Generative Testing

@stuarthalloway

The Problem:
Example-Based Testing

Example-Based Tests (EBT)

```ruby
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
```

inputs

EBT

execution

```
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
```

output

EBT

```ruby
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
```

validation

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

execution

```
(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 )
```

outputs

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 )
```

validation

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*

Genesis

Datomic

Reading the Code

Extensible Data Notation (edn)

Rich set of built in data types

Generic extensibility

Language neutral

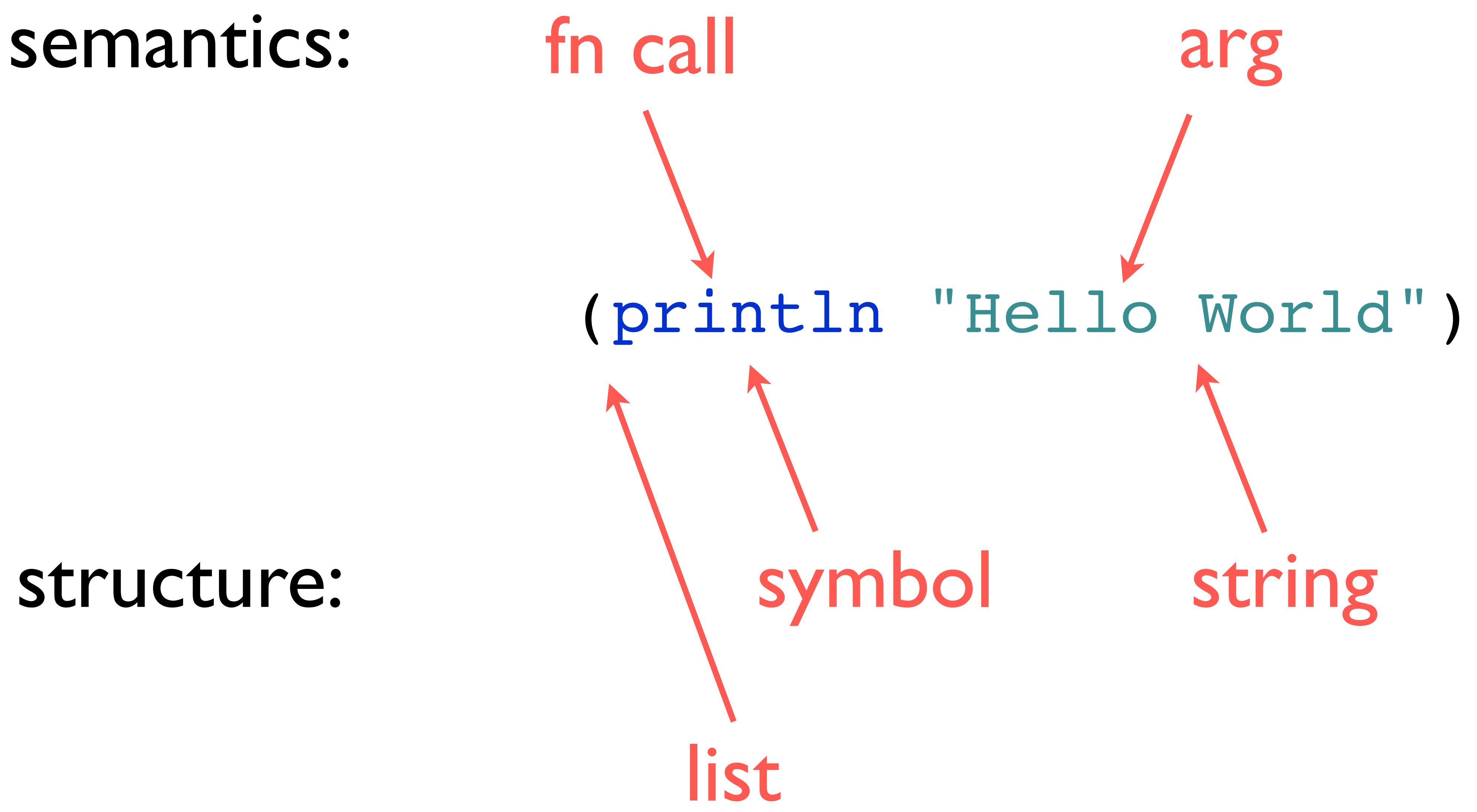
Represents values (not identities, objects)

type	example	java equivalent
string	`"foo"`	String
character	`\f`	Character
a. p. integer	`42`	Int/Long/BigInteger
double	`3.14159`	Double
a.p. double	`3.14159M`	BigDecimal
boolean	`true`	Boolean
nil	`nil`	`null`
ratio	22/7	N/A
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



```
(println "Hello World")
```

Function Definition

define a fn   fn name

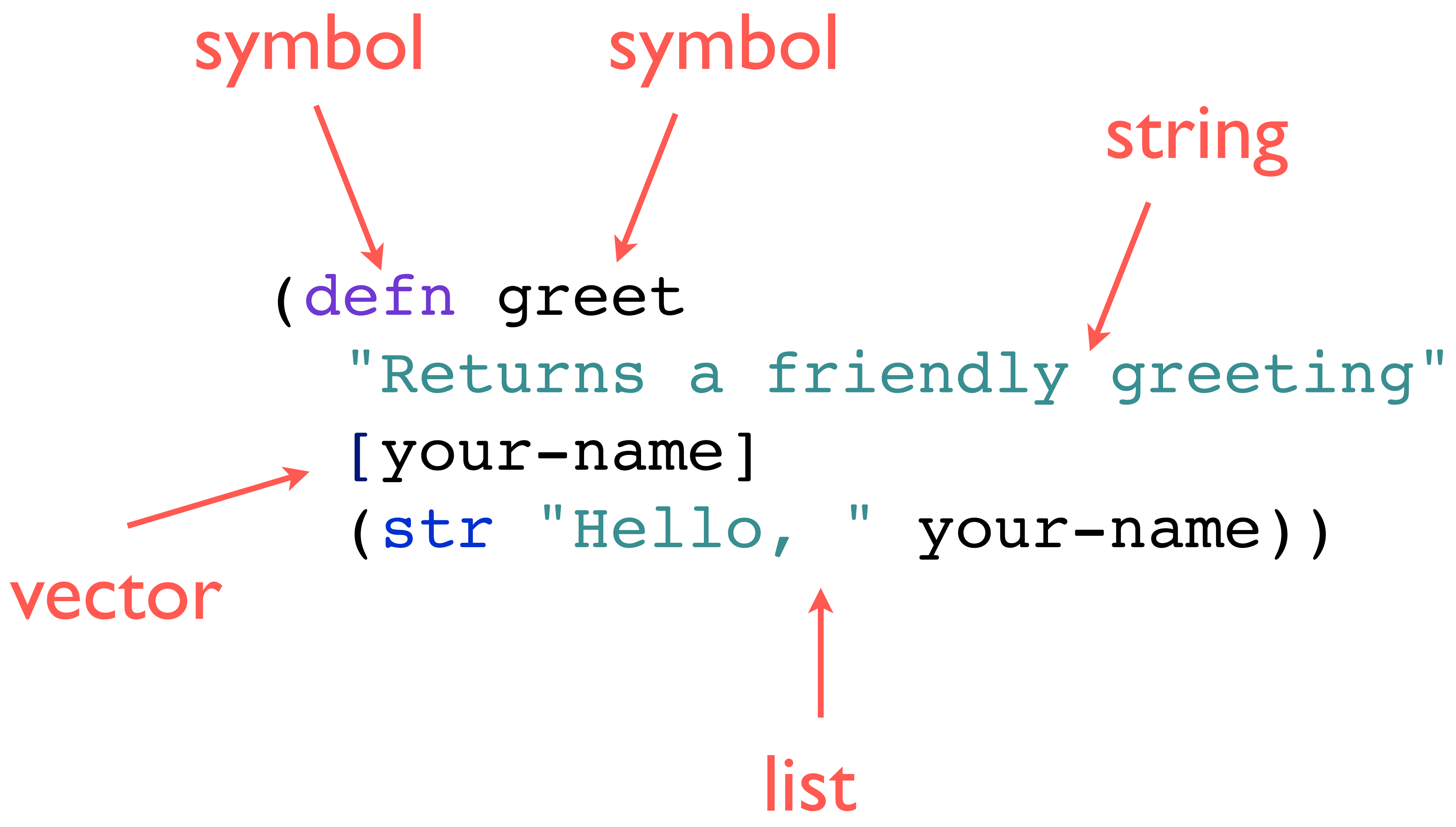
docstring

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

arguments

fn body

Still Just Data



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

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 API

add metadata

```
(def v [1 2 3])

(def trusted-v (with-meta v {:source :trusted}))

(:source (meta trusted-v)) -> :trusted
(:source (meta v)) -> nil

(= v trusted-v) -> true
```

retrieve metadata

Metadata in the Reader

metadata on [1 2 3]]

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

```
^String x
```

sugar for

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

Metadata on Vars

```
(def
 ^{:arglists '([& items])
   :doc "Creates a new list containing the items."
   :added "1.0"}
  list (. clojure.lang.PersistentList creator))
```

```
(meta (var list))
=> {:ns #<Namespace clojure.core>,
    :name list, :arglists ([& items]),
    :column 1,
    :added "1.0",
    :doc "Creates a new list containing the items.",
    :line 16,
    :file "clojure/core.clj"}
```

Metadata on Vars

metadata on the symbol "list"

```
(def
 ^{:arglists '([& items])
   :doc "Creates a new list containing the items."
   :added "1.0"}
  list (. clojure.lang.PersistentList creator))
```

```
(meta (var list))
=> {:ns #<Namespace clojure.core>,
     :name list, :arglists ([& items]),
     :column 1,
     :added "1.0",
     :doc "Creates a new list containing the items.",
     :line 16,
     :file "clojure/core.clj"}
```

Metadata on Vars

```
(def
 ^{:arglists '([& items])
   :doc "Creates a new list containing the items."
   :added "1.0"}
  list (. clojure.lang.PersistentList creator))
```

the var "list" itself, not the fn that "list" points to

```
(meta (var list))
=> {:ns #<Namespace clojure.core>,
     :name list, :arglists ([& items]),
     :column 1,
     :added "1.0",
     :doc "Creates a new list containing the items.",
     :line 16,
     :file "clojure/core.clj"}
```

Metadata on Vars

```
(def
 ^{:arglists '([& items])
   :doc "Creates a new list containing the items."
   :added "1.0"}
  list (. clojure.lang.PersistentList creator))
```

compiler copies metadata to the var, and adds more metadata

```
(meta (var list))
=> {:ns #<Namespace clojure.core>,
     :name list, :arglists ([& items]),
     :column 1,
     :added "1.0",
     :doc "Creates a new list containing the items.",
     :line 16,
     :file "clojure/core.clj"}
```

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
```

idiomatic ns
prefix

Scalar Generators

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

(gen/short)
=> 14913

(gen/uniform 0 10)
=> 6

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

value from platform range

Scalar Generators

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

(gen/short)
=> 14913

(gen/uniform 0 10)
=> 6

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

explicit distribution

Scalar Generators

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

(gen/short)
=> 14913

(gen/uniform 0 10)
=> 6

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

predictable seed
for c.c. methods

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

default size
fairly small

```
(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"}
```

$\boxed{\text{explicit size (\# or fn)}}$

Composition

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

(gen/weighted {gen/long 10 gen/keyword 1})
=> 4718031727356466O9

(gen/scalar)
=> -49

(gen/collection)
=> #{-3945240682015942560
     -4909497585342792620
     ...}
```

Composition

```
(gen/one-of gen/long gen/keyword)
=> :OBe0Mkclg7eqqQnGvcXq0m-McRzl9areH0NwR1

(gen/weighted {gen/long 10 gen/keyword 1})
=> 4718031727356466O9

(gen/scalar)
=> -49

(gen/collection)
=> #{-3945240682015942560
      -4909497585342792620
      ...}
```

$\boxed{\text{choose (equal weights)}}$

Composition

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

(gen/weighted {gen/long 10 gen/keyword 1})
=> 4718031727356466O9

(gen/scalar)
=> -49

(gen/collection)
=> #{-3945240682015942560
     -4909497585342792620
     ...}
```

explicit weights

Composition

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

(gen/weighted {gen/long 10 gen/keyword 1})
=> 4718031727356466O9

(gen/scalar)
=> -49

(gen/collection)
=> #{-3945240682015942560
     -4909497585342792620
     ...}
```

any scalar

Composition

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

(gen/weighted {gen/long 10 gen/keyword 1})
=> 47180317273564609

(gen/scalar)
=> -49

(gen/collection)
=> #{-3945240682015942560
     -4909497585342792620
     ...}
```

any collection
(of scalars)

test.generative

Objectives

Generate test inputs

Simplify data generation, execution, and validation

Knobs for intensity and duration

Produce and consume data

Play well with others

Approach

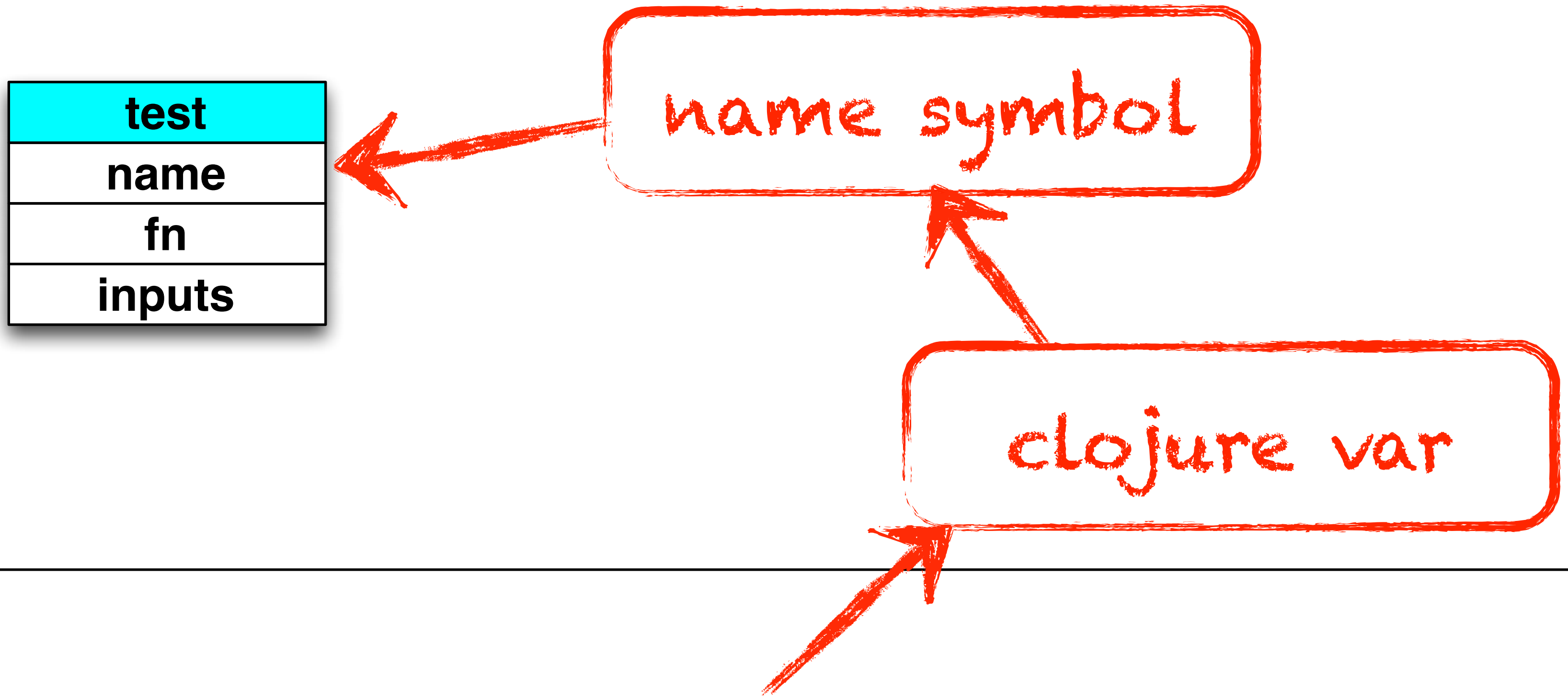
Tests are (possibly infinite) data structures

Runner executes tests, creates events

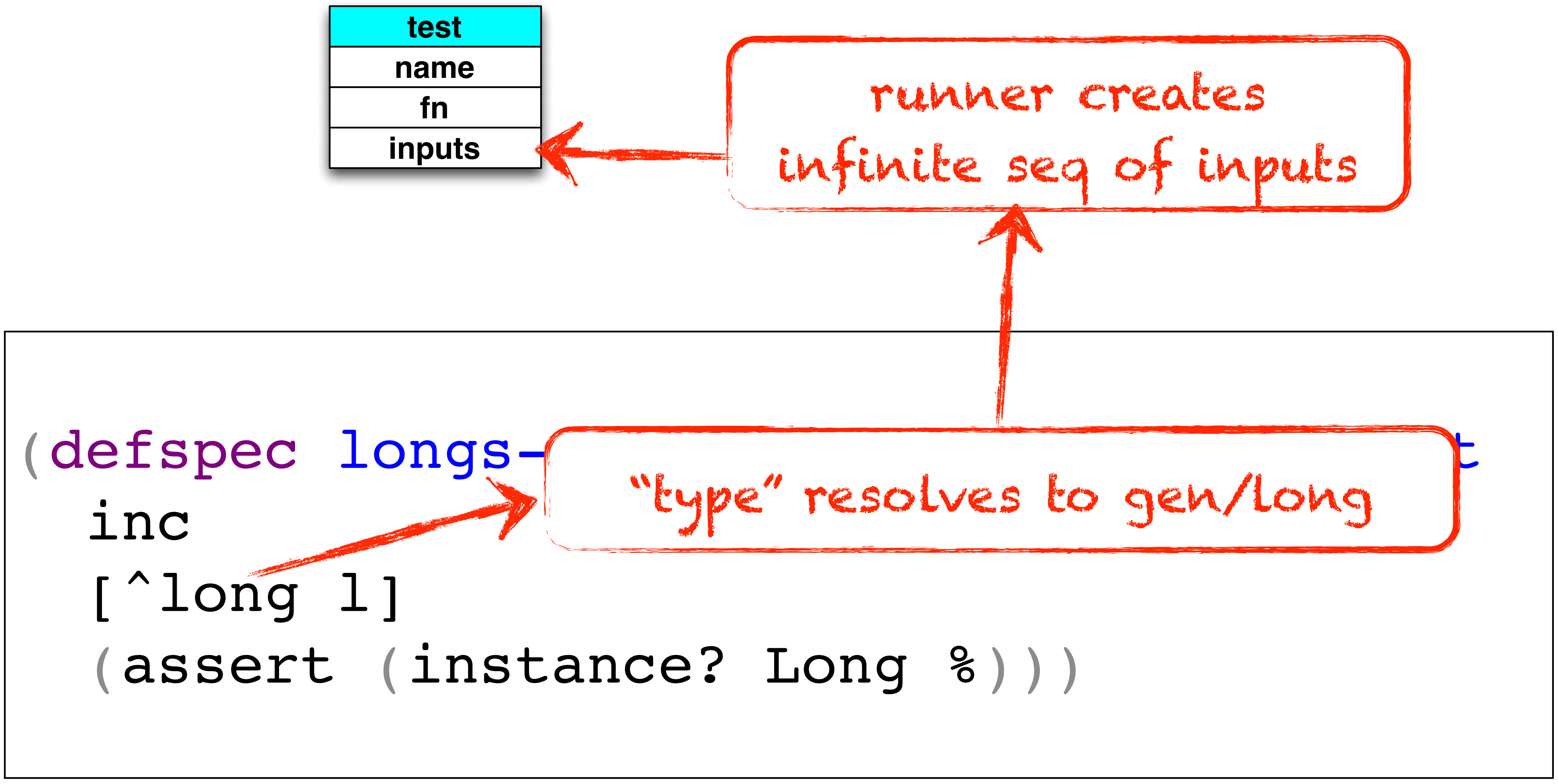
Handlers process events

DSL (defspec) is the least important part

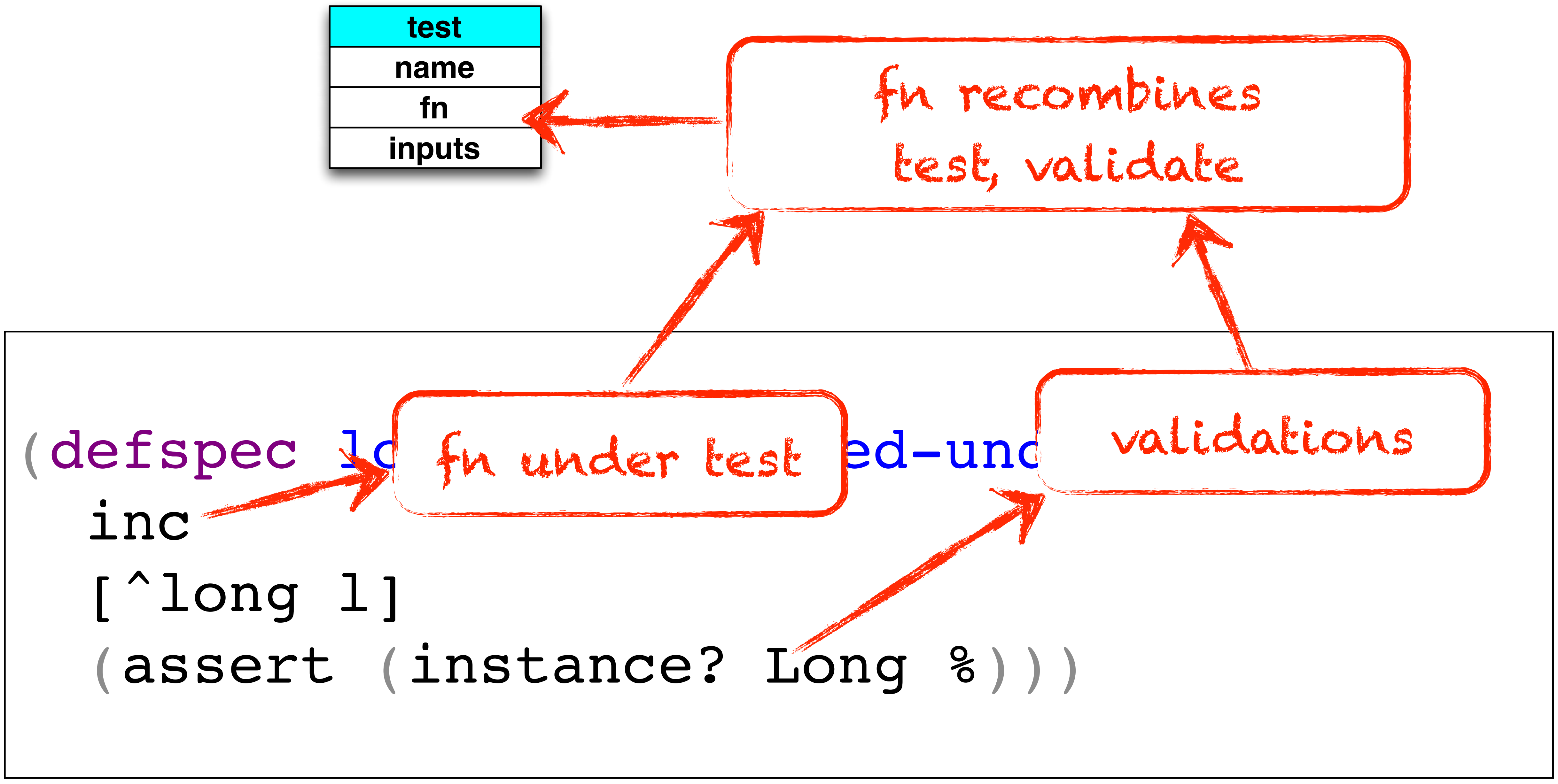
```
(defspec longs-are-closed-under-increment
  inc
  [^long l]
  (assert (instance? Long %)))
```



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


test
name
fn
inputs
runner creates infinite seq of inputs
"type" resolves to gen/long

```
(defspec longs-                                   t
  inc
  [^long l]
  (assert (instance? Long %)))
```



```
(defspec lo                ed-unc
  inc
  [^long l]
  (assert (instance? Long %)))
```

Conclusions

Let the computer do the heavy lifting

Decouple your tests

Automate your coverage

Resources

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

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@thinkrelevance.com