Extracted from:

# Programming Clojure, Third Edition



The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

# Programming Clojure

Third Edition

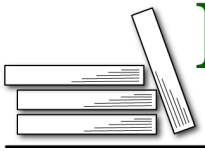
Alex Miller
with Stuart Halloway
and Aaron Bedra

*edited by Jacquelyn Carter*

# Programming Clojure, Third Edition

Alex Miller
with Stuart Halloway
and Aaron Bedra

The Pragmatic Bookshelf
Raleigh, North Carolina





Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Copy Editor: Paula Robertson
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.



Printed in the United States of America.
ISBN-13: 978-1-68050-246-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2018

## Generative Function Testing

Classic *example-based unit testing* relies on the programmer to test a function by writing a series of example inputs, then writing assertions about the return value when the function is invoked with each input.

In comparison, *generative testing* is a technique that produces thousands of random inputs, runs a procedure, and verifies a set of properties for each output. Generative testing is a great technique for getting broader test coverage of your code.

Spec can automatically perform generative testing for functions that have function specs. Let's see how that's done and then explore ways to influence those tests for more accurate coverage.

### Checking Functions

Spec implements automated generative testing with the function check in the namespace clojure.spec.test.alpha (commonly aliased as stest). You can run stest/check on any symbol or symbols that have been spec'ed with s/fdef.

#### Including the test.check Library

All invocations of spec generators (either directly or within stest/instrument, stest/check, etc.) require including the test.check library as a dependency, typically as either a test or dev profile dependency, so that it's not included at production runtime.

When you invoke check, it generates 1000 sets of arguments that are valid according to the function's :args spec. For each argument set, check invokes the function, then checks that the return value is valid according to the :ret spec and that the :fn spec is valid for the arguments and return value.

Let's see how it works with a spec for the Clojure core function symbol, which takes a name and an optional namespace:

```
(doc symbol)
| -------------------------
| clojure.core/symbol
| ([name] [ns name])
|   Returns a Symbol with the given namespace and name.
```

First we need to define the function spec:

src/examples/spec.clj

```
(s/fdef clojure.core/symbol
  :args (s/cat :ns (s/? string?) :name string?)
  :ret symbol?
```

```
  :fn (fn [{:keys [args ret]}]
        (and (= (name ret) (:name args))
          (= (namespace ret) (:ns args)))))
```

And then we can run the test as follows:

```
(stest/check 'clojure.core/symbol)
-> ({:sym clojure.core/symbol
     :spec #object[clojure.spec.alpha$fspec_impl$reify__14282 ...],
     :clojure.spec.test.check/ret {
           :result true,
           :num-tests 1000,
           :seed 1485241441400}})
```

You can see from the output that stest/check ran 1000 tests on clojure.core/symbol and found no problems. When an error occurs, the test.check library goes the extra step of "shrinking" the error to the least complicated possible input that still fails. In complex tests, this is a crucial step to produce tractable input for reproduction and fixing.

One step that we glossed over is *how* spec generated 1000 random input arguments. While we haven't mentioned it before, every spec is both a validator and a data generator for values that match the spec. Once we created a spec for the function arguments, check was able to use it to generate random arguments.

## Generating Examples

The argument spec we used above was (s/cat :ns (s/? string?) :name string?). To simulate how check generates random arguments from that spec, we can use the s/exercise function, which produces pairs of examples and their conformed values:

```
(s/exercise (s/cat :ns (s/? string?) :name string?))
-> ([("" "") {:ns "", :name ""}]
    [("F" "") {:ns "F", :name ""}]
        [("s" "73") {:ns "s", :name "73"}]
        [("u") {:name "u"}]
        [("") {:name ""}]
        [("" "3y") {:ns "", :name "3y"}]
        [("t" "9pudu") {:ns "t", :name "9pudu"}]
        [("Xhw25" "nPR7C9C") {:ns "Xhw25", :name "nPR7C9C"}]
        [("FXs3E" "N") {:ns "FXs3E", :name "N"}]
        [("UhUN5dZK1" "le8") {:ns "UhUN5dZK1", :name "le8"}])
```

This example works, but sometimes spec can't automatically create a valid generator, or you need to create a custom generator for related arguments. In the following sections, we'll see how to address these issues. First, let's consider a case where an s/and spec doesn't produce any values.

### Combining Generators With s/and

One of the most common ways to compose specs is with s/and. The s/and operation uses the generator of its first component spec, then filters the values by each subsequent component spec.

For example, consider the following spec for an odd number greater than 100:

```
(defn big? [ ] (> x 100))

(s/def ::big-odd (s/and odd? big?))
```

This would work as a spec, but its automatic generator doesn't work:

```
(s/exercise ::big-odd)
-> Unable to construct gen at: [] for: odd?
```

The problem is that while many common Clojure predicates have automatically mapped generators, the predicates we're using here do not. The odd? predicate works on more than one numeric type and so is not mapped to a generator. The big? predicate is a custom predicate that will never have mappings.

To fix this, we need to add an initial predicate that has a mapped generator—the type-oriented predicates are all good choices for that. Let's insert int? at the beginning:

```
(s/def ::big-odd-int (s/and int? odd? big?))
-> ::big-odd-int

(s/exercise ::big-odd-int)
-> ([1367 1367]
    [7669 7669]
    [171130765 171130765]
    ... )
```

When you debug generators for s/and specs, remember that only the first component spec's generator is used. Another related problem with s/and generators is when the component specs after the first one are too "sparse", filtering so many values that the generator would have to work for a long time to generate a valid one. The best way to solve this problem is with a custom generator.

### Creating Custom Generators

There are many cases where the automatic generator is either inefficient or will not produce related values that are useful. For example, a function might take two arguments—a collection and an element from that collection. An automatic generator is unlikely to independently produce valid combinations of values. Instead, you need to supply your own custom generator that satisfies this constraint.

You have a number of opportunities in spec to replace the automatically created generator with your own implementation. Some specs like s/coll-of, s/map-of, s/every, s/every-kv, and s/keys accept a custom generator option.

Also, you can explicitly add a replacement generator to any existing spec with s/with-gen. And you can temporarily override a generator by name or path with generator overrides in some functions like s/exercise, stest/instrument, and stest/check—those overrides take effect only for the duration of the call.

We can create generators in several ways. The simplest way is to first create a different spec, then use s/gen to retrieve its generator. Alternately, the clojure.spec.gen.alpha namespace, typically aliased as gen, contains other generators and functions to combine generators. The generators in this namespace are wrappers around the test.check library, so you need that library on your classpath to do any work with generators.

Let's look at how we can supply a replacement generator for :marble/color to hard-code exactly the color to return. Occasionally this is useful to reduce

the randomness of your inputs or to directly supply a complex input that would be difficult to generate.

Here we use s/with-gen to override the default generator for :marble/color (which produces marbles of all colors) and replace it with a generator that only produces red marbles:

```
(s/def :marble/color-red
  (s/with-gen :marble/color #(s/gen #{:red})))
-> :marble/color-red

(s/exercise :marble/color-red)
-> ([:red :red] [:red :red] [:red :red] ...)
```

The clojure.spec.gen.alpha namespace contains many functions to generate all of the standard Clojure types, collections, and more. One function it provides (gen/fmap) allows you to start from a source generator, then modify each generated value by applying another function.

For example, to generate strings that start with a standard prefix, you can generate the random suffix, then concatenate the prefix. The generator itself might look like this:

```
(require '[clojure.string :as str])

(s/def ::sku
  (s/with-gen (s/and string? #(str/starts-with? % "SKU-"))
    (fn [] (gen/fmap #(str "SKU-" %) (s/gen string?)))))
```

Here, gen/fmap starts with a source generator (the generator for any valid string), then applies a function to the generated values to prefix the random string with "SKU-". That generator is then attached to the spec. Let's try exercising it:

```
(s/exercise ::sku)
-> (["SKU-" "SKU-"] ["SKU-P" "SKU-P"] ["SKU-L56" "SKU-L56"] ...)
```

You now know how to adjust your specs to create better generators, and when necessary, how to replace your generators with your own custom implementations. With these tools, you can create good argument specs for your functions and automatically test your functions with generative testing.