gen/elements people)
    (gen/elements people)))

(def operations (gen/vector operation))
```

Create the Friend Tracker with operations that modified it's state

```
(defn apply-operations [operations]
  (let [db (create-friendship-db)]
    (doseq [op operations]
      (case (first op)
         :add-friendship
         (apply add-friendship! (cons db (rest op)))
         :remove-friendship
          (apply remove-friendship! (cons db (rest op)))))
    db))
```

Fresh Object Generator

+

Array of operations to perform

=>

Modified Object with trace of how it got to it’s current state

Property Testing eases testing pain points

- Cut down on code size and churn
- Compliments Unit tests
- Generators have multiple uses outside of tests

Fin

Questions?