RackUnit: Unit Testing

Version 9.4.0.4

Noel Welsh <noelwelsh@gmail.com>
and Ryan Culpepper <ryanc@racket-lang.org>

November 3, 2026

RackUnit is a unit-testing framework for Racket. It is designed to handle the needs of all Racket programmers, from novices to experts.

Contents

1 Quick Start Guide for RackUnit **4**

2 The Philosophy of RackUnit **7**

2.1 Historical Context 8

3 RackUnit API **9**

3.1 Overview of RackUnit 9

3.2 Checks 9

3.2.1 Basic Checks 9

3.2.2 Augmenting Information on Check Failure 19

3.2.3 Custom Checks 24

3.3 Compound Testing Forms 27

3.3.1 Test Cases 27

3.3.2 Test Suites 29

3.4 Test Control Flow 30

3.5 Miscellaneous Utilities 31

3.6 User Interfaces 32

3.6.1 Textual User Interface 33

3.6.2 Graphical User Interface 33

4 Testing Utilities **36**

4.1 Checking documentation completeness 36

4.2 Logging Test Results 36

5 RackUnit Internals and Extension API **38**

5.1 Customizing Check Evaluation 38

5.2 Customizing Test Evaluation 38
5.3 Programmatically Running Tests and Inspecting Results 39
5.3.1 Result Types 39
5.3.2 Functions to Run Tests 40

6 Release Notes **44**
6.1 Version 3.4 44
6.2 Version 3 44

7 Acknowlegements **45**

Index **46**

Index **46**

1 Quick Start Guide for RackUnit

Suppose we have code contained in `file.rkt`, which implements buggy versions of `+` and `*` called `my-+` and `my-*`:

```
#lang racket/base

(define (my-+ a b)
  (if (zero? a)
      b
      (my-+ (sub1 a) (add1 b))))

(define (my-* a b)
  (if (zero? a)
      b
      (my-* (sub1 a) (my-+ b b))))

(provide my-+
         my-*)
```

We want to test this code with RackUnit. We start by creating a file called `file-test.rkt` to contain our tests. At the top of `file-test.rkt` we import RackUnit and `file.rkt`:

```
#lang racket/base

(require rackunit
         "file.rkt")
```

Now we add some tests to check our library:

```
(check-equal? (my-+ 1 1) 2 "Simple addition")
(check-equal? (my-* 1 2) 2 "Simple multiplication")
```

This is all it takes to define tests in RackUnit. Now evaluate this file and see if the library is correct. Here's the result I get:

```
--------------------
FAILURE
name:       check-equal?
location:   (file-test.rkt 7 0 117 27)
expression: (check-equal? (my-* 1 2) 2)
params:     (4 2)
message:    "Simple multiplication"
actual:     4
```

```
expected:   2

--------------------
```

The first test passed and so prints nothing. The second test failed, as shown by the message.

Requiring RackUnit and writing checks is all you need to get started testing, but let's take a little bit more time to look at some features beyond the essentials.

Let's say we want to check that a number of properties hold. How do we do this? So far we've only seen checks of a single expression. In RackUnit a check is always a single expression, but we can group checks into units called test cases. Here's a simple test case written using the `test-begin` form:

```
(test-begin
 (let ([lst (list 2 4 6 9)])
   (check = (length lst) 4)
   (for-each
    (lambda (elt)
      (check-pred even? elt))
    lst)))
```

Evaluate this and you should see an error message like:

```
--------------------
A test
... has a FAILURE
name:       check-pred
location:   (#<path:/Users/noel/programming/schematics/rackunit/branches/v3/doc/file-
test.rkt> 14 6 252 22)
expression: (check-pred even? elt)
params:     (#<procedure:even?> 9)
--------------------
```

This tells us that the expression `(check-pred even? elt)` failed. The arguments of this check were `even?` and `9`, and as 9 is not even the check failed. A test case fails as soon as any check within it fails, and no further checks are evaluated once this takes place.

Naming our test cases is useful as it helps remind us what we're testing. We can give a test case a name with the `test-case` form:

```
(test-case
 "List has length 4 and all elements even"
 (let ([lst (list 2 4 6 9)])
   (check = (length lst) 4)
   (for-each
    (lambda (elt)
```

```
      (check-pred even? elt))
    lst)))
```

Now if we want to structure our tests a bit more we can group them into a test suite:

```
(define file-tests
  (test-suite
   "Tests for file.rkt"

   (check-equal? (my-+ 1 1) 2 "Simple addition")

   (check-equal? (my-* 1 2) 2 "Simple multiplication")

   (test-case
    "List has length 4 and all elements even"
    (let ([lst (list 2 4 6 9)])
      (check = (length lst) 4)
      (for-each
        (lambda (elt)
          (check-pred even? elt))
      lst)))))
```

Evaluate the module now and you'll see the tests no longer run. This is because test suites delay execution of their tests, allowing you to choose how you run your tests. You might, for example, print the results to the screen or log them to a file.

Let's run our tests, using RackUnit's simple textual user interface (there are fancier interfaces available but this will do for our example). In `file-test.rkt` add the following lines:

```
(require rackunit/text-ui)

(run-tests file-tests)
```

Now evaluate the file and you should see similar output again.

These are the basics of RackUnit. Refer to the documentation below for more advanced topics, such as defining your own checks. Have fun!

2 The Philosophy of RackUnit

RackUnit is designed to allow tests to evolve in step with the evolution of the program under testing. RackUnit scales from the unstructured checks suitable for simple programs to the complex structure necessary for large projects.

Simple programs, such as those in How to Design Programs, are generally purely functional with no setup required to obtain a context in which the function may operate. Therefore the tests for these programs are extremely simple: the test expressions are single checks, usually for equality, and there are no dependencies between expressions. For example, a HtDP student may be writing simple list functions such as length, and the properties they are checking are of the form:

```
(equal? (length null) 0)
(equal? (length '(a)) 1)
(equal? (length '(a b)) 2)
```

RackUnit directly supports this style of testing. A check on its own is a valid test. So the above examples may be written in RackUnit as:

```
(check-equal? (length null) 0)
(check-equal? (length '(a)) 1)
(check-equal? (length '(a b)) 2)
```

Simple programs now get all the benefits of RackUnit with very little overhead.

There are limitations to this style of testing that more complex programs will expose. For example, there might be dependencies between expressions, caused by state, so that it does not make sense to evaluate some expressions if earlier ones have failed. This type of program needs a way to group expressions so that a failure in one group causes evaluation of that group to stop and immediately proceed to the next group. In RackUnit all that is required is to wrap a `test-begin` expression around a group of expressions:

```
(test-begin
  (setup-some-state!)
  (check-equal? (foo! 1) 'expected-value-1)
  (check-equal? (foo! 2) 'expected-value-2))
```

Now if any expression within the `test-begin` expression fails no further expressions in that group will be evaluated.

Notice that all the previous tests written in the simple style are still valid. Introducing grouping is a local change only. This is a key feature of RackUnit's support for the evolution of the program.

The programmer may wish to name a group of tests. This is done using the `test-case` expression, a simple variant on test-begin:

```
(test-case
  "The name"
  ... test expressions ...)
```

Most programs will stick with this style. However, programmers writing very complex programs may wish to maintain separate groups of tests for different parts of the program, or run their tests in different ways to the normal RackUnit manner (for example, test results may be logged for the purpose of improving software quality, or they may be displayed on a website to indicate service quality). For these programmers it is necessary to delay the execution of tests so they can be processed in the programmer's chosen manner. To do this, the programmer simply wraps a test-suite around their tests:

```
(test-suite
  "Suite name"
  (check ...)
  (test-begin ...)
  (test-case ...))
```

The tests now change from expressions that are immediately evaluated to objects that may be programmatically manipulated. Note again this is a local change. Tests outside the suite continue to evaluate as before.

2.1 Historical Context

Most testing frameworks, including earlier versions of RackUnit, support only the final form of testing. This is likely due to the influence of the SUnit testing framework, which is the ancestor of RackUnit and the most widely used frameworks in Java, .Net, Python, and Ruby, and many other languages. That this is insufficient for all users is apparent if one considers the proliferation of "simpler" testing frameworks in Scheme such as SRFI-78, or the practice of beginner programmers. Unfortunately these simpler methods are inadequate for testing larger systems. To the best of my knowledge RackUnit is the only testing framework that makes a conscious effort to support the testing style of all levels of programmer.

3 RackUnit API

```
(require rackunit)
```
package: rackunit-lib

3.1 Overview of RackUnit

There are three basic concepts in RackUnit:

- A *check* is the basic unit of a test. As the name suggests, it checks whether some condition is true.
- A *test case* is a group of checks that form one conceptual unit. If any check within the case fails, the entire case fails.
- A *test suite* is a group of test cases and test suites that has a name.

3.2 Checks

Checks are the basic building block of RackUnit. A check checks some condition and always evaluates to `(void)`. If the condition doesn't hold, the check will report the failure using the current check-info stack (see `current-check-handler` for customizing how failures are handled).

Although checks are implemented as macros, which is necessary to grab source locations (see §3.2.3 "Custom Checks"), they are conceptually functions (with the exception of `check-match` below). This means, for instance, checks always evaluate their arguments. You can use a check as a first class function, though this will affect the source location that the check grabs.

Also, if the evaluation of the arguments to one of the checks raises an exception (except `exn:break?`) the exception is caught and the test case is considered to have failed.

3.2.1 Basic Checks

The following are the basic checks RackUnit provides. You can create your own checks using `define-check`.

```
(check-eq? v1 v2 [message]) → void?
  v1 : any/c
  v2 : any/c
  message : (or/c string? #f) = #f
(check-not-eq? v1 v2 [message]) → void?
```

```
  v1 : any/c
  v2 : any/c
  message : (or/c string? #f) = #f
(check-eqv? v1 v2 [message]) → void?
  v1 : any/c
  v2 : any/c
  message : (or/c string? #f) = #f
(check-not-eqv? v1 v2 [message]) → void?
  v1 : any/c
  v2 : any/c
  message : (or/c string? #f) = #f
(check-equal? v1 v2 [message]) → void?
  v1 : any/c
  v2 : any/c
  message : (or/c string? #f) = #f
(check-not-equal? v1 v2 [message]) → void?
  v1 : any/c
  v2 : any/c
  message : (or/c string? #f) = #f
```

Checks that *v1* is equal (or not equal) to *v2*, using `eq?`, `eqv?`, or `equal?`, respectively. The optional *message* is included in the output if the check fails.

For example, the following checks all fail:

```
> (check-eq? (list 1) (list 1) "allocated data not eq?")
--------------------
FAILURE
name:       check-eq?
location:   eval:3:0
message:    "allocated data not eq?"
actual:     '(1)
expected:   '(1)
--------------------
> (check-not-eq? 1 1 "fixnums are eq?")
--------------------
FAILURE
name:       check-not-eq?
location:   eval:4:0
params:     '(1 1)
message:    "fixnums are eq?"
--------------------
> (check-eqv? 1 1.0 "not eqv?")
--------------------
FAILURE
name:       check-eqv?
```

```
location:   eval:5:0
message:    "not eqv?"
actual:     1
expected:   1.0
--------------------
> (check-not-eqv? 1 1 "integers are eqv?")
--------------------
FAILURE
name:       check-not-eqv?
location:   eval:6:0
params:     '(1 1)
message:    "integers are eqv?"
--------------------
> (check-equal? 1 1.0 "not equal?")
--------------------
FAILURE
name:       check-equal?
location:   eval:7:0
message:    "not equal?"
actual:     1
expected:   1.0
--------------------
> (check-not-equal? (list 1) (list 1) "equal?")
--------------------
FAILURE
name:       check-not-equal?
location:   eval:8:0
params:     '((1) (1))
message:    "equal?"
--------------------
> (check-equal? (car #f) (car #f))
--------------------
ERROR
name:       check-equal?
location:   eval:9:0

car: contract violation
  expected: pair?
  given: #f
--------------------
```

```
(check-pred pred v [message]) → void?
  pred : (-> any/c any/c)
  v : any/c
  message : (or/c string? #f) = #f
```

Checks that *pred* returns a value that is not #f when applied to *v*. The optional *message* is included in the output if the check fails. The value returned by a successful check is the value returned by *pred*.

For example, the following check passes:

```
> (check-pred string? "I work")
```

The following check fails:

```
> (check-pred number? "I fail")
--------------------
FAILURE
name:          check-pred
location:    eval:11:0
params:        '(#<procedure:number?> "I fail")
--------------------
```

```
(check-= v1 v2 epsilon [message]) → void?
  v1 : number?
  v2 : number?
  epsilon : number?
  message : (or/c string? #f) = #f
```

Checks that *v1* and *v2* are numbers within *epsilon* of one another. Usually the first number is produced by a function, the second number is the expected value, and epsilon is the tolerance. The optional *message* is included in the output if the check fails.

For example, the following check passes:

```
> (define (golden-ratio) 1.62) ; computes the golden ratio
> (check-= (golden-ratio) 1.618033988749 0.01 "I work")
```

The following check fails:

```
> (check-= (golden-ratio) 1.618033988749 0.001 "I fail")
--------------------
FAILURE
name:          check-=
location:    eval:14:0
message:       "I fail"
actual:        1.62
expected:    1.618033988749
tolerance:   0.001
--------------------
```

```
(check-within v1 v2 epsilon [message]) → void?
  v1 : any/c
  v2 : any/c
  epsilon : number?
  message : (or/c string? #f) = #f
```

Checks that *v1* and *v2* are `equal?` to each other, while allowing numbers *inside* of them to be different by at most *epsilon* from one another. If `(equal? v1 v2)` would call `equal?` on sub-pieces that are numbers, then those numbers are considered "good enough" if they're within *epsilon*. In other words, this check is similar to `check-=` except it works on data structures like lists, flvectors, and hash tables.

For example, the following checks pass:

```
> (define (avogadro-constant) 6e+23) ; computes the Avogadro con-
stant
> (define (gravitational-acc) 10)    ; computes the gravitational
acceleration
> (check-within (list (avogadro-constant) (gravitational-acc))
                (list 6.02214076e+23 9.80665) 5e+21)
> (check-within (hash 'C 20 'F 68) (hash 'C 25 'F 77) 10)
> (check-within (flvector 3.0 4.0 5.0) (flvector 3.01 4.01 5.014) 0.02)
```

And the following checks fail:

```
> (check-within (list (avogadro-constant) (gravitational-acc))
                (list 6.02214076e+23 9.80665) 1e+21)
--------------------
FAILURE
name:          check-within
location:    eval:20:0
actual:       '(6e+23 10)
expected:    '(6.02214076e+23 9.80665)
tolerance:   1e+21
--------------------
> (check-within (hash 'C 18 'F 64) (hash 'C 25 'F 77) 10)
--------------------
FAILURE
name:          check-within
location:    eval:21:0
actual:       '#hash((C . 18) (F . 64))
expected:    '#hash((C . 25) (F . 77))
tolerance:   10
--------------------
```

```
> (check-within (flvector 3.0 4.0 5.0) (flvector 3.01 4.01 5.014) 1e-
5)
--------------------
FAILURE
name:       check-within
location:   eval:22:0
actual:     (flvector 3.0 4.0 5.0)
expected:   (flvector 3.01 4.01 5.014)
tolerance:  1e-5
--------------------
```

Added in version 1.10 of package `rackunit-lib`.

```
(check-true v [message]) → void?
  v : any/c
  message : (or/c string? #f) = #f
(check-false v [message]) → void?
  v : any/c
  message : (or/c string? #f) = #f
(check-not-false v [message]) → void?
  v : any/c
  message : (or/c string? #f) = #f
```

Checks that *v* is `#t`, is `#f`, or is not `#f`, respectively. The optional *message* is included in the output if the check fails.

For example, the following checks all fail:

```
> (check-true 1)
--------------------
FAILURE
name:       check-true
location:   eval:23:0
params:     '(1)
--------------------
> (check-false 1)
--------------------
FAILURE
name:       check-false
location:   eval:24:0
params:     '(1)
--------------------
> (check-not-false #f)
--------------------
FAILURE
name:       check-not-false
```

```
location:   eval:25:0
params:     '(#f)
--------------------
```

```
(check-exn exn-predicate thunk [message]) → void?
  exn-predicate : (or/c (-> any/c any/c) regexp?)
  thunk : (-> any)
  message : (or/c string? #f) = #f
```

Checks that *thunk* raises an exception and that either *exn-predicate* returns a true value if it is a function, or that it matches the message in the exception if *exn-predicate* is a regexp. In the latter case, the exception raised must be an exn:fail?. The optional *message* is included in the output if the check fails. A common error is to use an expression instead of a function of no arguments for *thunk*. Remember that checks are conceptually functions.

For example, the following checks succeed:

```
> (check-exn
   exn:fail?
   (lambda ()
     (raise (make-exn:fail "Hi there"
                           (current-continuation-marks)))))
> (check-exn
   #rx"[Hh]i there"
   (lambda ()
     (raise (make-exn:fail "Hi there"
                           (current-continuation-marks)))))
> (check-exn
   exn:fail?
   (lambda ()
     (error 'hi "there")))
> (check-exn exn:fail:contract:divide-by-zero?
             (lambda ()
               (/ 1 0)))
```

The following check fails:

```
> (check-exn exn:fail:contract:divide-by-zero?
             (lambda ()
               (car #f)))
--------------------
FAILURE
name:       check-exn
location:   eval:30:0
```

```
params:          '(#<procedure:exn:fail:contract:divide-by-zero?> #<procedure>)
message:         "Wrong exception raised"
exn-message:     "car: contract violation\n  expected: pair?\n  given: #f"
exn:
  #(struct:exn:fail:contract "car: contract violation\n  expected: pair?\n  given:
#f" #<continuation-mark-set>)
--------------------
> (check-exn
   #rx"Hello there"
   (lambda ()
     (raise (make-exn:fail "Hi there"
                           (current-continuation-marks)))))
--------------------
FAILURE
name:            check-exn
location:        eval:31:0
params:          '(#rx"Hello there" #<procedure>)
message:         "Wrong exception raised"
exn-message:     "Hi there"
exn:             #(struct:exn:fail "Hi there" #<continuation-mark-set>)
--------------------
```

The following example is a common mistake. The call to `error` is not within a `lambda`, so it bypasses `check-exn` entirely.

```
; Forgot to wrap the expression in a thunk. Don't do this!
> (check-exn exn:fail?
             (error 'hi "there"))
--------------------
ERROR
name:         check-exn
location:     eval:32:0

hi: there
--------------------
```

```
(check-not-exn thunk [message]) → void?
  thunk : (-> any)
  message : (or/c string? #f) = #f
```

Checks that *thunk* does not raise any exceptions. The optional *message* is included in the output if the check fails.

```
> (check-not-exn (λ () 1))
> (check-not-exn (λ () (car '())))
```

```
--------------------
FAILURE
name:                 check-not-exn
location:             eval:34:0
params:               '(#<procedure>)
message:              "Exception raised"
exception-message:    "car: contract violation\n  expected: pair?\n  given: '()"
exception:
  car: contract violation
    expected: pair?
    given: '()
--------------------
> (check-not-exn (λ () (/ 1 0)) "don't divide by 0")
--------------------
FAILURE
name:                 check-not-exn
location:             eval:35:0
params:               '(#<procedure>)
message:              "don't divide by 0"
exception-message:    "/: division by zero"
exception:
  /: division by zero
--------------------
```

```
(check-regexp-match regexp string) → void?
  regexp : (or/c regexp? byte-regexp? string? bytes?)
  string : (or/c string? bytes? path? input-port?)
```

Checks that *regexp* matches the *string*.

For example, the following check succeeds:

```
> (check-regexp-match "a+bba" "aaaaaabba")
```

The following check fails:

```
> (check-regexp-match "a+bba" "aaaabbba")
--------------------
FAILURE
name:          check-regexp-match
location:      eval:37:0
params:        '("a+bba" "aaaabbba")
--------------------
```

```
(check-match v pattern)
(check-match v pattern pred)
```

A check that pattern matches on the test value. Matches the test value *v* against *pattern* as a `match` clause. If no *pred* is provided, then if the match succeeds, the entire check succeeds. For example, this use succeeds:

```
> (check-match (list 1 2 3) (list _ _ 3))
```

This check fails to match:

```
> (check-match (list 1 2 3) (list _ _ 4))
--------------------
FAILURE
name:          check-match
location:    eval:39:0
actual:       '(1 2 3)
pattern:      (list _ _ 4)
--------------------
```

If *pred* is provided, it is evaluated with the bindings from the match pattern. If it produces `#t`, the entire check succeeds, otherwise it fails. For example, this use succeeds, binding `x` in the predicate:

```
> (check-match (list 1 (list 3)) (list x (list _)) (odd? x))
```

This check fails because the *pred* fails:

```
> (check-match 6 x (odd? x))
--------------------
FAILURE
name:          check-match
location:    eval:41:0
actual:       6
pattern:      x
condition:   (odd? x)
--------------------
```

This check fails because of a failure to match:

```
> (check-match (list 1 2) (list x) (odd? x))
--------------------
FAILURE
name:          check-match
location:    eval:42:0
actual:       '(1 2)
pattern:      (list x)
condition:   (odd? x)
--------------------
```

```
(check op v1 v2 [message]) → void?
  op : (-> any/c any/c any/c)
  v1 : any/c
  v2 : any/c
  message : (or/c string? #f) = #f
```

The most generic check. Succeeds if *op* applied to *v1* and *v2* is not `#f`, otherwise raises an exception of type `exn:test:check`. The optional *message* is included in the output if the check fails.

For example, the following check succeeds:

```
> (check < 2 3)
```

The following check fails:

```
> (check memq 'pine '(apple orange pear))
--------------------
FAILURE
name:          check
location:    eval:44:0
params:       '(#<procedure:memq> pine (apple orange pear))
--------------------
```

```
(fail [message]) → void?
  message : (or/c string? #f) = #f
```

This check fails unconditionally. Good for creating test stubs that you intend to fill out later. The optional *message* is included in the output.

3.2.2 Augmenting Information on Check Failure

When a check fails, it may add information about the failure to RackUnit's check-info stack. Additional information can be stored by using the `with-check-info*` function, and the `with-check-info` macro.

```
(struct check-info (name value)
    #:extra-constructor-name make-check-info
    #:transparent)
  name : symbol?
  value : any/c
```

A *check-info structure* stores information associated with the context of the execution of a check. The `value` is normally written in a check failure message using `write`, but the `rackunit` library provides several special formatting wrappers that can influence how the check info value is printed.

Changed in version 1.6 of package `rackunit-lib`: Changed from opaque to transparent

```
(struct string-info (value)
    #:transparent)
  value : string?
```

A special wrapper around a string for use as a check-info value. When displayed in a check failure message, `value` is displayed without quotes. Used to print messages instead of writing values.

```
> (define-check (string-info-check)
    (with-check-info (['value "hello world"]
                      ['message (string-info "hello world")])
      (fail-check)))
> (string-info-check)
--------------------
FAILURE
name:          string-info-check
location:    eval:46:0
params:        '()
value:         "hello world"
message:       hello world
--------------------
```

Added in version 1.2 of package `rackunit-lib`.

```
(struct nested-info (values)
    #:transparent)
  values : (listof check-info?)
```

A special wrapper around a list of check-infos for use as a `check-info` value. A check info whose value is a nested info is displayed as an indented subsequence of infos. Nested infos can be placed inside nested infos, yielding greater indentation.

```
> (define-check (nested-info-check)
    (define infos
      (list (make-check-info 'foo "foo") (make-check-
info 'bar "bar")))
    (with-check-info (['nested (nested-info infos)]) (fail-
check)))
> (nested-info-check)
```

```
--------------------
FAILURE
name:          nested-info-check
location:      eval:48:0
params:        '()
nested:
  foo:             "foo"
  bar:             "bar"
--------------------
```

Added in version 1.7 of package `rackunit-lib`.

```
(struct dynamic-info (proc)
    #:transparent)
  proc : (-> any/c)
```

A special wrapper around a procedure that produces a value for a check-info. When a `dynamic-info` is displayed in a check info stack, `proc` is called to determine what value to display.

```
> (with-check-info (['current-dir (dynamic-info current-
directory)])
    (check-equal? 1 2)
    (parameterize ([current-directory (find-system-path 'temp-
dir)])
      (check-equal? 1 2)))
--------------------
FAILURE
current-dir:
  #<path:/Users/robby/snapshot/racket/build/docs/share/pkgs/rackunit-
doc/rackunit/>
name:              check-equal?
location:      eval:49:0
actual:         1
expected:       2
--------------------
--------------------
FAILURE
current-dir:   #<path:/var/folders/7y/jrt5816d25l9tflxsrp3w_6c0000gn/T/>
name:              check-equal?
location:      eval:49:0
actual:         1
expected:       2
--------------------
```

The value returned by `proc` may itself be a special formatting value such as `nested-info`

(or even another dynamic-info), in which case that value is rendered as it would be if it had not been wrapped in dynamic-info.

```
> (define current-foo (make-parameter #f))
> (with-check-info (['foo (dynamic-info current-foo)])
    (check-equal? 1 2)
    (parameterize ([current-foo
                    (nested-info (list (make-check-
info 'nested 'foo)))])
      (check-equal? 1 2)))
--------------------
FAILURE
foo:           #f
name:          check-equal?
location:    eval:51:0
actual:        1
expected:      2
--------------------
--------------------
FAILURE
foo:
  nested:        foo
name:          check-equal?
location:    eval:51:0
actual:        1
expected:      2
--------------------
```

Added in version 1.9 of package rackunit-lib.

The are several predefined functions that create check-info structures with predefined names. This avoids misspelling errors:

```
(make-check-name name) → check-info?
  name : string?
(make-check-params params) → check-info?
  params : (listof any/c)
(make-check-location loc) → check-info?
  loc : (list/c any/c (or/c number? #f) (or/c number? #f)
                      (or/c number? #f) (or/c number? #f))
(make-check-expression msg) → check-info?
  msg : any/c
(make-check-message msg) → check-info?
  msg : string?
(make-check-actual param) → check-info?
  param : any/c
```

```
(make-check-expected param) → check-info?
  param : any/c
```

```
(with-check-info* info thunk) → any
  info : (listof check-info?)
  thunk : (-> any)
```

Pushes the given *info* on the check-info stack for the duration (the dynamic extent) of the execution of *thunk*

```
> (with-check-info*
    (list (make-check-info 'time (current-seconds)))
    (lambda () (check = 1 2)))
--------------------
FAILURE
time:        1793758231
name:        check
location:    eval:52:0
params:      '(#<procedure:=> 1 2)
--------------------
```

When this check fails the message

`time: <current-seconds-at-time-of-running-check>`

is printed along with the usual information on an check failure.

```
(with-check-info ((name val) ...) body ...)
```

The `with-check-info` macro pushes the given information onto the check-info stack for the duration of the execution of the body expressions. Each *name* must be a quoted symbol and each *val* must be a value.

```
> (for-each
    (lambda (elt)
      (with-check-info
       (('current-element elt))
       (check-pred odd? elt)))
    (list 1 3 5 7 8))
--------------------
FAILURE
current-element:   8
name:              check-pred
location:          eval:53:0
params:            '(#<procedure:odd?> 8)
--------------------
```

When this test fails the message

```
current-element: 8
```

is displayed along with the usual information on an check failure.

```
(with-default-check-info* info thunk) → any
  info : (listof check-info?)
  thunk : (-> any)
```

Similar to `with-check-info*`, but ignores elements of *info* whose name (as determined by `check-info-name`) matches the name of an element on the current check-info stack.

```
> (with-default-check-info*
    (list (make-check-name 'first-name))
    (λ ()
      (with-default-check-info*
        (list (make-check-name 'last-name))
        (λ ()
          (check-true #false)))))
--------------------
FAILURE
name:          first-name
location:    eval:54:0
params:        '(#f)
--------------------
```

The error message above should include `'first-name` but not `'last-name`.

3.2.3 Custom Checks

Custom checks can be defined using `define-check` and its variants. To effectively use these macros it is useful to understand a few details about a check's evaluation model.

First, a check should be considered a function, even though most uses are actually macros. In particular, checks always evaluate their arguments exactly once before executing any expressions in the body of the checks. Hence if you wish to write checks that evaluate user defined code that code must be wrapped in a thunk (a function of no arguments) by the user. The predefined `check-exn` is an example of this type of check.

Second, checks add information to the *check-info stack*: an internal list of check-info structures that RackUnit interprets to build error messages. The basic checks treat the stack as a source of optional arguments; if the stack is missing some information, then the check may supply a default value. For example, `check-equal?` adds a default source location if

the check-info stack does not contain a check-info with the name `'location` (see `make-check-location`).

```
(define-simple-check (name param ...) body ...)
```

The `define-simple-check` macro constructs a check called *name* that takes the params and an optional message as arguments and evaluates the *body*s. The check fails if the result of the last *body* is `#f`. Otherwise the check succeeds.

Simple checks cannot report extra information by using `with-check-info` inside their *body*.

For example, the following code defines a check `check-odd?`

```
> (define-simple-check (check-odd? number)
    (odd? number))
```

We can use these checks in the usual way:

```
> (check-odd? 3)
> (check-odd? 2)
--------------------
FAILURE
name:          check-odd?
location:    eval:57:0
params:        '(2)
--------------------
```

```
(define-binary-check (name pred actual expected))
(define-binary-check (name actual expected) body ...)
```

The `define-binary-check` macro constructs a check that tests a binary predicate. It adds the values of *actual* and *expected* to the check-info stack. The first form of `define-binary-check` accepts a binary predicate and tests if the predicate holds for the given values. The second form tests if the last *body* evaluates to a non-false value.

Here's the first form, where we use a predefined predicate to construct a binary check:

```
> (define-binary-check (check-char=? char=? actual expected))
```

In use:

```
> (check-char=? (read-char (open-input-string "a")) #\a)
```

If the expression is more complicated, the second form should be used. For example, below we define a binary check that tests whether a number is within 0.01 of the expected value:

```
> (define-binary-check (check-in-tolerance actual expected)
    (< (abs (- actual expected)) 0.01))
```

```
(define-check (name param ...) body ...)
```

The `define-check` macro is similar to `define-simple-check`, except the check only fails if `fail-check` is called in the body of the check. This allows more flexible checks, and in particular more flexible reporting options.

```
> (define-check (check-even? number)
    (unless (even? number)
      (fail-check)))
> (check-even? 0)
> (check-even? 1)
--------------------
FAILURE
name:       check-even?
location:   eval:63:0
params:     '(1)
--------------------
```

Checks defined with `define-check` add the source location and source syntax at their use-site to the check-info stack, unless the stack already contains values for the keys `'location` and `'expression`.

```
> (check-equal? 0 1)
--------------------
FAILURE
name:       check-equal?
location:   eval:64:0
actual:     0
expected:   1
--------------------
> (with-check-info*
    (list (make-check-location (list 'custom 6 1 #f #f)))
    (λ () (check-equal? 0 1)))
--------------------
FAILURE
location:   custom:6:1
name:       check-equal?
actual:     0
expected:   1
--------------------
```

Changed in version 1.9 of package `rackunit-lib`: Documented the protocol for adding `'location` and `'expression` information.

```
(fail-check message) → void?
  message : string?
```

Raises an `exn:test:check` with the contents of the check-info stack. The optional message is used as the exception's message.

3.3 Compound Testing Forms

3.3.1 Test Cases

As programs increase in complexity the unit of testing grows beyond a single check. For example, it may be the case that if one check fails it doesn't make sense to run another. To solve this problem compound testing forms can be used to group expressions. If any expression in a group fails (by raising an exception) the remaining expressions will not be evaluated.

```
(test-begin expr ...)
```

A `test-begin` form groups the *expr*s into a single unit. If any *expr* fails the following ones are not evaluated.

For example, in the following code the world is not destroyed as the preceding check fails:

```
(test-begin
  (check-eq? 'a 'b)
  ; This line won't be run
  (destroy-the-world))
```

```
(test-case name body ...+)
```

Like a `test-begin` except a name is associated with the *body*s. The name will be reported if the test fails.

Here's the above example rewritten to use `test-case` so the test can be named.

```
(test-case
  "Example test"
  (check-eq? 'a 'b)
  ; This line won't be run
  (destroy-the-world))
```

```
(test-case? obj) → boolean?
  obj : any/c
```

True if *obj* is a test case, and false otherwise.

Shortcuts for Defining Test Cases

```
(test-check name operator v1 v2) → void?
  name : string?
  operator : (-> any/c any/c any/c)
  v1 : any/c
  v2 : any/c
(test-pred name pred v) → void?
  name : string?
  pred : (-> any/c any/c)
  v : any/c
(test-equal? name v1 v2) → void?
  name : string?
  v1 : any/c
  v2 : any/c
(test-eq? name v1 v2) → void?
  name : string?
  v1 : any/c
  v2 : any/c
(test-eqv? name v1 v2) → void?
  name : string?
  v1 : any/c
  v2 : any/c
(test-= name v1 v2 epsilon) → void?
  name : string?
  v1 : real?
  v2 : real?
  epsilon : real?
(test-true name v) → void?
  name : string?
  v : any/c
(test-false name v) → void?
  name : string?
  v : any/c
(test-not-false name v) → void?
  name : string?
  v : any/c
(test-exn name pred thunk) → void?
  name : string?
  pred : (or/c (-> any/c any/c) regexp?)
  thunk : (-> any)
(test-not-exn name thunk) → void?
  name : string?
  thunk : (-> any)
```

Creates a test case with the given *name* that performs the corresponding check. For example,

```
(test-equal? "Fruit test" "apple" "pear")
```

is equivalent to

```
(test-case "Fruit test" (check-equal? "apple" "pear"))
```

3.3.2 Test Suites

Test cases can themselves be grouped into test suites. A test suite can contain both test cases and test suites. Unlike a check or test case, a test suite is not immediately run. Instead use one of the functions described in §3.6 “User Interfaces” or §5.3 “Programmatically Running Tests and Inspecting Results”.

```
(test-suite name-expr maybe-before maybe-after test ...)

maybe-before =
             | #:before before-thunk

 maybe-after =
             | #:after after-thunk

  name-expr : string?
```

Constructs a test suite with the given name and tests. The tests may be checks, test cases, constructed using `test-begin` or `test-case`, or other test suites.

The *before-thunk* and *after-thunk* are optional thunks (functions with no argument). They are run before and after the tests are run, respectively.

Unlike a check or test case, a test suite is not immediately run. Instead use one of the functions described in §3.6 “User Interfaces” or §5.3 “Programmatically Running Tests and Inspecting Results”.

For example, here is a test suite that displays `Before` before any tests are run, and `After` when the tests have finished.

```
(test-suite
  "An example suite"
  #:before (lambda () (display "Before"))
  #:after  (lambda () (display "After"))
  (test-case
    "An example test"
    (check-eq? 1 1))
```

```
(test-suite "A nested test suite"
  (test-case "Another test"
    (check < 1 2))))
```

```
(make-test-suite name
                 tests
                [#:before before-thunk
                 #:after after-thunk]) → test-suite?
  name : string?
  tests : (listof (or/c test-case? test-suite?))
  before-thunk : (-> any) = void
  after-thunk : (-> any) = void
```

Constructs a test suite with the given *name* containing the given *tests*. Unlike the `test-suite` form, the tests are represented as a list of test values.

```
(test-suite? obj) → boolean?
  obj : any/c
```

True if *obj* is a test suite, and false otherwise

Utilities for Defining Test Suites

There are some macros that simplify the common cases of defining test suites:

```
(define-test-suite name test ...)
```

The `define-test-suite` form creates a test suite with the given name (converted to a string) and tests, and binds it to the same name.

For example, this code creates a binding for the name `example-suite` as well as creating a test suite with the name `"example-suite"`:

```
(define-test-suite example-suite
  (check = 1 1))
```

```
(define/provide-test-suite name test ...)
```

This form is just like `define-test-suite`, and in addition it `provide`s the test suite.

3.4 Test Control Flow

The `before`, `after`, and `around` macros allow you to specify code that is always run before, after, or around expressions in a test case.

```
(before before-expr expr-1 expr-2 ...)
```

Whenever control enters the scope execute the *before-expr* before executing *expr-1*, and *expr-2* ...

```
(after expr-1 expr-2 ... after-expr)
```

Whenever control exits the scope execute the *after-expr* after executing *expr-1*, and *expr-2* ... The *after-expr* is executed even if control exits via an exception or other means.

```
(around before-expr expr-1 expr-2 ... after-expr)
```

Whenever control enters the scope execute the *before-expr* before executing *expr-1 expr-2* ..., and execute *after-expr* whenever control leaves the scope.

Example:

The test below checks that the file `test.dat` contains the string `"foo"`. The before action writes to this file. The after action deletes it.

```
(around
  (with-output-to-file "test.dat"
     (lambda ()
       (write "foo")))
  (with-input-from-file "test.dat"
    (lambda ()
      (check-equal? "foo" (read))))
  (delete-file "test.dat"))
```

```
(delay-test test1 test2 ...)
```

This somewhat curious macro evaluates the given tests in a context where `current-test-case-around` is parameterized to `test-suite-test-case-around`. This has been useful in testing RackUnit. It might be useful for you if you create test cases that create test cases.

3.5 Miscellaneous Utilities

The `require/expose` macro allows you to access bindings that a module does not provide. It is useful for testing the private functions of modules.

```
(require/expose module (id ...))
```

Requires *id* from *module* into the current module. It doesn't matter if the source module provides the bindings or not; `require/expose` can still get at them.

Note that `require/expose` can be a bit fragile, especially when mixed with compiled code. Use at your own risk!

This example gets `make-failure-test`, which is defined in a RackUnit test:

```
(require/expose rackunit/private/check-test (make-failure-test))
```

```
(dynamic-require/expose mod name) → any
  mod : (or/c module-path?
              module-path-index?
              resolved-module-path?)
  name : symbol?
```

Like `dynamic-require`, but gets internal bindings like `require/expose`.

Checks defined with `define-check` provide a compile-time API to access information associated with the check.

```
(check-transformer? v) → boolean?
  v : any/c
```

Determines if *v* is a syntax transformer defined with `define-check`. Typically, this is used on the result of `syntax-local-value`.

Provided by `rackunit` at phase 1.

```
(check-transformer-impl-name ct) → identifier?
  ct : check-transformer?
```

Given a transformer *ct* defined with `define-check`, produces an identifier which names the procedure implementing the check. This procedure takes the same arguments as the check form, as well as two mandatory keyword arguments: `#:location` whose argument must be a list representing a source location as in the third argument of `datum->syntax`, and `#:exp`, whose argument is an s-expression representing the original syntax of the check for printing.

Provided by `rackunit` at phase 1.

3.6 User Interfaces

RackUnit provides a textual and a graphical user interface

3.6.1 Textual User Interface

```
(require rackunit/text-ui)
```
package: rackunit-lib

The textual UI is in the rackunit/text-ui module. It is run via the run-tests function.

```
(run-tests test [verbosity]) → natural-number/c
  test : (or/c test-case? test-suite?)
  verbosity : (symbols 'quiet 'normal 'verbose) = 'normal
```

The given *test* is run and the result of running it output to the current-output-port if all tests pass, and to current-error-port when there are test failures. The output is compatible with the (X)Emacs next-error command (as used, for example, by (X)Emacs's compile function).

The optional *verbosity* is one of 'quiet, 'normal, or 'verbose. Quiet output displays only the number of successes, failures, and errors. Normal reporting suppresses some extraneous check information (such as the expression). Verbose reports all information.

run-tests returns the number of unsuccessful tests.

3.6.2 Graphical User Interface

```
(require rackunit/gui)
```
package: rackunit-gui

RackUnit also provides a GUI test runner, available from the rackunit/gui module.

```
(test/gui test ... [#:wait? wait?]) → void?
  test : (or/c test-case? test-suite?)
  wait? : boolean? = #f
```

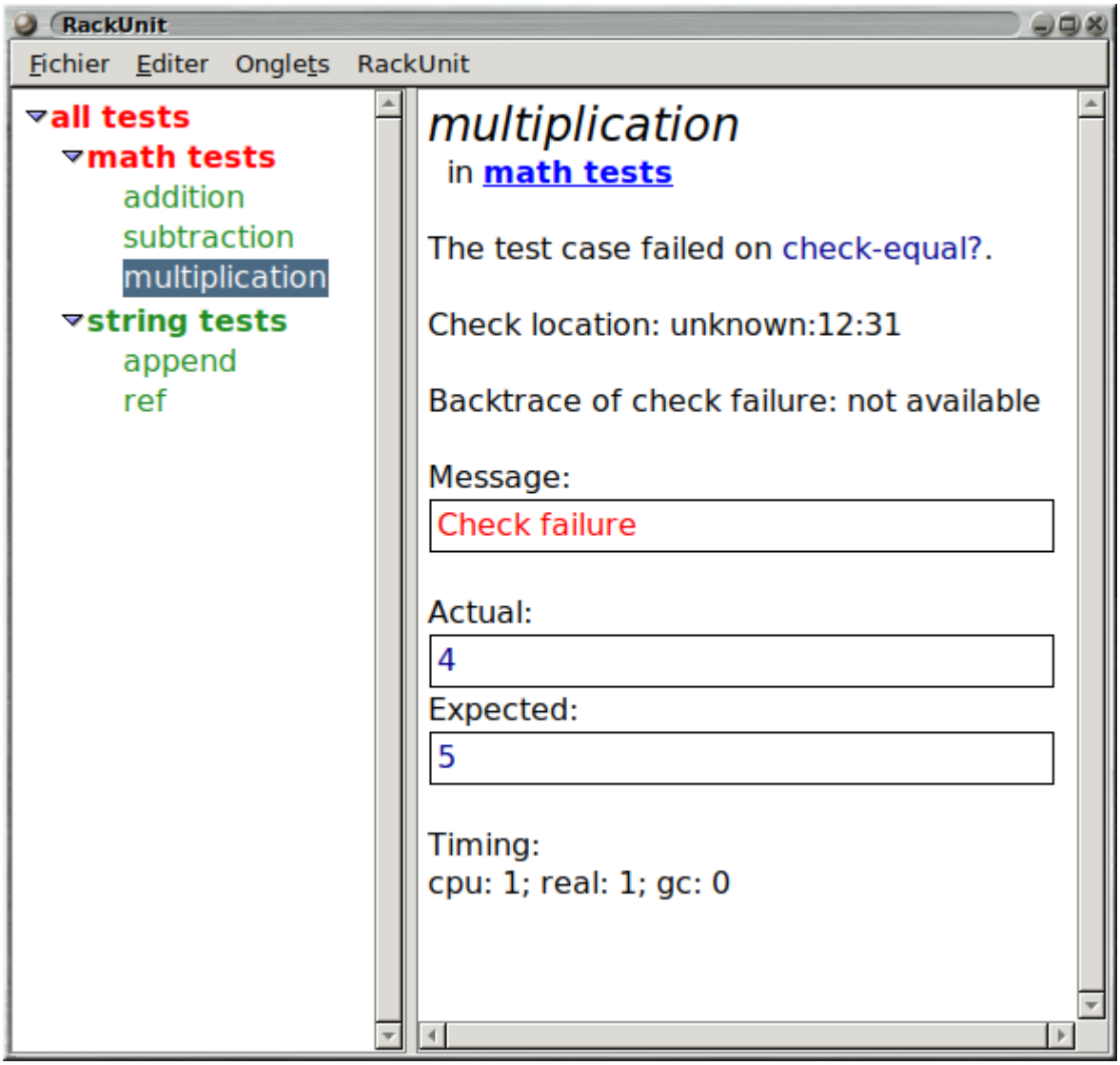
Creates a new RackUnit GUI window and runs each *test*. The GUI is updated as tests complete.

When *wait?* is true, test/gui does not return until the test runner window has been closed.

Given the following program, the RackUnit GUI will look as shown below:

```
#lang racket
(require rackunit rackunit/gui)
(test/gui
 (test-suite
  "all tests"
  (test-suite
   "math tests"
```

```
   (test-case "addition" (check-equal? (+ 1 1) 2))
   (test-case "subtraction" (check-equal? (- 0 0) 0))
   (test-case "multiplication" (check-equal? (* 2 2) 5)))
  (test-suite
   "string tests"
   (test-case "append" (check-equal? (string-
append "a" "b") "ab"))
   (test-case "ref" (check-equal? (string-ref "abc" 1) #\b)))))
```




```
(make-gui-runner) → (-> (or/c test-case? test-suite?) ... any)
```

Creates a new RackUnit GUI window and returns a procedure that, when applied, runs the given tests and displays the results in the GUI.

4 Testing Utilities

4.1 Checking documentation completeness

```
(require rackunit/docs-complete)
```
package: racket-index

```
(check-docs lib [#:skip skip]) → any
  lib : module-path?
  skip : (or/c regexp?                          = #f
               symbol?
               (listof (or/c regexp? symbol?))
               (-> symbol? any)
               #f)
```

Checks to see if the module path named by *lib* (e.g. `'racket/list`) has documented all of its exports and prints an error message to `(current-error-port)` if not.

If *skip* is a regexp, then exporting matching that regexp are ignored. If it is a symbol, then that export is ignored. If it is a list of symbols and regexps, then any exporting matching any of the symbols or regexps are ignored. If it is a function, the function is treated as a predicate and passed each export of the module. If *skip* is `#f`, no exports are skipped.

Changed in version 1.10 of package `racket-index`: Changed *lib* to accept any module path.

4.2 Logging Test Results

```
(require rackunit/log)
```
package: testing-util-lib

NOTE: This library is deprecated; use `raco/testing`, instead.

```
(test-log! result) → void?
  result : any/c
```

Re-exports `test-log!` from `raco/testing`.

```
(test-log [#:display? display?
           #:exit? exit?])
 → (cons/c exact-nonnegative-integer?
           exact-nonnegative-integer?)
  display? : any/c = #f
  exit? : any/c = #f
```

Re-exports `test-report` from `raco/testing`.

Changed in version 1.11 of package `testing-util-lib`: Allow any value for the *`display?`* and *`exit?`* arguments, not just booleans.

```
(test-log-enabled?) → boolean?
(test-log-enabled? enabled?) → void?
  enabled? : any/c
 = #t
```

Re-exports `test-log-enabled?` from `raco/testing`.

Added in version 1.1 of package `testing-util-lib`.
Changed in version 1.11: Allow any value for the parameter and coerce it to a boolean.

```
(current-test-invocation-directory) → (or/c #f path-string?)
(current-test-invocation-directory test-invocation-directory)
 → void?
  test-invocation-directory : (or/c #f path-string?)
 = #f
```

Re-exports `current-test-invocation-directory` from `raco/testing`.

Added in version 1.2 of package `testing-util-lib`.

5 RackUnit Internals and Extension API

This section describes RackUnit's facilities for customizing the behavior of checks and tests and for creating new kinds of test runners.

5.1 Customizing Check Evaluation

The semantics of checks are determined by the parameters `current-check-around` and `current-check-handler`. Other testing form such as `test-begin` and `test-suite` change the value of these parameters.

```
(current-check-handler) → (-> any/c any)
(current-check-handler handler) → void?
  handler : (-> any/c any)
```

Parameter containing the function that handles exceptions raised by check failures. The default value is a procedure that will display the exception data in a user-friendly format.

```
(current-check-around) → (-> (-> any) any)
(current-check-around check) → void?
  check : (-> (-> any) any)
```

Parameter containing the function that handles the execution of checks. The default value wraps the evaluation of `thunk` in a `with-handlers` call that calls `current-check-handler` if an exception is raised and then (when an exception is not raised) discards the result, returning `(void)`.

5.2 Customizing Test Evaluation

Just like with checks, there are several parameters that control the semantics of compound testing forms.

```
(current-test-name) → (or/c string? false/c)
(current-test-name name) → void?
  name : (or/c string? false/c)
```

This parameter stores the name of the current test case. A value of `#f` indicates a test case with no name, such as one constructed by `test-begin`.

```
(current-test-case-around) → (-> (-> any) any)
(current-test-case-around handler) → void?
  handler : (-> (-> any) any)
```

This parameter handles evaluation of test cases. The value of the parameter is a function that is passed a thunk (a function of no arguments). The function, when applied, evaluates the expressions within a test case. The default value of the current-test-case-around parameters evaluates the thunk in a context that catches exceptions and prints an appropriate message indicating test case failure.

```
(test-suite-test-case-around thunk) → any
  thunk : (-> any)
```

The current-test-case-around parameter is parameterized to this value within the scope of a test-suite. This function creates a test case structure instead of immediately evaluating the thunk.

```
(test-suite-check-around thunk) → any/c
  thunk : (-> any/c)
```

The current-check-around parameter is parameterized to this value within the scope of a test-suite. This function creates a test case structure instead of immediately evaluating a check.

5.3 Programmatically Running Tests and Inspecting Results

RackUnit provides an API for running tests, from which custom UIs can be created.

5.3.1 Result Types

```
(struct exn:test exn:fail ()
    #:extra-constructor-name make-exn:test)
```

The base structure for RackUnit exceptions. You should never catch instances of this type, only the subtypes documented below.

```
(struct exn:test:check exn:test (stack)
    #:extra-constructor-name make-exn:test:check)
  stack : (listof check-info)
```

A exn:test:check is raised when an check fails, and contains the contents of the check-info stack at the time of failure.

```
(struct test-result (test-case-name)
    #:extra-constructor-name make-test-result)
  test-case-name : (or/c string #f)
```

A test-result is the result of running the test with the given name (with `#f` indicating no name is available).

```
(struct test-failure test-result (result)
    #:extra-constructor-name make-test-failure)
  result : any
```

Subtype of test-result representing a test failure.

```
(struct test-error test-result (result)
    #:extra-constructor-name make-test-error)
  result : exn
```

Subtype of test-result representing a test error.

```
(struct test-success test-result (result)
    #:extra-constructor-name make-test-success)
  result : any
```

Subtype of test-result representing a test success.

5.3.2 Functions to Run Tests

```
(run-test-case name action) → test-result
  name : (or/c string #f)
  action : (-> any)
```

Runs the given test case, returning a result representing success, failure, or error.

```
(run-test test)
 → (flat-murec-contract ([R (listof (or/c test-result? R))]) R)
  test : (or/c test-case? test-suite?)
```

Runs the given test (test case or test suite) returning a tree (list of lists) of results

Example:

```
(run-test
   (test-suite
    "Dummy"
    (test-case "Dummy" (check-equal? 1 2))))
```

```
(fold-test-results result-fn
                   seed
                   test
                   #:run run
                   #:fdown fdown
                   #:fup fup)    → 'a
  result-fn : ('b 'c ... 'a . -> . 'a)
  seed : 'a
  test : (or/c test-case? test-suite?)
  run : (string (() -> any) . -> . 'b 'c ...)
  fdown : (string 'a . -> . 'a)
  fup : (string 'a . -> . 'a)
```

Fold *result-fn* pre-order left-to-right depth-first over the results of *run*. By default *run* is run-test-case and *fdown* and *fup* just return the seed, so *result-fn* is folded over the test results.

This function is useful for writing custom folds (and hence UIs) over test results without you having to take care of all the expected setup and teardown. For example, fold-test-results will run test suite before and after actions for you. However it is still flexible enough, via its keyword arguments, to do almost anything that foldts-test-suite can. Hence it should be used in preference to foldts-test-suite.

The *result-fn* argument is a function from the results of *run* (defaults to a test-result) and the seed to a new seed.

The *seed* argument is any value.

The *test* argument is a test case or test suite.

The *run* argument is a function from a test case name (string) and action (thunk) to any values. The values produced by *run* are fed into the *result-fn*.

The *fdown* argument is a function from a test suite name (string) and the seed, to a new seed.

The *fup* argument is a function from a test suite name (string) and the seed, to a new seed.

Examples:

The following code counts the number of successes:

```
(define (count-successes test)
  (fold-test-results
   (lambda (result seed)
     (if (test-success? result)
         (add1 seed)
```

```
        seed))
  0
  test))
```

The following code returns the symbol `'burp` instead of running test cases. Note how the `result-fn` receives the value of `run`.

```
(define (burp test)
  (fold-test-results
   (lambda (result seed) (cons result seed))
   null
   test
   #:run (lambda (name action) 'burp)))
```

```
(foldts-test-suite fdown fup fhere seed test) → 'a
  fdown : (test-suite string thunk thunk 'a -> 'a)
  fup : (test-suite string thunk thunk 'a 'a -> 'a)
  fhere : (test-case string thunk 'a -> 'a)
  seed : 'a
  test : (or/c test-case? test-suite?)
```

The `foldts-test-suite` function is a nifty tree fold (created by Oleg Kiselyov) that folds over a test in a useful way (`fold-test-results` isn't that useful as you can't specify actions around test cases).

The *`fdown`* argument is a function of test suite, test suite name, before action, after action, and the seed. It is run when a test suite is encountered on the way down the tree (pre-order).

The *`fup`* argument is a function of test suite, test suite name, before action, after action, the seed at the current level, and the seed returned by the children. It is run on the way up the tree (post-order).

The *`fhere`* argument is a function of the test case, test case name, the test case action, and the seed. (Note that this might change in the near future to just the test case. This change would be to allow *`fhere`* to discriminate subtypes of test-case, which in turn would allow test cases that are, for example, ignored).

Example:

Here's the implementation of `fold-test-results` in terms of `foldts-test-suite`:

```
(define (fold-test-results suite-fn case-fn seed test)
  (foldts-test-suite
   (lambda (suite name before after seed)
     (before)
     (suite-fn name seed))
```

```
(lambda (suite name before after seed kid-seed)
  (after)
  kid-seed)
(lambda (case name action seed)
  (case-fn
    (run-test-case name action)
    seed))
seed
test))
```

If you're used to folds you'll probably be a bit surprised that the functions you pass to `foldts-test-suite` receive both the structure they operate on, and the contents of that structure. This is indeed unusual. It is done to allow subtypes of test-case and test-suite to be run in customised ways. For example, you might define subtypes of test case that are ignored (not run), or have their execution time recorded, and so on. To do so the functions that run the test cases need to know what type the test case has, and hence is is necessary to provide this information.

6 Release Notes

6.1 Version 3.4

This version allows arbitrary expressions within test suites, fixing the semantics issue below.

There are also miscellaneous Scribble fixes.

6.2 Version 3

This version of RackUnit is largely backwards compatible with version 2 but there are significant changes to the underlying model, justifying incrementing the major version number. These changes are best explained in §2 “The Philosophy of RackUnit”.

There are a few omissions in this release, that will hopefully be corrected in later minor version releases:

- There is no graphical UI, and in particular no integration with DrRacket.
- The semantics of `test-suite` are not the desired ones. In particular, only checks and test cases have their evaluation delayed by a test suite; other expressions will be evaluated before the suite is constructed. This won’t affect tests written in the version 2 style. In particular this doesn’t effect test suites that contain other test suites; they continue to work in the expected way. However people incrementally developing tests from plain checks to test suites might be surprised. I’m hoping that few enough people will do this that no-one will notice before it’s fixed.

7 Acknowlegements

The following people have contributed to RackUnit:

- Robby Findler pushed me to release version 3
- Matt Jadud and his students at Olin College suggested renaming `test/text-ui`
- Dave Gurnell reported a bug in check-not-exn and suggested improvements to RackUnit
- Danny Yoo reported a bug in and provided a fix for trim-current-directory
- Jacob Matthews and Guillaume Marceau for bug reports and fixes
- Eric Hanchrow suggested test/text-ui return a useful result
- Ray Racine and Richard Cobbe provided require/expose
- John Clements suggested several new checks
- Jose A. Ortega Ruiz alerted me a problem in the packaging system and helped fix it.
- Sebastian H. Seidel provided help packaging RackUnit into a .plt
- Don Blaheta provided the method for grabbing line number and file name in checks
- Patrick Logan ported example.rkt to version 1.3
- The PLT team made Racket
- The Extreme Programming community started the whole testing framework thing

Index

Acknowlegements
after
around
Augmenting Information on Check Failure
Basic Checks
before
check
check-=
check-docs
check-eq?
check-equal?
check-eqv?
check-exn
check-false
check-info (struct)
check-info stack
check-info structure
check-info-name
check-info-value
check-info?
check-match
check-not-eq?
check-not-equal?
check-not-eqv?
check-not-exn
check-not-false
check-pred
check-regexp-match
check-transformer-impl-name
check-transformer?
check-true
check-within
Checking documentation completeness
Checks
Compound Testing Forms
current-check-around
current-check-handler
current-test-case-around
current-test-invocation-directory
current-test-name
Custom Checks
Customizing Check Evaluation
Customizing Test Evaluation
define-binary-check
define-check
define-simple-check
define-test-suite
define/provide-test-suite
delay-test
dynamic-info (struct)
dynamic-info-proc
dynamic-info?
dynamic-require/expose
exn:test (struct)
exn:test:check (struct)
exn:test:check-stack
exn:test:check?
exn:test?
fail
fail-check
fold-test-results
foldts-test-suite
Functions to Run Tests
Graphical User Interface
Historical Context
Logging Test Results
make-check-actual
make-check-expected
make-check-expression
make-check-info
make-check-location
make-check-message
make-check-name
make-check-params
make-exn:test
make-exn:test:check
make-gui-runner
make-test-error
make-test-failure
make-test-result
make-test-success
make-test-suite

Miscellaneous Utilities
nested-info (struct)
nested-info-values
nested-info?
Overview of RackUnit
Programmatically Running Tests and Inspecting Results
Quick Start Guide for RackUnit
rackunit
RackUnit API
RackUnit Internals and Extension API
rackunit/docs-complete
rackunit/gui
rackunit/log
rackunit/text-ui
RackUnit: Unit Testing
Release Notes
require/expose
Result Types
run-test
run-test-case
run-tests
Shortcuts for Defining Test Cases
string-info (struct)
string-info-value
string-info?
struct:check-info
struct:dynamic-info
struct:exn:test
struct:exn:test:check
struct:nested-info
struct:string-info
struct:test-error
struct:test-failure
struct:test-result
struct:test-success
Test Cases
Test Control Flow
Test Suites
test-=
test-begin
test-case
test-case?
test-check
test-eq?
test-equal?
test-eqv?
test-error (struct)
test-error-result
test-error?
test-exn
test-failure (struct)
test-failure-result
test-failure?
test-false
test-log
test-log!
test-log-enabled?
test-not-exn
test-not-false
test-pred
test-result (struct)
test-result-test-case-name
test-result?
test-success (struct)
test-success-result
test-success?
test-suite
test-suite-check-around
test-suite-test-case-around
test-suite?
test-true
test/gui
Testing Utilities
Textual User Interface
The Philosophy of RackUnit
User Interfaces
Utilities for Defining Test Suites
Version 3
Version 3.4
with-check-info
with-check-info*
with-default-check-info*