Generative Testing

@stuarthalloway

Testing is Hard

Difficult to interpret tests as knowledge about system

Difficult to achieve good test coverage

Trivial and serial vs. complex and parallel

Scaling is worse-than-linear in *human* effort

Generative Testing

Programmer models the domain

A *program* generates the individual tests

Validate categoric properties, not specific outcomes

Key Use Case

Generative testing should be the default approach for automated testing.

Key Operations

Generator generates inputs

Runner connects inputs / code / properties

Validator assesses inputs, outputs

Shrinker finds tractable failure cases

State machine models time and concurrency

Don’t Use For

Design

Code Exploration (use a REPL)

Examples (prefer example-based tests)

Integration (use simulation)

Load

Singletons

Resources

env	resource
Wikipedia	http://en.wikipedia.org/wiki/QuickCheck
Haskell	QuickCheck
Clojure	data.generators test.check test.generative
Java	QuickCheck junit-quickcheck
Scala	ScalaCheck

Genesis

Datomic

Reading the Code

Extensible Data Notation (edn)

Rich set of built in data types

Generic extensibility

Language neutral

Represents values (not identities, objects)

type	example	java equivalent
string	`"foo"`	String
character	`\f`	Character
integer	`42`	Long
a.p. integer	`42N`	BigInteger
double	`3.14159`	Double
a.p. double	`3.14159M`	BigDecimal
boolean	`true`	Boolean
nil	`nil`	`null`
symbol	`foo, +`	N/A
keyword	`:foo, ::foo`	N/A

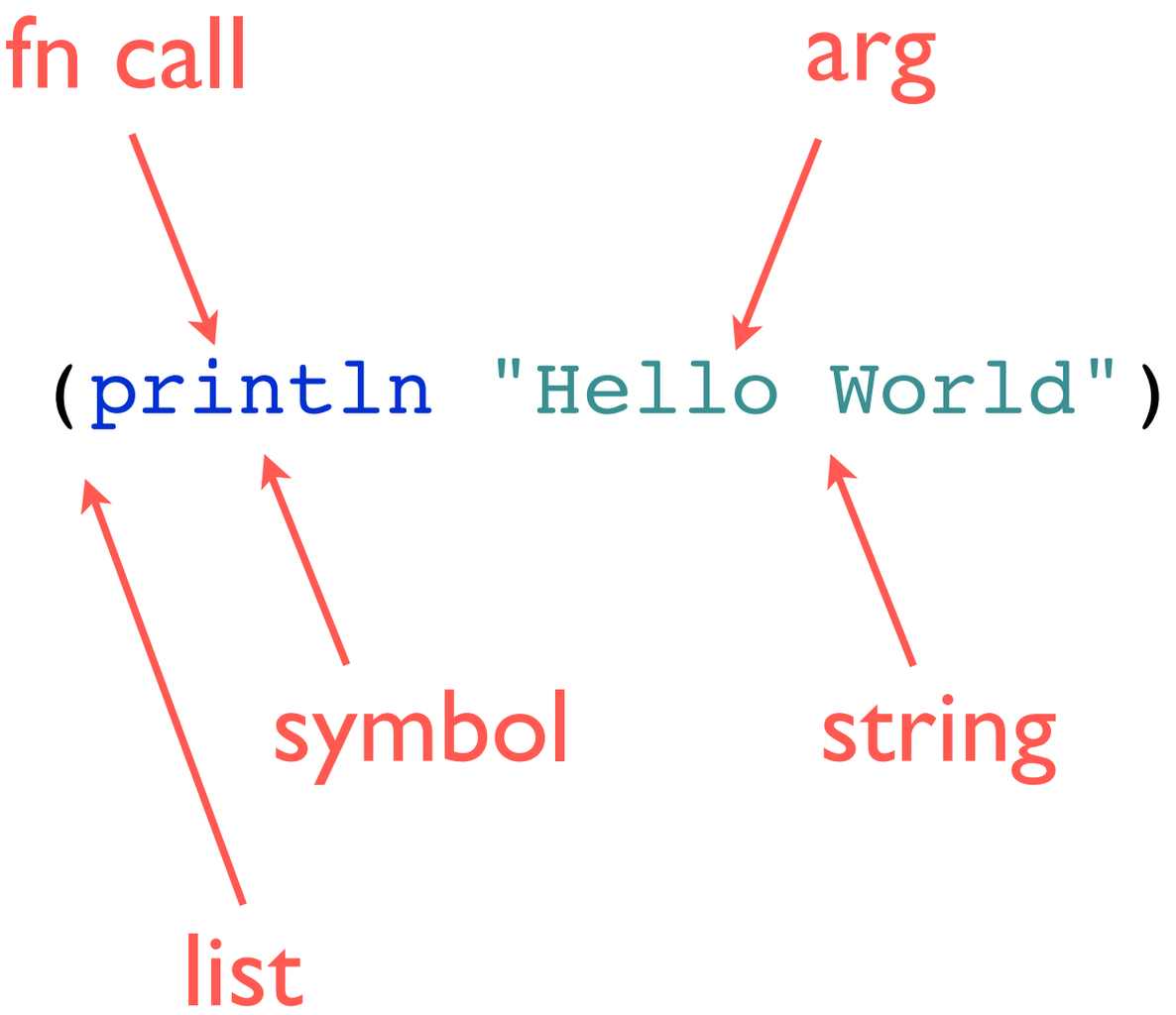
type	properties	example
list	singly-linked, insert at front	`(1 2 3)`
vector	indexed, insert at rear	`[1 2 3]`
map	key/value	`{:a 100` `:b 90}`
set	key	`#{:a :b}`

Clojure programs are written in data, not text

Function Call

semantics:

structure:



Function Definition

define a fn

fn name

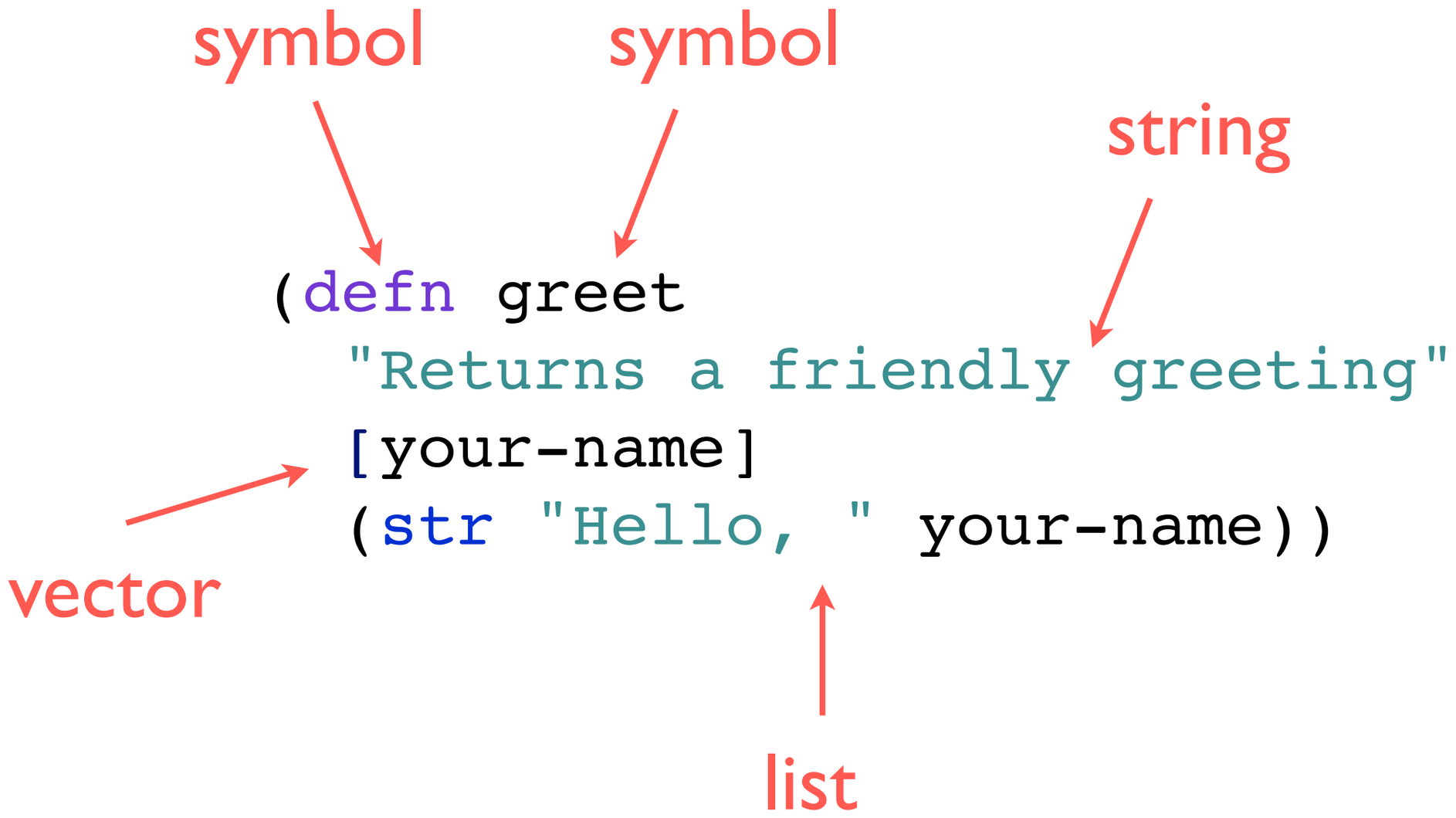
docstring

```
(defn greet
  "Returns a friendly greeting"
  [your-name]
  (str "Hello, " your-name))
```

arguments

fn body

Still Just Data



Metadata

Orthogonal to logical value of data

Available as map associated with symbol or collection

Does not impact equality or in any way intrude on value

Reader support

Not part of edn

Metadata in the Reader

metadata on [1 2 3]]

```
^{:a 1 :b 2} [1 2 3]
```

```
^String x
```

sugar for

```
^{:tag String} x
```

data.generators

Objectives

Generate all kinds of data

Various distributions

Predictable

Approach

Generator fns shadow related fns in clojure.core

Default integer distributions are uniform on range

Other defaults are arbitrary

Repeatable via dynamic binding of *rnd*

Scalar Generators

```
(require '[clojure.data.generators :as gen])

(gen/short)
=> 14913
```

```
(gen/uniform 0 10)
=> 6

(gen/rand-nth [:a :b :c])
=> :a
```

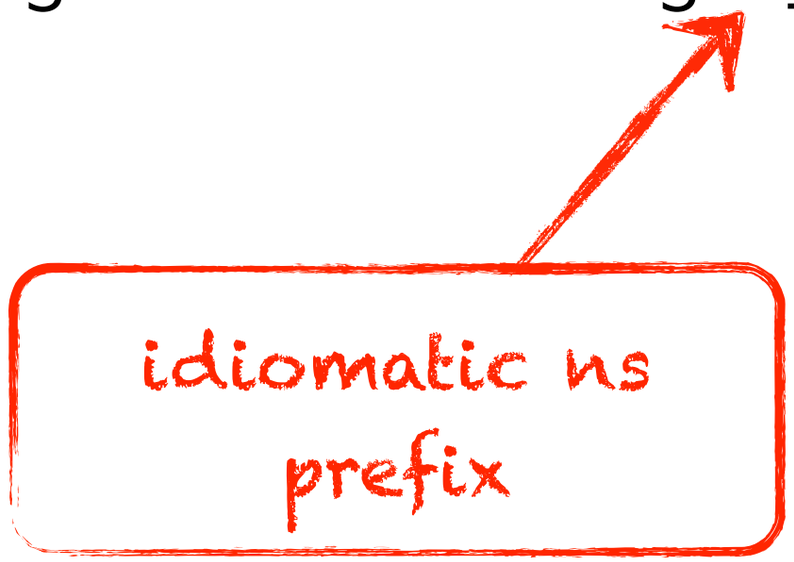
Scalar Generators

```
(require '[clojure.data.generators :as gen])

(gen/short)
=> 14913

(gen/uniform 0 10)
=> 6

(gen/rand-nth [:a :b :c])
=> :a
```



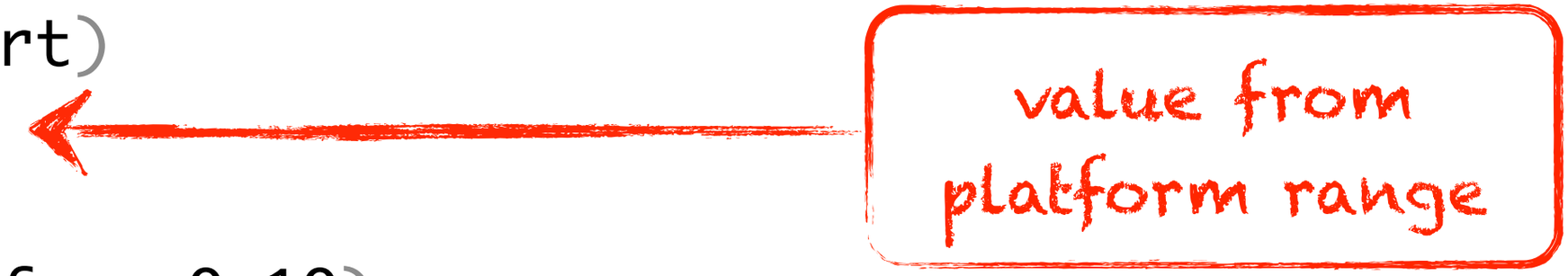
Scalar Generators

```
(require '[clojure.data.generators :as gen])

(gen/short)
=> 14913

(gen/uniform 0 10)
=> 6

(gen/rand-nth [:a :b :c])
=> :a
```



Scalar Generators

```
(require '[clojure.data.generators :as gen])

(gen/short)
=> 14913

(gen/uniform 0 10)
=> 6

(gen/rand-nth [:a :b :c])
=> :a
```

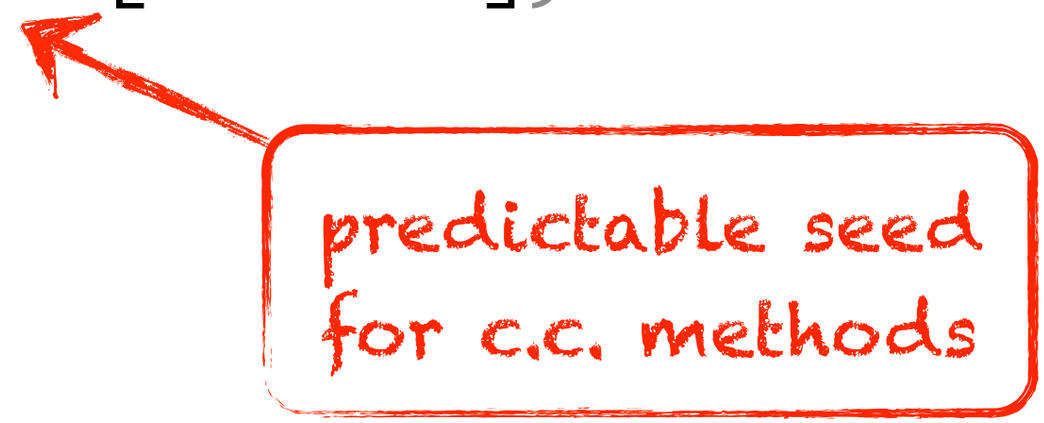
Scalar Generators

```
(require '[clojure.data.generators :as gen])

(gen/short)
=> 14913

(gen/uniform 0 10)
=> 6

(gen/rand-nth [:a :b :c])
=> :a
```

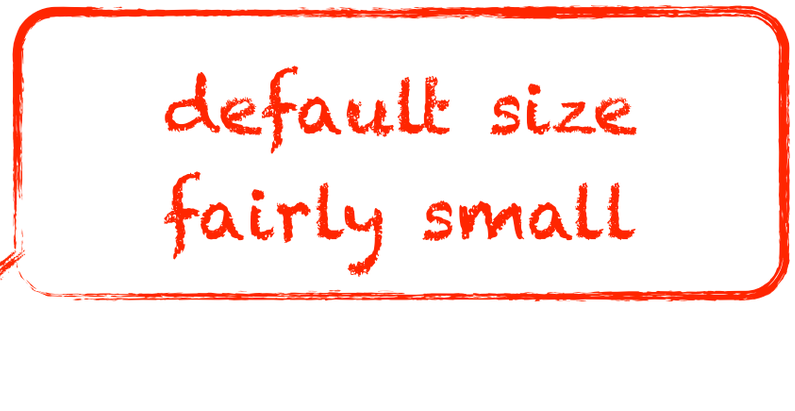


Collection Generators

```
(gen/list gen/short)
=> (-8600 -14697 -2382 18540 27481)
```

```
(gen/hash-map gen/short gen/string 2)
=> {-7110 "UBL)l",
    11472 "Q5|>^>rQNL9E..y#}IMpw>gnM']jD'<q"}
```

Collection Generators




```
(gen/list gen/short)
=> (-8600 -14697 -2382 18540 27481)
```

```
(gen/hash-map gen/short gen/string 2)
=> {-7110 "UBL)l",
    11472 "Q5|>^>rQNL9E..y#}IMpw>gnM']jD'<q"}
```

Collection Generators

```
(gen/list gen/short)
=> (-8600 -14697 -2382 18540 27481)
```

```
(gen/hash-map gen/short gen/string 2)
=> {-7110 "UBL)l",
    11472 "Q5|>^>rQNL9E..y#}IMpw>gnM']jD'<q"}
```

Composition

```
(gen/one-of gen/long gen/keyword)
=> :OBe0Mkc1g7eqqQnGvcXq0m-McRzl9areH0NwR1

(gen/weighted {gen/long 10 gen/keyword 1})
=> 47180317273564660 9
```

```
(gen/scalar)
=> -49

(gen/collection)
=> #{-3945240682015942560
     -490949758534279262 0
     ...}
```

Composition

```
(gen/one-of gen/long gen/keyword)
=> :OBe0MkdIgZeqqQnGvcXq0m-McRzl9areH0NwR1

(gen/weighted {gen/long 10 gen/keyword 1})
=> 47180317273564660 9

(gen/scalar)
=> -49

(gen/collection)
=> #{-3945240682015942560
     -490949758534279262 0
     ...}
```

choose
(equal weights)

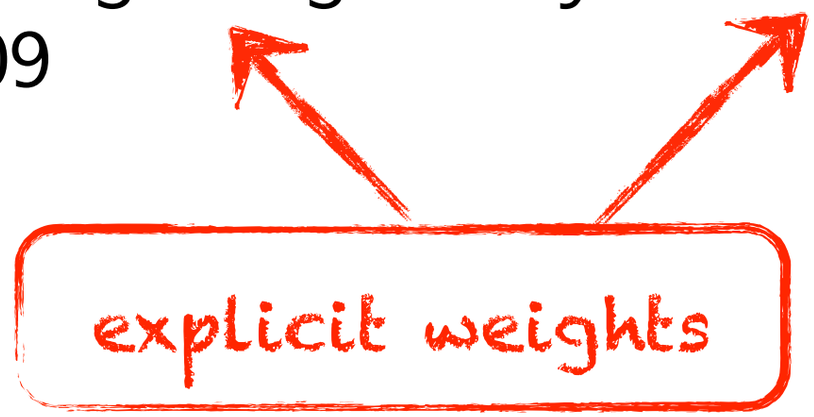
Composition

```
(gen/one-of gen/long gen/keyword)
=> :OBe0Mkc1g7eqqQnGvcXq0m-McRzl9areH0NwR1

(gen/weighted {gen/long 10 gen/keyword 1})
=> 47180317273564660 9

(gen/scalar)
=> -49

(gen/collection)
=> #{-3945240682015942560
     -490949758534279262 0
     ...}
```



Composition

```
(gen/one-of gen/long gen/keyword)
=> :OBe0Mkc1g7eqqQnGvcXq0m-McRzl9areH0NwR1

(gen/weighted {gen/long 10 gen/keyword 1})
=> 47180317273564660 9

(gen/scalar)
=> -49

(gen/collection)
=> #{-3945240682015942560
     -490949758534279262 0
     ...}
```

Composition

```
(gen/one-of gen/long gen/keyword)
=> :OBe0Mkc1g7eqqQnGvcXq0m-McRzl9areH0NwR1

(gen/weighted {gen/long 10 gen/keyword 1})
=> 47180317273564660 9
```

```
(gen/scalar)
=> -49

(gen/collection)
=> #{-3945240682015942560
     -490949758534279262 0
     ...}
```

test.generative

```
(defspec longs-are-closed-under-increment
  inc
  [^long l]
  (assert (instance? Long %)))
```

spec fn name

```
(defspec longs-are-closed-under-increment
  inc
  [^long l]
  (assert (instance? Long %)))
```

"type" resolves to gen/long

```
(defspec longs-are-closed-under-increment
  inc
  [^long l]
  (assert (instance? Long %)))
```

fn under test

```
(defspec longs-are-closed-under-increment
  inc
  [^long l]
  (assert (instance? Long %)))
```

```
(defspec longs-are-closed-under-increment
  inc
  [^long l]
  (assert (instance? Long %)))
```

validations throw exceptions

test development workflow

brainstorm property invariants for fut

(write helper fut)

test spec interactively

design generators for fut

run under build / CI

example: edn roundtrip

edn compatible things should roundtrip

edn incompatible things should not roundtrip

fut: roundtrip

```clojure
(defn roundtrip
  [o]
  (binding [*print-length* nil
            *print-dup* nil
            *print-level* nil]
    (try
     (-> o pr-str edn/read-string)
     (catch Throwable t t))))
```

properties

```
(defspec types-that-should-roundtrip
  roundtrip
  [^{:tag cgen/ednable} o]
  (when-not (= o %)
    (throw (ex-info "Value cannot roundtrip, see ex-data"
                    {:printed o :read %}))))
```

```
(defspec types-that-should-not-roundtrip
  roundtrip
  [^{:tag cgen/non-ednable} o]
  (when-not (instance? Throwable %)
    (throw (ex-info "edn/read should have thrown, see ex-data"
                    {:printed o :read %}))))
```

test specs interactively

```
(test-edn/types-that-should-roundtrip 10)
=> 10
```

```
(test-edn/types-that-should-roundtrip #'+)
=> ExceptionInfo Value cannot roundtrip, see ex-data
```

generators

```
(def ednable-collections
  [[gen/vec [ednable-scalars]]
   [gen/set [ednable-scalars]]
   [gen/hash-map [ednable-scalars ednable-scalars]]])

(defn ednable-collection
  []
  (let [[coll args] (rand-nth ednable-collections)]
    (apply coll (map rand-nth args))))

(defn ednable
  []
  (gen/one-of ednable-scalar ednable-collection))
```

var specs

use cases

finite generation

dependent generation

metadata on var, which must contain a map

:test fn to test

:input-gen fn generates seq of arglists

var spec example

```
(def ^::tgen/specs
  redundant-datom-elimination-spec
  [{:test redundant-datoms-should-be-eliminated
    :input-gen
    #(let [db (d/db (dgen/schema-combination-connection))]
       (map vector
            (repeat db)
            (repeatedly (partial partially-superfluous-tx db))))}])
```

test.check

property example

```
(def property
  (prop/for-all [v (gen/vector gen/int)]
    (let [s (sort v)]
      (and (= (count v) (count s))
           (ascending? s)))))
```

shrinking cases

```
(def bad-property
  (prop/for-all [v (gen/vector gen/int)]
    (ascending? v)))

(sc/quick-check 100 bad-property)

=>
{:result false, :failing-size 7, :num-tests 8,
 :fail [[-2 4 -7 5 -2 7 -4]],
 :shrunk {:total-nodes-visited 19, :depth 8, :result false,
          :smallest [[0 -1]]}}
```

initial failure

simpler failure

Design Choices

feature	t.generative	t.check	other
data generation	fns	HOs	HOs (usually)
generator decls	metadata on args	metadata on args	static types a la carte schema
predictable seed	dynamic var	?	?
varying intensity	by time	by test count	by test count
validation	whatever	properties	properties
finite inputs	metadata on fns	?	?
generator dependencies	metadata on fns	bind	bind
shrinking	no	sourdough	sourdough
state machine (seq/par/scheduler)	no	no/no/threatened	yes/yes/PULSE

Background: Example-Based Testing

Example-Based Tests (EBT)

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

EBT

setup

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

EBT

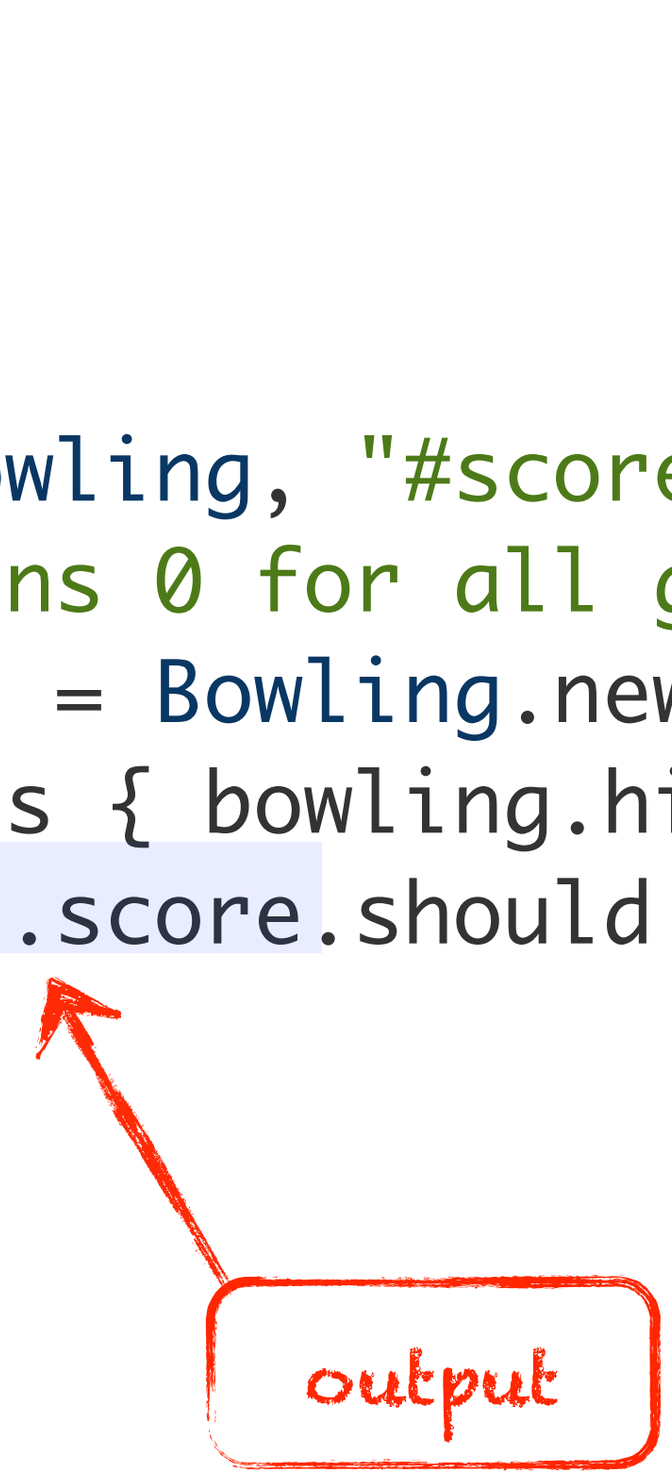
```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

EBT


```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

EBT

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```



EBT

```
describe Bowling, "#score" do
  it "returns 0 for all gutter game" do
    bowling = Bowling.new
    20.times { bowling.hit(0) }
    bowling.score.should eq(0)
  end
end
```

EBT

```
(are [x y] (= x y)
     (+) 0
     (+ 1) 1
     (+ 1 2) 3
     (+ 1 2 3) 6

     (+ -1) -1
     (+ -1 -2) -3
     (+ -1 +2 -3) -2

     (+ 2/3) 2/3
     (+ 2/3 1) 5/3
     (+ 2/3 1/3) 1 )
```

EBT

```
(are [x y] (= x y)
     (+) 0
     (+ 1) 1
     (+ 1 2) 3
     (+ 1 2 3) 6

     (+ -1) -1
     (+ -1 -2) -3
     (+ -1 +2 -3) -2

     (+ 2/3) 2/3
     (+ 2/3 1) 5/3
     (+ 2/3 1/3) 1 )
```

no setup

EBT

```
(are [x y] (= x y)
     (+)          0
     (+ 1)        1
     (+ 1 2)      3
     (+ 1 2 3)    6

     (+ -1)       -1
     (+ -1 -2)    -3
     (+ -1 +2 -3) -2

     (+ 2/3)      2/3
     (+ 2/3 1)    5/3
     (+ 2/3 1/3)  1 )
```

inputs

EBT

```
(are [x y] (= x y)
     (+)          0
     (+ 1)        1
     (+ 1 2)      3
     (+ 1 2 3)    6

     (+ -1)       -1
     (+ -1 -2)    -3
     (+ -1 +2 -3) -2

     (+ 2/3)      2/3
     (+ 2/3 1)    5/3
     (+ 2/3 1/3)  1 )
```

EBT

```
(are [x y] (= x y)
     (+)          0
     (+ 1)        1
     (+ 1 2)      3
     (+ 1 2 3)    6

     (+ -1)       -1
     (+ -1 -2)    -3
     (+ -1 +2 -3) -2

     (+ 2/3)      2/3
     (+ 2/3 1)    5/3
     (+ 2/3 1/3)  1 )
```

EBT

```
(are [x y] (= x y)
     (+)          0
     (+ 1)        1
     (+ 1 2)      3
     (+ 1 2 3)    6

     (+ -1)       -1
     (+ -1 -2)    -3
     (+ -1 +2 -3) -2

     (+ 2/3)      2/3
     (+ 2/3 1)    5/3
     (+ 2/3 1/3)  1 )
```

EBT in the Wild

Scales: Unit, Functional, Acceptance

Styles: Test-After, TDD, BDD

Common Idioms: Fixtures, Stubs, Mocks

Weaknesses of EBT

Severely limited coverage

Fragility

Poor scalability

Deconstructing EBT

Inputs

Execution

Outputs

Validation

Generative Testing

Model

Outputs

Execution

Inputs

Validation

Loose Coupling FTW

decouple	**benefits**
model	improve design generate load
inputs	increase comprehensiveness by running longer
execution	test different layers with same code only part that must change with your app
outputs	expert analysis persist for future study
validation	test generic *properties* run against prod data
all	*functional programming* *feedback loops in test development*

Solutions

Generative Tests represent categoric knowledge

Scale coverage with CPU time

Scale complexity with CPU time

Resources

Talk

https://github.com/clojure/data.generators. Data generators library.
https://github.com/clojure/test.generative. Generative testing library.
https://github.com/clojure/test.check. Generative testing library.
http://clojure.com. The Clojure language.
http://www.datomic.com/. Datomic.
http://pragprog.com/book/shcloj2/programming-clojure. *Programming Clojure*.

Finding Race Conditions in Erlang with QuickCheck and PULSE.

Stuart Halloway

https://github.com/stuarthalloway/presentations/wiki. Presentations.
http://www.linkedin.com/pub/stu-halloway/0/110/543/
https://twitter.com/stuarthalloway
mailto:stu@cognitect.com

Try It

Codebreaker is a REPL session showing a workflow for using test.generative.

The convert-to-generative sample poses example-based tests from Clojure for conversion to generative form.

ztellman's byte-transforms test is a small but complete test.check example showing generation and properties.