

# LectroTest for Perl

## Complete Documentation

Tom Moertel\*

May 31, 2006

### Abstract

This document describes LectroTest for Perl, which takes the form of the `Test::LectroTest` family of Perl modules. LectroTest is an automated, specification-based testing system. Rather than forcing you to specify individual test cases, LectroTest generates them automatically according to property definitions that you provide. Each property definition represents a specification of your software's expected behavior. LectroTest tests this behavior by running your software in thousands of random test cases. When your software's actual behavior can be shown to deviate from the specifications, LectroTest will emit a "counterexample" to show where the deviation occurred. Counterexamples can be used to debug your software, and they also make great regression tests.

## Contents

<b>1</b>	<b>Test::LectroTest</b>	<b>2</b>
<b>2</b>	<b>Test::LectroTest::Tutorial</b>	<b>6</b>
<b>3</b>	<b>Test::LectroTest::Compat</b>	<b>9</b>
<b>4</b>	<b>Test::LectroTest::RegressionTesting</b>	<b>12</b>
<b>5</b>	<b>Test::LectroTest::Generator</b>	<b>14</b>
<b>6</b>	<b>Test::LectroTest::Property</b>	<b>25</b>
<b>7</b>	<b>Test::LectroTest::TestRunner</b>	<b>31</b>
<b>8</b>	<b>Test::LectroTest::FailureRecorder</b>	<b>39</b>

---

\*tom@moertel.com

# 1 Test::LectroTest

Easy, automatic, specification-based tests

## SYNOPSIS

```
#!/usr/bin/perl -w

use MyModule; # contains code we want to test
use Test::LectroTest;

Property {
    ##[ x <- Int, y <- Int ]##
    MyModule::my_function( $x, $y ) >= 0;
}, name => "my_function output is non-negative" ;

Property { ... }, name => "yet another property" ;

# more properties to check here
```

## DESCRIPTION

This module provides a simple (yet full featured) interface to LectroTest, an automated, specification-based testing system for Perl. To use it, declare properties that specify the expected behavior of your software. LectroTest then checks your software to see whether those properties hold.

Declare properties using the `Property` function, which takes a block of code and promotes it to a `Test::LectroTest::Property`:

```
Property {
    ##[ x <- Int, y <- Int ]##
    MyModule::my_function( $x, $y ) >= 0;
}, name => "my_function output is non-negative" ;
```

The first part of the block must contain a generator-binding declaration. For example:

```
##[ x <- Int, y <- Int ]##
```

(Note the special bracketing, which is required.) This particular binding says, “For all integers  $x$  and  $y$ .” (By the way, you aren’t limited to integers. LectroTest also gives you booleans, strings, lists, hashes, and more, and it lets you define your own generator types. See `Test::LectroTest::Generator` for more.)

The second part of the block is simply a snippet of code that makes use of the variables we bound earlier to test whether a property holds for the piece of software we are testing:

```
MyModule::my_function( $x, $y ) >= 0;
```

In this case, it asserts that `MyModule::my_function($x,$y)` returns a non-negative result. (Yes, `$x` and `$y` refer to the same  $x$  and  $y$  that we bound to the generators earlier. LectroTest automagically loads these lexically bound Perl variables with values behind the scenes.)

**Note:** If you want to use testing assertions like `ok` from *Test::Simple* or `is`, `like`, or `cmp_ok` from *Test::More* (and the related family of *Test::Builder*-based testing modules), see *Test::LectroTest::Compat*, which lets you mix and match *LectroTest* with these modules.

Finally, we give the whole Property a name, in this case “my\_function output is non-negative.” It’s a good idea to use a meaningful name because *LectroTest* refers to properties by name in its output.

Let’s take a look at the finished property specification:

```
Property {
    ##[ x <- Int, y <- Int ]##
    MyModule::my_function( $x, $y ) >= 0;
}, name => "my_function output is non-negative" ;
```

It says, “For all integers  $x$  and  $y$ , we assert that `my_function`’s output is non-negative.”

To check whether this property holds, simply put it in a Perl program that uses the *Test::LectroTest* module. (See the **SYNOPSIS** for an example.) When you run the program, *LectroTest* will load the property (and any others in the file) and check it by running random trials against the software you’re testing.

**Note:** If you want to place *LectroTest* property checks into a test plan managed by *Test::Builder*-based modules such as *Test::Simple* or *Test::More*, see *Test::LectroTest::Compat*.

If *LectroTest* is able to “break” your software during the property check, it will emit a counterexample to your property’s assertions and stop. You can plug the counterexample back into your software to debug the problem. (You might also want to add the counterexample to a list of regression tests.)

A successful *LectroTest* looks like this:

```
1..1
ok 1 - 'my_function output is non-negative' (1000 attempts)
```

On the other hand, if you’re not so lucky:

```
1..1
not ok 1 - 'my_function output is non-negative' falsified \
    in 324 attempts
# Counterexample:
# $x = -34
# $y = 0
```

## EXIT CODE

The exit code returned by running a suite of property checks is the number of failed checks. The code is 0 if all properties passed their checks or  $N$  if  $N$  properties failed. (If more than 254 properties failed, the exit code will be 254.)

## ADJUSTING THE TESTING PARAMETERS

There is one testing parameter (among others) that you might wish to change from time to time: the number of trials to run for each property checked. By default it is 1,000. If you want to try more or fewer trials, pass the `trials=>N` flag:

```
use Test::LectroTest trials => 10_000;
```

## TESTING FOR REGRESSIONS AND CORNER CASES

LectroTest can record failure-causing test cases to a file, and it can play those test cases back as part of its normal testing strategy. The easiest way to take advantage of this feature is to set the *regressions* parameter when you use this module:

```
use Test::LectroTest
    regressions => "regressions.txt";
```

This tells LectroTest to use the file “regressions.txt” for both recording and playing back failures. If you want to record and play back from separate files, or want only to record *or* play back, use the *record\_failures* and/or *playback\_failures* options:

```
use Test::LectroTest
    playback_failures => "regression_suite_for_my_module.txt",
    record_failures   => "failures_in_the_field.txt";
```

See *Test::LectroTest::RegressionTesting* for more.

## CAVEATS

When you use this module, it imports all of the generator-building functions from *Test::LectroTest::Generator* into the your code’s namespace. This is almost always what you want, but I figured I ought to say something about it here to reduce the possibility of surprise.

A Property specification must appear in the first column, i.e., without any indentation, in order for it to be automatically loaded and checked. If this poses a problem, let me know, and this restriction can be lifted.

## SEE ALSO

For a gentle introduction to LectroTest, see *Test::LectroTest::Tutorial*. Also, the slides from my LectroTest talk for the Pittsburgh Perl Mongers make for a great introduction. Download a copy from the LectroTest home (see below).

*Test::LectroTest::RegressionTesting* explains how to test for regressions and corner cases using LectroTest.

*Test::LectroTest::Compat* lets you mix LectroTest with the popular family of *Test::Builder*-based modules such as *Test::Simple* and *Test::More*.

*Test::LectroTest::Property* explains in detail what you can put inside of your property specifications.

*Test::LectroTest::Generator* describes the many generators and generator combinators that you can use to define the test or condition space that you want LectroTest to search for bugs.

*Test::LectroTest::TestRunner* describes the objects that check your properties and tells you how to turn their control knobs. You'll want to look here if you're interested in customizing the testing procedure.

## **LECTROTEST HOME**

The LectroTest home is <http://community.moertel.com/LectroTest>. There you will find more documentation, presentations, mailing-list archives, a wiki, and other helpful LectroTest-related resources. It's also the best place to ask questions.

## **AUTHOR**

Tom Moertel ([tom@moertel.com](mailto:tom@moertel.com))

## **INSPIRATION**

The LectroTest project was inspired by Haskell's QuickCheck module by Koen Claessen and John Hughes: <http://www.cs.chalmers.se/~rjmh/QuickCheck/>.

## **COPYRIGHT and LICENSE**

Copyright (c) 2004-05 by Thomas G Moertel. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

## 2 Test::LectroTest::Tutorial

How to use LectroTest to test your software

### SYNOPSIS

LectroTest is an automated, specification-based testing system. To use it, declare properties that specify the expected behavior of your software. Then invoke LectroTest to test whether those properties hold.

LectroTest does this by running repeated random trials against your software. If LectroTest finds that a property doesn't hold, it emits the counterexample that "broke" your software. You can then plug the counterexample into your software to debug the problem. (It's also a good idea to add the counterexample to your list of regression tests.)

### OVERVIEW

Think of your software's behavior as a haystack that you're searching for needles. Each error is a needle. You want to find the needles and remove of them. LectroTest will search the haystack for you – it's nice that way – but first you must tell it about the shape of the haystack and how to recognize a needle when it sees one.

#### The Haystack

The shape of the haystack is defined by a set of "generator bindings," in which variables are bound to the output of value generators:

```
x <- Int, c <- Char( charset=>"A-Z" )
```

The above can be read, "For all integers  $x$  and all characters  $c$  in the range A through Z." The idea is that each unique instance of the pair  $(x, c)$  specifies a point in the haystack that we can search for needles.

#### The Needle Recognizer

The "needle recognizer" is defined by a snippet of code that uses the bound variables to inspect a given point in the haystack. It returns a "thumbs up" (true) if the point is free of needles or a "thumbs down" (false) if it finds a needle:

```
the_thing_we_are_testing($x, $c) >= 0;
```

The above asserts for each point in the haystack that the output of the function `the_thing_we_are_testing` must be non-negative.

#### Put them together to make a Property

The generator bindings and needle recognizer are combined to make a property:

```
Property {
  ##[ x <- Int, c <- Char( charset=>"A-Z" ) ]##
  the_thing_we_are_testing($x, $c) >= 0;
}, name => "the_thing_we_are_testing(...) is non-negative";
```

You'll note that we also added a meaningful name. Although not strictly required, it's an excellent practice that makes life easier. (You'll also note that we placed the generator bindings inside of the magic delimiters `##[ ]##`. This tells Perl that our bindings are bindings and not regular Perl code.)

We can read the above property like so: "For all integers  $x$  and all characters  $c$  in the range A through Z, we assert that `the_thing_we_are_testing` is non-negative."

## Testing whether your Properties hold

After you define properties for your software, just add them to a small Perl program that uses the `Test::LectroTest` module:

```
# MyProperties.1.t

use MyModule; # provides the_thing_we_are_testing
use Test::LectroTest;

Property {
  ##[ x <- Int, c <- Char( charset=>"A-Z" ) ]##
  the_thing_we_are_testing($x, $c) >= 0;
}, name => "the_thing_we_are_testing(...) is non-negative";
```

Then you can test your properties simply by running the program:

```
$ perl MyProperties.1.t
```

If your properties check out, you'll see something like this:

```
1..1
ok 1 - 'the_thing_we_are_testing(...) is non-negative' (1000 attempts)
```

If something goes wrong, however, `LectroTest` will tell you where it happened:

```
1..1
not ok 1 - 'the_thing_we_are_testing(...) is non-negative' \
  falsified in 23 attempts
# Counterexample:
# $x = 4
# $c = "R"
```

What this says is that at the point ( $x=4$ ,  $c="R"$ ) in the haystack, there is a needle (i.e., your property doesn't hold). With this information, you can examine your code to determine the cause of the error.

## LET’S DO IT!

Now that we have big-picture understanding of “LectroTesting,” let’s try a few examples together.

[TODO: write the step-by-step tutorial examples. For now, take a look at the slides from my LectroTest talk for two such examples. The slides are available at the §2.]

## SEE ALSO

*Test::LectroTest* gives a quick overview of automatic, specification-based testing with LectroTest.

*Test::LectroTest::Property* explains in detail what you can put inside of your property specifications.

*Test::LectroTest::Generator* describes the many generators and generator combinators that you can use to define the shapes of the haystacks you encounter during your testing adventures.

*Test::LectroTest::TestRunner* describes the objects that check your properties and tells you how to turn their control knobs. You’ll want to look here if you’re interested in customizing the testing procedure.

## LECTROTEST HOME

The LectroTest home is <http://community.moertel.com/LectroTest>. There you will find more documentation, presentations, mailing-list archives, a wiki, and other helpful LectroTest-related resources. It’s also the best place to ask questions.

## AUTHOR

Tom Moertel ([tom@moertel.com](mailto:tom@moertel.com))

## INSPIRATION

The LectroTest project was inspired by Haskell’s QuickCheck module by Koen Claessen and John Hughes: <http://www.cs.chalmers.se/~rjmh/QuickCheck/>.

## COPYRIGHT and LICENSE

Copyright (c) 2004-05 by Thomas G Moertel. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

### 3 Test::LectroTest::Compat

Use LectroTest property checks in a Test::Simple world

#### SYNOPSIS

```
#!/usr/bin/perl -w

use MyModule; # contains code we want to test
use Test::More tests => 2;
use Test::LectroTest::Compat;

# property specs can now use Test::Builder-based
# tests such as Test::More's cmp_ok()

my $prop_nonnegative = Property {
    ##[ x <- Int, y <- Int ]##
    cmp_ok(MyModule::my_function( $x, $y ), '>=', 0);
}, name => "my_function output is non-negative" ;

# and we can now check whether properties hold
# as a Test::Builder-style test that integrates
# with other T::B tests

holds( $prop_nonnegative ); # test whether prop holds
cmp_ok( 0, '<', 1, "trivial 0<1 test" ); # a "normal" test
```

#### DESCRIPTION

This module lets you use mix LectroTest property checking with other popular Test::\* modules. With it, you can use `is()`- and `ok()`-style assertions from Test::\* modules within your LectroTest property specifications and you can check LectroTest properties as part of a Test::Simple or Test::More test plan. (You can actually take advantage of any module based on Test::Builder, not just Test::Simple and Test::More.)

The module exports a single function `holds` which is described below.

#### `holds(property, opts...)`

```
holds( $prop_nonnegative ); # check prop_nonnegative

holds( $prop_nonnegative, trials => 100 );

holds(
    Property {
        ##[ x <- Int ]##
        my_function2($x) < 0;
    }, name => "my_function2 is non-positive"
);
```

Checks whether the given property holds.

When called, this method creates a new `Test::LectroTest::TestRunner`, asks the `TestRunner` to check the property, and then reports the result to `Test::Builder`, which in turn reports to you as part of a typical

Test::Simple- or Test::More-style test plan. Any options you provide to `holds` after the property will be passed to the `TestRunner` so you can change the number of trials to run and so on. (See the docs for `new` in `Test::LectroTest::TestRunner` for the complete list of options.)

## TESTING FOR REGRESSIONS AND CORNER CASES

LectroTest can record failure-causing test cases to a file, and it can play those test cases back as part of its normal testing strategy. The easiest way to take advantage of this feature is to set the *regressions* parameter when you use this module:

```
use Test::LectroTest::Compat
    regressions => "regressions.txt";
```

This tells LectroTest to use the file “regressions.txt” for both recording and playing back failures. If you want to record and play back from separate files, or want only to record *or* play back, use the *record\_failures* and/or *playback\_failures* options:

```
use Test::LectroTest::Compat
    playback_failures => "regression_suite_for_my_module.txt",
    record_failures   => "failures_in_the_field.txt";
```

See `Test::LectroTest::RegressionTesting` for more.

\*NOTE:\* If you pass any of the recording or playback parameters to `Test::LectroTest::Compat`, you must have version 0.3500 or greater of LectroTest installed. Module authors, update your modules’ build dependencies accordingly.

## BUGS

In order to integrate with the `Test::Builder` testing harness (whose underlying testing model is somewhat incompatible with the needs of random trial-based testing) this module redefines two `Test::Builder` functions (`ok()` and `diag()`) for the duration of each property check.

## SEE ALSO

For a gentle introduction to LectroTest, see `Test::LectroTest::Tutorial`. Also, the slides from my LectroTest talk for the Pittsburgh Perl Mongers make for a great introduction. Download a copy from the LectroTest home (see below).

`Test::LectroTest::RegressionTesting` explains how to test for regressions and corner cases using LectroTest.

`Test::LectroTest::Property` explains in detail what you can put inside of your property specifications.

`Test::LectroTest::Generator` describes the many generators and generator combinators that you can use to define the test or condition space that you want LectroTest to search for bugs.

*Test::LectroTest::TestRunner* describes the objects that check your properties and tells you how to turn their control knobs. You'll want to look here if you're interested in customizing the testing procedure.

*Test::Simple* and *Test::More* explain how to do simple case-based testing in Perl.

*Test::Builder* is the test harness upon which this module is built.

## **LECTROTEST HOME**

The LectroTest home is <http://community.moertel.com/LectroTest>. There you will find more documentation, presentations, mailing-list archives, a wiki, and other helpful LectroTest-related resources. It's also the best place to ask questions.

## **AUTHOR**

Tom Moertel ([tom@moertel.com](mailto:tom@moertel.com))

## **INSPIRATION**

The LectroTest project was inspired by Haskell's QuickCheck module by Koen Claessen and John Hughes: <http://www.cs.chalmers.se/~rjmh/QuickCheck/>.

## **COPYRIGHT and LICENSE**

Copyright (c) 2004-05 by Thomas G Moertel. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

## 4 Test::LectroTest::RegressionTesting

How to do regression testing (for free!)

### SYNOPSIS

```
use Test::LectroTest
    regressions => "regressions.txt";

# -- OR --

use Test::LectroTest
    playback_failures => "regression_suite_for_my_module.txt",
    record_failures   => "failures_in_the_field.txt";

# -- OR --

use Test::LectroTest::Compat
    regressions => "regressions.txt";

# -- OR --

use Test::LectroTest::Compat
    playback_failures => "regression_suite_for_my_module.txt",
    record_failures   => "failures_in_the_field.txt";
```

### DESCRIPTION

Say that LectroTest uncovers a bug in your software by finding a random test case that proves one of your properties to be false. If you apply a fix for the bug, how can you be sure that LectroTest will re-test the property using the exact same test case that “broke” it before, just to be certain the bug really is fixed? And how can you be sure that future changes to your code will not reintroduce the same bug without your knowing it?

For situations like these, LectroTest can record failure-causing test cases to a file, and it can play those test cases back as part of its normal testing strategy.

The easiest way to take advantage of this feature is to set the *regressions* parameter when you use *Test::LectroTest* or *Test::LectroTest::Compat*:

```
use Test::LectroTest
    regressions => "regressions.txt";
```

This tells LectroTest to use the file “regressions.txt” for both recording and playing back failures. If you want to record and play back from separate files, use the *record\_failures* and *playback\_failures* options:

```
use Test::LectroTest::Compat
    playback_failures => "regression_suite_for_my_module.txt",
    record_failures   => "failures_in_the_field.txt";
```

Here is how it works:

1. When testing a property named *N*, `LectroTest` will check for a playback file. If the file exists, `LectroTest` will search it for test cases associated with *N*. If any such test cases exist, `LectroTest` will play them back *before* and *in addition to* performing the usual, random testing of the property.
2. When performing the usual, random testing of a property named *N*, if a failure occurs (i.e., `LectroTest` finds a counterexample), `LectroTest` will record the test case that caused the failure to the recording file, associating the test case with the name *N*.

\*NOTE:\* If you pass any of the recording or playback parameters to `Test::LectroTest::Compat`, you must have version 0.3500 or greater of `Test::LectroTest` installed. (Module authors, update your modules' build dependencies accordingly.) The `Test::LectroTest` module itself, however, has always ignored unfamiliar parameters, and thus these options are backward compatible with older versions.

## SEE ALSO

`Test::LectroTest` gives a quick overview of automatic, specification-based testing with `LectroTest`. This module accepts failure recording and playback options.

`Test::LectroTest::Compat` lets you mix `LectroTest` with the popular family of `Test::Builder`-based modules such as `Test::Simple` and `Test::More`. This module accepts failure recording and play-back options.

## LECTROTEST HOME

The `LectroTest` home is <http://community.moertel.com/LectroTest>. There you will find more documentation, presentations, mailing-list archives, a wiki, and other helpful `LectroTest`-related resources. It's also the best place to ask questions.

## AUTHOR

Tom Moertel ([tom@moertel.com](mailto:tom@moertel.com))

## COPYRIGHT and LICENSE

Copyright (c) 2004-06 by Thomas G Moertel. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

## 5 Test::LectroTest::Generator

Random value generators and combinators

### SYNOPSIS

```
use Test::LectroTest::Generator qw(:common :combinators);

my $int_gen = Int;
my $pct_gen = Int( range=>[0,100] );
my $flt_gen = Float( range=>[0,1] );
my $bln_gen = Bool;
my $chr_gen = Char( charset=>"a-z" );
my $str_gen = String( charset=>"A-Z0-9", length=>[3,] );
my $ary_gen = List( Int(sized=>0) );
my $hsh_gen = Hash( $str_gen, $pct_gen );
my $uni_gen = Unit( "e" ); # always returns "e"
my $elm_gen = Elements("e1", "e2", "e3", "e4");

for my $sizing_guidance (1..100) {
    my $i = $int_gen->generate( $sizing_guidance );
    print "$i ";
}
print "\n";

# generates single digits
my $digit_gen = Elements( 0..9 ); # or Int(range=>[0,9],sized=>0)

# generates SSNs like "910-77-2236"
my $ssn_gen = Paste( Paste( ($digit_gen) x 3 ),
                    Paste( ($digit_gen) x 2 ),
                    Paste( ($digit_gen) x 4 ),
                    glue => "-" );

# print 10 SSNs
print( map { $ssn_gen->generate($_)."\n" } 1..10 );

my $english_dist_vowel_gen =
    Frequency( [8.167,Unit("a")], [12.702,Unit("e")],
              [6.996,Unit("i")], [ 7.507,Unit("o")],
              [2.758,Unit("u")] );
# Source: http://www.csm.astate.edu/~rossa/datasec/frequency.html
```

### DESCRIPTION

This module provides random value generators for common data types and provides an interface and tools for creating your own generators. It also provides generator combinators that can be used to create more-complex generators by combining simple ones.

A generator is an object having a method `generate`, which takes a single argument, `size` and returns a new random value. The generated value is always a scalar. Generators that produce data structures return references to them.

## Sizing guidance

The `generate` method interprets its *size* argument as guidance about the complexity of the value it should create. Typically, smaller *size* values result in smaller generated numbers and shorter generated strings and lists. Some generators, for which sizing doesn't make sense, ignore sizing guidance altogether; those that do use sizing guidance can be told to ignore it via the `sized` modifier.

The purpose of sizing is to allow LectroTest to generate simple values at first and then, as testing progresses, to slowly ramp up the complexity. In this way, counterexamples for obvious problems will be easier for you to understand.

## Generators

The following functions create fully-formed generators, ready to use. These functions are exported into your code's namespace if you ask for `:generators` or `:all` when you `use` this module.

Each generator has a `generate` method that you can call to extract a new, random value from the generator.

### Int

```
my $gen = Int( range=>[0,9], sized=>0 );
```

Creates a generator for integer values, by default in the range `[-32768,32767]`, inclusive, but this can be changed via the optional `range` modifier.

**Int( range=>[*low*, *high*] )**

Causes the generated values to be constrained to the range `[low, high]`, inclusive. By default, the range is `[-32768, 32767]`.

**Note:** If your range is empty (i.e., `low > high`), LectroTest will complain.

**Note:** If zero is not within the range you provide, sizing makes no sense because the intersection of your range and the sizing range can be empty, and thus you must turn off sizing with `sized=>0`. If you forget, LectroTest will complain.

**Int( sized=>*bool* )**

If true (the default), constrains the absolute value of the generated integers to the sizing guidance provided to the `generate` method. Otherwise, the generated values are constrained only by the range.

### Float

```
my $gen = Float( range=>[-2.0,2.0], sized=>1 );
```

Creates a generator for floating-point values, by default in the range `[-32768.0,32768.0)`, but this can be changed via the optional `range` modifier. By default Float generators are sized.

**Float( range=>[*low*, *high*] )**

Causes the generated values to be constrained to the range `[low, high)`. By default, the range is `[-32768.0,32768.0)`. (Note that the

*high* value itself can never be generated, but values infinitesimally close to it can.)

**Note:** If your range is empty (i.e., *low* > *high*), LectroTest will complain.

**Note:** If zero is not within the range you provide, sizing makes no sense because the intersection of your range and the sizing range can be empty, and thus you must turn off sizing with `sized=>0`. If you forget, LectroTest will complain.

#### **Float( sized=>*bool* )**

If true (the default), constrains the absolute value of the generated values to the sizing guidance provided to the `generate` method. Otherwise, the generated values are constrained only by the range.

#### **Bool**

```
my $gen = Bool;
```

Creates a generator for boolean values: 0 for false, 1 for true. The generator ignores sizing guidance.

#### **Char**

```
my $gen = Char( charset=>"A-Za-z0-9_" );
```

Creates a generator for characters. By default the characters are in the ASCII range [0,127], inclusive, but this behavior can be changed with the `charset` modifier:

#### **Char( charset=>*cset* )**

Characters will be drawn from the character set given by the character-set specification *cset*. The syntax of *cset* is similar the Perl `tr` built-in and is a string comprised of characters and character ranges:

*c*

Adds the character *c* to the set.

*c-d*

Adds the characters in the range *c* through *d* (inclusive) to the set. Note: If *c* is lexicographically greater than *d*, the range is empty, and no characters will be added to the set.

Examples:

```
charset=>"abcdwxyz"
```

The characters "a", "b", "c", "d", "w", "x", "y", and "z" are in the set.

```
charset=>"a-dx-z"
```

Shorter version of the previous example.

```
charset=>"\x00-\x7f"
```

The ASCII character set.

```
charset=>"-_A-Za-z0-9"
```

The character set contains "-", "\_", upper- and lower-case ASCII letters, and the digits 0-9. Notice that the dash must occur first so that it is not misinterpreted as denoting a range of characters.

### List(*elemgen*)

```
my $gen = List( Bool, length=>[1,10] );
```

Creates a generator for lists (which are returned as array refs). The elements of the lists are generated by the generator given as *elemgen*. The lengths of the generated lists are constrained by sizing guidance at the time of generation. You can override the default sizing behavior using the optional **length** modifier:

When the list generator calls the element generator, it divides the sizing guidance by the length of the list. For example, if the list being generated will have 7 elements, when the list generator calls the element generator to generate each element, it will scale the sizing guidance by 1/7. In this way the sizing guidance provides a rough constraint on the total number of elements produced, regardless of the depth of the list structure being generated.

#### List( ..., length=>*N* )

Generated lists are exactly length *N*.

#### List( ..., length=>[*M*,] )

Generated lists are at least length *M*. (Maximum length is constrained by sizing factor.)

#### List( ..., length=>[*M*,*N*] )

Generated lists are of length between *M* and *N*, inclusive. Sizing guidance is ignored.

**Advanced Note:** If more than one *elemgen* is given, they will be used in turn to create successive elements. In this case, the length of the list will be multiplied by the number of generators given. For example, providing two generators will create double-length lists.

### Hash(*keygen*, *valgen*)

```
my $gen = Hash( String( charset=>"A-Z", length=>3 ),
               Float( range=>[0.0, 100.0] ) );
```

Creates a generator for hashes (which are returned as hash refs). The keys of the hash are generated by the generator given as *keygen*, and the values are generated by the generator *valgen*.

The Hash generator takes an optional **length** modifier that specifies the desired hash length (= number of keys):

#### Hash( ..., length=>*length-spec* )

Specifies the desired length of the generated hashes, using the same *length-spec* syntax as for the List generator. Note that the generated hashes may be smaller than expected because of key collision.

### String

```
my $gen = String( length=>[3,], charset=>"A-Z" );
```

Creates a generator for strings. By default the strings will be drawn from the ASCII character set (0 through 127) and be of length constrained by the sizing factor. Both defaults can be changed using modifiers:

**String( charset=>*cset* )**

Characters will be drawn from the character set given by the character-set specification *cset*. The syntax of *cset* is similar the Perl `tr` operator and is a string comprised of characters and character ranges. See `Char` for a full description.

**String( length=>*length-spec* )**

Specifies the desired length of generated strings, using the same *length-spec* syntax as for the `List` generator.

**Elements(*e1*, *e2*, ...)**

```
my $gen = Elements( "alpha", "beta", "gamma" );
```

Creates a generator that chooses among the given elements *e1*, *e2*, ... with equal probability. Each call to the `generate` method will return one of the element values. Sizing guidance has no effect on this generator.

**Note:** This generator builder does not accept modifiers. If you pass any, they will be interpreted as elements to be added to the pool from which the generator randomly selects, which is probably not what you want.

**Unit(*e*)**

```
my $gen = Unit( "alpha" );
```

Creates a generator that always returns the value *e*. Not too useful on its own but can be handy as a building block for combinators to chew on. Naturally, sizing guidance has no effect on this generator.

**Note:** This generator builder does not accept modifiers.

**Generator combinators**

The following combinators allow you to build more complicated generators from simpler ones. These combinators are exported into your code's namespace if you ask for `:combinators` or `:all` when you `use` this module.

**Paste(*gens...*, glue=>*str*)**

```
my $gen = Paste( (String(charset=>"0-9",length=>4)) x 4,
                 glue => " " );
# gens credit-card numbers like "4592 9459 9023 1369"

my $lgen = Paste( List( String(charset=>"0-9",length=>4)
                       , length=>4 ), glue => " " );
# another way of doing the same
```

Creates a combined generator that generates values by joining the values generated by each of the supplied sub-generators *gens*. (Generated list values will have their elements “flattened” into the rest of the generated results before joining.) The resulting string is returned.

The values are joined using the given glue string *str*. If no **glue** modifier is provided, the default glue is the empty string.

The sizing guidance given to the combined generator will be passed unchanged to each of the sub-generators.

#### **OneOf(*gens...*)**

```
my $gen = OneOf( Unit(0), List(Int,length=>3) );  
# generates scalar 0 or a 3-element list of integers
```

Creates a combined generator that generates each value by selecting at random (with equal probability) one of the sub-generators in *gens* and using that generator to generate the output value.

The sizing guidance given to the combined generator will be passed unchanged to the selected sub-generator.

**Note:** This combinator does not accept modifiers.

#### **Frequency(*[freq1, gen1], [freq2, gen2], ...*)**

```
my $gen = Frequency( [50, Unit("common"   )],  
                    [35, Unit("less common")],  
                    [15, Unit("uncommon"  )] );  
# generates one of "common", "less common", or  
# "uncommon" with respective probabilities  
# 50%, 35%, and 15%.
```

Creates a combined generator that generates each value by selecting at random one of the generators *gen1* or *gen2* or ... and using that generator to generate the output value. Each generator is selected with probability proportional to its associated frequency. (If all of the given frequencies are the same, the Frequency combinator effectively becomes OneOf.) The frequencies can be any non-negative numerical values you want and will be normalized to a 0-to-1 scale internally. At least one frequency must be greater than zero.

The sizing guidance given to the combined generator will be passed unchanged to the selected sub-generator.

**Note:** This combinator does not accept modifiers.

#### **Each(*gens...*)**

```
my $gen = Each( Unit(1), Unit("X") );  
# always generates [ 1, "X" ]
```

Creates a generator that returns a list (array ref) whose successive elements are the successive values generated by the given generators *gens*.

The sizing guidance given to the combined generator will be passed unchanged to each sub-generator.

**Note:** This combinator does not accept modifiers.

(Note for technical buffs: `Each(...)` is exactly equivalent to `List(..., length=>1)`).

#### **Apply(*fn, gens...*)**

```
my $gen = Apply( sub { $_[0] x $_[1] }  
                , Unit("X"), Unit(4) );  
# always generates "XXXX"
```

Creates a generator that applies the given function *fn* to arguments generated from each of the given sub-generators *gens* and returns the resulting value. Each sub-generator contributes one value, and the values are passed to *fn* as arguments in the same order as the sub-generators were given to `Apply`.

The sizing guidance given to the combined generator will be passed unchanged to each sub-generator.

**Note:** The function *fn* is always evaluated in scalar context. If you need to generate an array, return it as an array reference.

**Note:** This combinator does not accept modifiers.

#### `Map(fn, gens...)`

```
my $gen = Map( sub { "X" x $_[0] }
              , Unit(4), Unit(3), Unit(0) );
# always generates [ "XXXX", "XXX", "" ]
```

Creates a generator that applies the given function *fn* to the values generated by the given generators *gen* one at a time and returns a list (array ref) whose elements are each of the successive results.

The sizing guidance given to the combined generator will be passed unchanged to each sub-generator.

**Note:** The function *fn* is always evaluated in scalar context. If you need to generate an array, return it as an array reference.

**Note:** This combinator does not accept modifiers.

#### `Concat(gens...)`

```
my $gen = Concat( List( Unit(1), length=>3 )
                 , List( Unit("x"), length=>1 ) );
# always generates [1, 1, 1, "x"]
```

Creates a generator that concatenates the values generated by each of its sub-generators, resulting in a list (which is returned as a array reference). The values returned by the sub-generators are expected to be lists (array refs). If a sub-generator returns a scalar value, it will be treated like a single-element list that contains the value.

The sizing guidance given to the combined generator will be passed unchanged to each sub-generator.

**Note:** If a sub-generator returns something other than a list or scalar, you will get a run-time error.

**Note:** This combinator does not accept modifiers.

#### `Flatten(gens...)`

```
my $gen = Flatten( Unit( [[[[[ 1 ]]]]] ) );
# generates [1]
```

`Flatten` is just like `Concat` except that it recursively flattens any sublists generated by the generators *gen* and then concatenates them to generate a final a list of depth one, regardless of the depth of any sublists.

The sizing guidance given to the combined generator will be passed unchanged to each sub-generator.

**Note:** If a sub-generator returns something other than a list or scalar, you will get a run-time error.

**Note:** This combinator does not accept modifiers.

### ConcatMap(*fn*, *gens*)

```
sub take_odds { my $x = shift;
                $x % 2 ? [$x] : [] }
my $gen = ConcatMap( \&take_odds
                    , Unit(1), Unit(2), Unit(3) );
# generates [1, 3]
```

Creates a generator that applies the function *fn* to each of the values generated by the given generators *gen* in turn, and then concatenates the results.

The sizing guidance given to the combined generator will be passed unchanged to each sub-generator.

**Note:** The function *fn* is always evaluated in scalar context. If you need to generate an array, return it as an array reference.

**Note:** If a sub-generator returns something other than a list or scalar, you will get a run-time error.

**Note:** This combinator does not accept modifiers.

### FlattenMap(*fn*, *gens*)

```
my $gen = FlattenMap( sub { [ ($_[0]) x 3 ] }
                     , Unit([1]), Unit([[2]]) );
# generates [1, 1, 1, 2, 2, 2]
```

Creates a generator that applies the function *fn* to each of the values generated by the given generators *gen* in turn, and then flattens and concatenates the results.

The sizing guidance given to the combined generator will be passed unchanged to each sub-generator.

**Note:** The function *fn* is always evaluated in scalar context. If you need to generate an array, return it as an array reference.

**Note:** If a sub-generator returns something other than a list or scalar, you will get a run-time error.

**Note:** This combinator does not accept modifiers.

### Sized(*fn*, *gen*)

```
my $gen = Sized { 2 * $_[0] } List(Int);
# ^ magnify sizing guidance by factor of two
my $gen2 = Sized { 10 } Int;
# ^ use constant guidance of 10
```

Creates a generator that adjusts sizing guidance by passing it through the function *fn*. Then it calls the generator *gen* with the adjusted guidance and returns the result.

**Note:** This combinator does not accept modifiers.

## Rolling your own generators

You can create your own generators by creating any object that has a `generate` method. Your method should accept as its first argument sizing guidance *size* and, if it makes sense, adjust the complexity of the values it generates accordingly.

The easiest way to create a generator is by using the magic function `Gen`. It promotes a block of code into a generator. For example, here's a home-brew generator for times in `ctime(3)` format that is built on top of an `Int` generator:

```
use Test::LectroTest::Generator qw( :common Gen );

my $time_gen = Int(range=>[0, 2_147_483_647], sized=>0);
my $ctime_gen = Gen {
    scalar localtime $time_gen->generate( @_ );
};

print($ctime_gen->generate($_), "\n") for 1..5;
# Fri Jun  2 18:13:21 1978
# Thu Mar 28 00:55:51 1974
# Wed Mar 26 06:41:09 2025
# Sun Sep 11 15:39:44 2016
# Fri Dec 26 00:39:31 1975
```

Alternatively, we could build the generator using the `Apply` combinator:

```
my $ctime_gen2 = Apply { localtime $_[0] } $time_gen;
```

**Note:** `Gen` is not exported into your code's namespace by default. If you want to use it, you must import it by name or `import :all` when you use this module.

## EXAMPLES

Here are some examples to consider.

### Simple examples

```
use strict;
use Test::LectroTest::Generator qw(:common);

show("Ints (sized by default)", Int);

show("Floats (sized by default)", Float);

show("Percentages (unsized)",
     Int( range=>[0,100], sized=>0 ));

show("Lists (sized by default) of Ints (unsized) in [0,10]",
     List( Int( sized=>0, range=>[0,10] ) ));

show("Uppercase-alpha identifiers at least 3 chars long",
     String( length=>[3,], charset=>"A-Z" ));
```

```

show("Hashes (sized by default) of form AAA=>Digit",
     Hash( String( length=>3, charset=>"A-Z" ),
           Int( sized=>0, range=>[0,9] ) ));

sub show {
  print "\n", shift(), "\n";
  my ($gen) = @_;
  for (1..10) {
    my $val = $gen->generate($_);
    printf "Size %2d: ", $_;
    if (ref $val eq "HASH") {
      my @pairs = map {"$_=>$val->{$_}"} keys %$val;
      print "{ @pairs }";
    }
    elsif (ref $val eq "ARRAY") {
      print "[ @$val ]"
    }
    else {
      print $val;
    }
    print "\n";
  }
}

```

## Advanced examples

For these examples we use `Data::Dumper` to inspect the data structures we generate. Also, we import not only the common generator constructors (like `Int`) but also the generic `Gen` constructor, which lets us build generators out of blocks on the fly.

```

use Data::Dumper;
use Test::LectroTest::Generator qw(:common Gen);

```

First, here's a recipe for building a list of lists of integers:

```

my $loloi_gen = List( List( Int(sized=>0) ) );
print Dumper($loloi_gen->generate(10));

```

You may want to run the example several times to get a feel for the distribution of the generated output.

Now, a more complicated example. Here we build sized trees of random depth using a recursive set of generators.

```

my $tree_gen = do {
  my $density = 0.5;
  my $leaf_gen = Int( sized=>0 );
  my $tree_helper = \1;
  my $branch_gen = List( Gen { $$tree_helper->generate(@_) } );
  $tree_helper = \Gen {
    my ($size) = @_;
    return rand($size) < $density
      ? $leaf_gen->generate($size)

```

```
        : $branch_gen->generate($size + 1);
    };
    $$tree_helper;
};

print Dumper($tree_gen->generate(30));
```

We define a tree as either a leaf or a branch, and we randomly decide between the two at each node in the growing tree. Leaves are just integers and become more likely when the sizing guidance diminishes (which happens as we go deeper). The code uses `$density` as a control knob for leaf density. (Try re-running the above code after changing the value of `$density`. Try 0, 1, and 2.) Branches, on the other hand, are lists of trees. Because branches generate trees, and trees generate branches, we use a reference trick to set up the mutually recursive relationship. This we encapsulate within a `do` block for tidiness.

## SEE ALSO

*Test::LectroTest* gives a quick overview of automatic, specification-based testing with LectroTest.

## LECTROTEST HOME

The LectroTest home is <http://community.moertel.com/LectroTest>. There you will find more documentation, presentations, mailing-list archives, a wiki, and other helpful LectroTest-related resources. It's also the best place to ask questions.

## AUTHOR

Tom Moertel ([tom@moertel.com](mailto:tom@moertel.com))

## INSPIRATION

The LectroTest project was inspired by Haskell's QuickCheck module by Koen Claessen and John Hughes: <http://www.cs.chalmers.se/~rjmh/QuickCheck/>.

## COPYRIGHT and LICENSE

Copyright (c) 2004-05 by Thomas G Moertel. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

## 6 Test::LectroTest::Property

Properties that make testable claims about your software

### SYNOPSIS

```
use MyModule; # provides my_function_to_test

use Test::LectroTest::Generator qw( :common );
use Test::LectroTest::Property qw( Test );
use Test::LectroTest::TestRunner;

my $prop_non_neg = Property {
    ##[ x <- Int, y <- Int ]##
    $tcon->label("negative") if $x < 0;
    $tcon->label("odd")      if $x % 2;
    $tcon->retry             if $y == 0; # 0 can't be used in test
    my_function_to_test( $x, $y ) >= 0;
}, name => "my_function_to_test output is non-negative";

my $runner = Test::LectroTest::TestRunner->new();
$runner->run_suite(
    $prop_non_neg,
    # ... more properties here ...
);
```

### DESCRIPTION

**STOP!** If you're just looking for an easy way to write and run unit tests, see *Test::LectroTest* first. Once you're comfortable with what is presented there and ready to delve into the full offerings of properties, this is the document for you.

This module allows you to define Properties that can be checked automatically by *Test::LectroTest*. A Property is a specification of your software's required behavior over a given set of conditions. The set of conditions is given by a generator-binding specification. The required behavior is defined implicitly by a block of code that tests your software for a given set of generated conditions; if your software matches the expected behavior, the block of code returns true; otherwise, false.

This documentation serves as reference documentation for LectroTest Properties. If you don't understand the basics of Properties yet, see **OVERVIEW** in *Test::LectroTest::Tutorial* before continuing.

#### Two ways to create Properties

There are two ways to create a property:

1. Use the **Property** function to promote a block of code that contains both a generator-binding specification and a behavior test into a `Test::LectroTest::Property` object. **This is the preferred method.** Example:

```

my $prop1 = Property {
  ##[ x <- Int ]##
  thing_to_test($x) >= 0;
}, name => "thing_to_test is non-negative";

```

2. Use the `new` method of `Test::LectroTest::Property` and provide it with the necessary ingredients via named parameters:

```

my $prop2 = Test::LectroTest::Property->new(
  inputs => [ x => Int ],
  test   => sub { my ($tcon,$x) = @_;
               thing_to_test($x) >= 0 },
  name   => "thing_to_test is non-negative"
);

```

Both are equivalent, but the first is concise, easier to read, and lets `LectroTest` do some of the heavy lifting for you. The second is probably better, however, if you are constructing property specifications programmatically.

### Generator-binding specification

The generator-binding specification declares that certain variables are to be bound to certain kinds of random-value generators during the tests of your software's behavior. The number and kind of generators define the "condition space" that is examined during property checks.

If you use the `Property` function to create your properties, your generator-binding specification must come first in your code block, and you must use the following syntax:

```
##[ var1 <- gen1, var2 <- gen2, ... ]##
```

Comments are not allowed within the specification, but you may break it across multiple lines:

```
##[ var1 <- gen1,
    var2 <- gen2, ...
]##
```

or

```
##[
    var1 <- gen1,
    var2 <- gen2, ...
]##
```

Further, for better integration with syntax-highlighting IDEs, the terminating `]##` delimiter may be preceded by a hash symbol `#` and optional whitespace to make it appear like a comment:

```
##[
    var1 <- gen1,
    var2 <- gen2, ...
# ]##
```

On the other hand, if you use `Test::LectroTest::Property->new()` to create your objects, the generator-binding specification takes the form of an array reference containing variable-generator pairs that is passed to `new()` via the parameter named `inputs`:

```
inputs => [ var1 => gen1, var2 => gen2, ... ]
```

Normal Perl syntax applies here.

### Specifying multiple sets of generator bindings

Sometimes you may want to repeat a property check with multiple sets of generator bindings. This can happen, for instance, when your condition space is vast and you want to ensure that a particular portion of it receives focused coverage while still sampling the overall space. For times like this, you can list multiple sets of bindings within the `##[ and ]##` delimiters, like so:

```
##[ var1 <- gen1A, ... ],  
  [ var1 <- gen1B, ... ],  
  ... more sets of bindings ...  
  [ var1 <- gen1N, ... ]##
```

Note that only the first and last set need the special delimiters.

The equivalent when using `new()` is as follows:

```
inputs => [ [ var1 => gen1A, ... ],  
          [ var1 => gen1B, ... ],  
          ...  
          [ var1 => gen1N, ... ] ]
```

Regardless of how you declare the sets of bindings, each set must provide bindings for the exact same set of variables. (The generators, of course, can be different.) For example, this kind of thing is illegal:

```
##[ x <- Int ], [ y <- Int ]##
```

The above is illegal because both sets of bindings must use `x` or both must use `y`; they can't each use a different variable.

```
##[ x <- Int           ],  
  [ x <- Int, y <- Float ]##
```

The above is illegal because the second set has an extra variable that isn't present in the first. Both sets must use exactly the same variables. None of the variables may be extra, none may be missing, and all must be named identically across the sets of bindings.

### Behavior test

The behavior test is a subroutine that accepts a test-controller object and a given set of input conditions, tests your software's observed behavior against the required behavior with respect to the input conditions, and returns true or false to indicate acceptance or rejection. If you are using the `Property` function to create your property objects, lexically bound

variables are created and loaded with values automatically, per your input-generator specification, so you can just go ahead and use the variables immediately:

```
my $prop = Property {
  ##[ i <- Int, delta <- Float(range=>[0,1]) ]##
  my $lo_val = my_thing_to_test($i);
  my $hi_val = my_thing_to_test($i + $delta);
  $lo_val == $hi_val;
}, name => "my_thing_to_test ignores fractions" ;
```

On the other hand, if you are using `Test::LectroTest::Property->new()`, you must declare and initialize these variables manually from Perl's `@_` variable *in lexicographically increasing order* after receiving `$tcon`, the test controller object. (This inconvenience, by the way, is why the former method is preferred.) The hard way:

```
my $prop = Test::LectroTest::Property->new(
  inputs => [ i => Int, delta => Float(range=>[0,1]) ],
  test => sub {
    my ($tcon, $delta, $i) = @_;
    my $lo_val = my_thing_to_test($i);
    my $hi_val = my_thing_to_test($i + $delta);
    $lo_val == $hi_val
  },
  name => "my_thing_to_test ignores fractions"
) ;
```

## Control logic, retries, and labeling

Inside the behavior test, you have access to a special variable `$tcon` that allows you to interact with the test controller. Through `$tcon` you can do the following:

- retry the current trial with different inputs (if you don't like the inputs you were given at first)
- add labels to the current trial for reporting purposes
- attach notes and variable dumps to the current trial for diagnostic purposes, should the trial fail

(For the full details of what you can do with `$tcon` see the “testcontroller” section of *Test::LectroTest::TestRunner*.)

For example, let's say that we have written a function `my_sqrt` that returns the square root of its input. In order to check whether our implementation fulfills the mathematical definition of square root, we might specify the following property:

```
my $epsilon = 0.000_001;

Property {
  ##[ x <- Float ]##
  return $tcon->retry if $x < 0;
  $tcon->label("less than one") if $x < 1;
```

```

    my $sx = my_sqrt( $x );
    abs($sx * $sx - $x) < $epsilon;
  }, name => "my_sqrt satisfies defn of square root";

```

Because we don't want to deal with imaginary numbers, our square-root function is defined only over non-negative numbers. To make sure we don't accidentally check our property "at" a negative number, we use the following line to re-start the trial with a different input should the input we are given at first be negative:

```

    return $tcon->retry if $x < 0;

```

An interesting fact is that for all values  $x$  between zero and one, the square root of  $x$  is larger than  $x$  itself. Perhaps our implementation treats such values as a special case. In order to be confident that we are checking this case, we added the following line:

```

    $tcon->label("less than one") if $x < 1;

```

In the property-check output, we can see what percentage of the trials checked this case:

```

1..1
ok 1 - 'my_sqrt satisfies defn of square root' (1000 attempts)
# 1% less than one

```

## Trivial cases

Random-input generators may create some inputs that are trivial and don't provide much testing value. To make it easy to label such cases, you can use the following from within your behavior tests:

```

$tcon->trivial if ... ;

```

The above is exactly equivalent to the following:

```

$tcon->label("trivial") if ... ;

```

## SEE ALSO

*Test::LectroTest::Generator* describes the many generators and generator combinators that you can use to define the test or condition spaces that you want LectroTest to search for bugs.

*Test::LectroTest::TestRunner* describes the objects that check your properties and tells you how to turn their control knobs. You'll want to look here if you're interested in customizing the testing procedure.

## HERE BE SOURCE FILTERS

The special syntax used to specify generator bindings relies upon a source filter (see *Filter::Util::Call*). If you don't want to use the syntax, you can disable the filter like so:

```

use Test::LectroTest::Property qw( NO_FILTER );

```

## **LECTROTEST HOME**

The LectroTest home is <http://community.moertel.com/LectroTest>. There you will find more documentation, presentations, mailing-list archives, a wiki, and other helpful LectroTest-related resources. It's also the best place to ask questions.

## **AUTHOR**

Tom Moertel ([tom@moertel.com](mailto:tom@moertel.com))

## **INSPIRATION**

The LectroTest project was inspired by Haskell's QuickCheck module by Koen Claessen and John Hughes: <http://www.cs.chalmers.se/~rjmh/QuickCheck/>.

## **COPYRIGHT and LICENSE**

Copyright (c) 2004-05 by Thomas G Moertel. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

## 7 Test::LectroTest::TestRunner

Configurable TAP-compatible engine for running LectroTest property checks

### SYNOPSIS

```
use Test::LectroTest::TestRunner;

my @args = trials => 1_000, retries => 20_000;
my $runner = Test::LectroTest::TestRunner->new( @args );

# test a single property and print details upon failure
my $result = $runner->run( $a_single_lectrotest_property );
print $result->details unless $result->success;

# test a suite of properties, w/ Test::Harness::TAP output
my $num_successful = $runner->run_suite( @properties );
print "All passed!" if $num_successful == @properties;
```

### DESCRIPTION

**STOP!** If you just want to write and run simple tests, see *Test::LectroTest*. If you really want to learn about the property-checking apparatus or turn its control knobs, read on.

This module provides `Test::LectroTest::TestRunner`, a class of objects that tests properties by running repeated random trials. Create a `TestRunner`, configure it, and then call its `run` or `run_suite` methods to test properties individually or in groups.

### METHODS

The following methods are available.

#### **new(*named-params*)**

```
my $runner = new Test::LectroTest::TestRunner(
    trials      => 1_000,
    retries     => 20_000,
    scalefn     => sub { $_[0] / 2 + 1 },
    verbose     => 1,
    regressions => "/path/to/regression_suite.txt",
);
```

Creates a new `Test::LectroTest::TestRunner` and configures it with the given named parameters, if any. Typically, you need only provide the `trials` parameter because the other values are reasonable for almost all situations. Here is what each parameter means:

#### **trials**

The number of trials to run against each property checked. The default is 1.000.

**retries**

The number of times to allow a property to retry trials (via `$tcon->retry`) during the entire property check before aborting the check. This is used to prevent infinite looping, should the property retry every attempt.

**scalefn**

A subroutine that scales the sizing guidance given to input generators.

The TestRunner starts with an initial guidance of 1 at the beginning of a property check. For each trial (or retry) of the property, the guidance value is incremented. This causes successive trials to be tried using successively more complex inputs. The `scalefn` subroutine gets to adjust this guidance on the way to the input generators. Typically, you would change the `scalefn` subroutine if you wanted to change the rate and which inputs grow during the course of the trials.

**verbose**

If this parameter is set to true (the default) the TestRunner will use verbose output that includes things like label frequencies and counterexamples. Otherwise, only one-line summaries will be output. Unless you have a good reason to do otherwise, leave this parameter alone because verbose output is almost always what you want.

**record\_failures**

If this parameter is set to a file's pathname (or a FailureRecorder object), the TestRunner will record property-check failures to the file (or recorder). (This is an easy way to build a regression-testing suite.) If the file cannot be created or written to, this parameter will be ignored. Set this parameter to `undef` (the default) to turn off recording.

**playback\_failures**

If this parameter is set to a file's pathname (or a FailureRecorder object), the TestRunner will load previously recorded failures from the file (or recorder) and use them as *additional* test cases when checking properties. If the file cannot be read, this option will be ignored. Set this parameter to `undef` (the default) to turn off recording.

**regressions**

If this parameter is set to a file's pathname (or a FailureRecorder object), the TestRunner will load failures from and record failures to the file (or recorder). Setting this parameter is a shortcut for, and exactly equivalent to, setting `record_failures` and `<playback_failures>` to the same value, which is typically what you want when managing a persistent suite of regression tests.

This is a write-only accessor.

You can also set and get the values of the configuration properties using accessors of the same name. For example:

```
$runner->trials( 10_000 );
```

### **run(*property*)**

```
$results = $runner->run( $a_property );
print $results->summary, "\n";
if ($results->success) {
    # celebrate!
}
```

Checks whether the given property holds by running repeated random trials. The result is a `Test::LectroTest::TestRunner::results` object, which you can query for fine-grained information about the outcome of the check.

The `run` method takes an optional second argument which gives the test number. If it is not provided (usually the case), the next number available from the `TestRunner`'s internal counter is used.

```
$results = $runner->run( $third_property, 3 );
```

Additionally, if the `TestRunner`'s `playback_failures` parameter is defined, this method will play back any relevant failure cases from the given playback file (or `FailureRecorder`).

Additionally, if the `TestRunner`'s `record_failures` parameter is defined, this method will record any new failures to the given file (or `FailureRecorder`).

### **run\_suite(*properties...*)**

```
my $num_successful = $runner->run_suite( @properties );
if ($num_successful == @properties) {
    # celebrate most jubilantly!
}
```

Checks a suite of properties, sending the results of each property checked to `STDOUT` in a form that is compatible with `Test::Harness::TAP`. For example:

```
1..5
ok 1 - Property->new disallows use of 'tcon' in bindings
ok 2 - magic Property syntax disallows use of 'tcon' in bindings
ok 3 - exceptions are caught and reported as failures
ok 4 - pre-flight check catches new w/ no args
ok 5 - pre-flight check catches unbalanced arguments list
```

By default, labeling statistics and counterexamples (if any) are included in the output if the `TestRunner`'s `verbose` property is true. You may override the default by passing the `verbose` named parameter after all of the properties in the argument list:

```
my $num_successes = $runner->run_suite( @properties,
                                       verbose => 1 );
my $num_failed = @properties - $num_successes;
```

## HELPER OBJECTS

There are two kinds of objects that TestRunner uses as helpers. Neither is meant to be created by you. Rather, a TestRunner will create them on your behalf when they are needed.

The objects are described in the following subsections.

### Test::LectroTest::TestRunner::results

```
my $results = $runner->run( $a_property );
print "Property name: ", $results->name, ": ";
print $results->success ? "Winner!" : "Loser!";
```

This is the object that you get back from `run`. It contains all of the information available about the outcome of a property check and provides the following methods:

#### success

Boolean value: True if the property checked out successfully; false otherwise.

#### summary

Returns a one line summary of the property-check outcome. It does not end with a newline. Example:

```
ok 1 - Property->new disallows use of 'tcon' in bindings
```

#### details

Returns all relevant information about the property-check outcome as a series of lines. The last line is terminated with a newline. The details are identical to the summary (except for the terminating newline) unless label frequencies are present or a counterexample is present, in which case the details will have these extras (the summary does not). Example:

```
1..1
not ok 1 - 'my_sqrt meets defn of sqrt' falsified in 1 attempts
# Counterexample:
# $x = '0.546384454460178';
```

#### name

Returns the name of the property to which the results pertain.

#### number

The number assigned to the property that was checked.

#### counterexample

Returns the counterexample that “broke” the code being tested, if there is one. Otherwise, returns an empty string. If any notes have been attached to the failing trial, they will be included.

#### labels

Label counts. If any labels were applied to trials during the property check, this value will be a reference to a hash mapping each

combination of labels to the count of trials that had that particular combination. Otherwise, it will be undefined.

Note that each trial is counted only once – for the *most-specific* combination of labels that was applied to it. For example, consider the following labeling logic:

```
Property {  
  ##[ x <- Int ]##  
  $tcon->label("negative") if $x < 0;  
  $tcon->label("odd")      if $x % 2;  
  1;  
}, name => "negative/odd labeling example";
```

For a particular trial, if  $x$  was 2 (positive and even), the trial would receive no labels. If  $x$  was 3 (positive and odd), the trial would be labeled “odd”. If  $x$  was -2 (negative and even), the trial would be labeled “negative”. If  $x$  was -3 (negative and odd), the trial would be labeled “negative & odd”.

#### **label\_frequencies**

Returns a string containing a line-by-line accounting of labels applied during the series of trials:

```
print $results->label_frequencies;
```

The corresponding output looks like this:

```
25% negative  
25% negative & odd  
25% odd
```

If no labels were applied, an empty string is returned.

#### **exception**

Returns the text of the exception or error that caused the series of trials to be aborted, if the trials were aborted because an exception or error was intercepted by LectroTest. Otherwise, returns an empty string.

#### **attempts**

Returns the count of trials performed.

#### **incomplete**

In the event that the series of trials was halted before it was completed (such as when the retry count was exhausted), this method will return the reason. Otherwise, it returns an empty string.

Note that a series of trials *is* complete if a counterexample was found.

#### **Test::LectroTest::TestRunner::testcontroller**

During a live property-check trial, the variable `$tcon` is available to your Properties. It lets you label the current trial or request that it be re-tried with new inputs.

The following methods are available.

## retry

```
Property {
  ##[ x <- Int ]##
  return $tcon->retry if $x == 0;
}, ... ;
```

Stops the current trial and tells the TestRunner to re-try it with new inputs. Typically used to reject a particular case of inputs that doesn't make for a good or valid test. While not required, you will probably want to call `$tcon->retry` as part of a `return` statement to prevent further execution of your property's logic, the results of which will be thrown out should it run to completion.

The return value of `$tcon->retry` is itself meaningless; it is the side-effect of calling it that causes the current trial to be thrown out and re-tried.

## label(*string*)

```
Property {
  ##[ x <- Int ]##
  $tcon->label("negative") if $x < 0;
  $tcon->label("odd")      if $x % 2;
}, ... ;
```

Applies a label to the current trial. At the end of the trial, all of the labels are gathered together, and the trial is dropped into a bucket bearing the combined label. See the discussion of `labels` for more.

## trivial

```
Property {
  ##[ x <- Int ]##
  $tcon->trivial if $x == 0;
}, ... ;
```

Applies the label "trivial" to the current trial. It is identical to calling `label` with "trivial" as the argument.

## note(*string...*)

```
Property {
  ##[ s <- String( charset=>"A-Za-z0-9" ) ]##
  my $s_enc      = encode($s);
  my $s_enc_dec = decode($s_enc);
  $tcon->note("s_enc      = $s_enc",
            "s_enc_dec = $s_enc_dec");
  $s eq $s_enc_dec;
}, name => "decode is encode's inverse" ;
```

Adds a note (or notes) to the current trial. In the event that the trial fails, these notes will be emitted as part of the counterexample. For example:

```
1..1
not ok 1 - property 'decode is encode's inverse' \
```

```

        falsified in 68 attempts
#   Counterexample:
#   $s = "0";
#   Notes:
#   $s_enc      = "";
#   $s_enc_dec = "";

```

Notes can help you debug your code when something goes wrong. Use them as debugging hints to yourself. For example, you can use notes to record the output of each stage of a multi-stage test. That way, if the test fails, you can see what happened in each stage without having to plug the counterexample into your code under a debugger.

If you want to include complicated values or data structures in your notes, see the `dump` method, next, which may be more appropriate.

`dump(value, name)`

```

Property {
  ##[ s <- String ]##
  my $s_enc      = encode($s);
  my $s_enc_dec = decode($s_enc);
  $tcon->dump($s_enc, "s_enc");
  $tcon->dump($s_enc_dec, "s_enc_dec");
  $s eq $s_enc_dec;
}, name => "decode is encode's inverse" ;

```

Adds a note to the current trial in which the given *value* is dumped. The value will be dumped via *Data::Dumper* and thus may be complex and contain weird control characters and so on. If you supply a *name*, it will be used to name the dumped value. Returns *value* as its result.

In the event that the trial fails, the note (and any others) will be emitted as part of the counterexample.

See `note` above for more.

## SEE ALSO

*Test::LectroTest::Property* explains in detail what you can put inside of your property specifications.

*Test::LectroTest::RegressionTesting* explains how to test for regressions and corner cases using *LectroTest*.

*Test::Harness:TAP* documents the Test Anything Protocol, Perl's simple text-based interface between testing modules such as *Test::LectroTest* and the test harness *Test::Harness*.

## LECTROTEST HOME

The *LectroTest* home is <http://community.moertel.com/LectroTest>. There you will find more documentation, presentations, mailing-list archives, a wiki, and other helpful *LectroTest*-related resources. It's also the best place to ask questions.

## **AUTHOR**

Tom Moertel (tom@moertel.com)

## **INSPIRATION**

The LectroTest project was inspired by Haskell's QuickCheck module by Koen Claessen and John Hughes: <http://www.cs.chalmers.se/~rjmh/QuickCheck/>.

## **COPYRIGHT and LICENSE**

Copyright (c) 2004-06 by Thomas G Moertel. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

## 8 Test::LectroTest::FailureRecorder

Records/plays failures for regression testing

### SYNOPSIS

```
use Test::LectroTest::Recorder;

my $recorder = Test::LectroTest::Recorder->new("storage_file.txt");

my $recorder->record_failure_for_property(
    "property name",
    $input_hashref_from_counterexample
);

my $failures = $recorder->get_failures_for_property("property name");
for my $input_hashref (@$failures) {
    # do something with hashref
}
```

### DESCRIPTION

This module provides a simple means of recording property-check failures so they can be reused as regression tests. You do not need to use this module yourself because the higher-level LectroTest modules will use it for you when needed. (These docs are mainly for LectroTest developers.)

The basic idea is to record a failure as a pair of the form

```
[ <property_name>, <input hash from counterexample> ]
```

and Dump these pairs into a text file, each record terminated by blank line so that the file can be read using paragraph-slurp mode.

The module provides methods to add such pairs to a recorder file and to retrieve the recorded failures by property name. It uses a cache to avoid repetitive reads.

### METHODS

#### *new(storage-file)*

```
my $recorder = Test::LectroTest::Recorder->new("/path/to/storage.txt");
```

Creates a new recorder object and tells it to use *storage-file* for the reading and writing of failures.

The recorder will not access the storage file until you attempt to get or record a failure. Thus it is OK to specify a storage file that does not yet exist, provided you record failures to it before you attempt to get failures from it.

### **get\_failures\_for\_property(*propname*)**

```
my $failures = $recorder->get_failures_for_property("property name");
for my $input_hashref (@$failures) {
    # do something with hashref
    while (my ($var, $value) = each %$input_hashref) {
        # ...
    }
}
```

Returns a reference to an array that contains the recorded failures for the property with the name *propname*. In the event no such failures exist, the array will be empty. Each failure is represented by a hash containing the inputs that caused the failure.

If the recorder's storage file does not exist or cannot be opened for reading, this method dies. Thus, you should call it from within an `eval` block.

### **record\_failure\_for\_property(*propname*, *input-hashref*)**

```
my $recorder->record_failure_for_property(
    "property name",
    $input_hashref_from_counterexample
);
```

Adds a failure record for the property named *propname*. The record captures the counterexample represented by the *input-hashref*. The record is immediately appended to the recorder's storage file.

Returns 1 upon success; dies otherwise.

If the recorder's storage file cannot be opened for writing, this method dies. Thus, you should call it from within an `eval` block.

## **SEE ALSO**

*Test::LectroTest::TestRunner* explains the internal testing apparatus, which uses the failure recorders to record and play back failures for regression testing.

## **LECTROTEST HOME**

The LectroTest home is <http://community.moertel.com/LectroTest>. There you will find more documentation, presentations, mailing-list archives, a wiki, and other helpful LectroTest-related resources. It's also the best place to ask questions.

## **AUTHOR**

Tom Moertel ([tom@moertel.com](mailto:tom@moertel.com))

## **COPYRIGHT and LICENSE**

Copyright (c) 2004-06 by Thomas G Moertel. All rights reserved.

This program is free software; you can redistribute it and/or modify it under the same terms as Perl itself.