RackUnit: Unit Testing for Racket

Version 5.0

Noel Welsh <noelwelsh@gmail.com>
and Ryan Culpepper <ryan_sml@yahoo.com>

June 6, 2010

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

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:

```
#t
----------
FAILURE
name:       check-equal?
location:   (file-test.rkt 7 0 117 27)
expression: (check-equal? (my-* 1 2) 2)
params:     (4 2)
actual:     4
expected:   2
```

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

The first `#t` indicates the first test passed. 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)))
```

Evalute 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 if 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 are 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 unstructed 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 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 Racket 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)
```

3.1 Overview of RackUnit

There are three basic data types in RackUnit:

- A *check* is the basic unit of a test. As the name suggests, it checks 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. If the condition holds the check evaluates to `#t`. If the condition doesn't hold the check raises an instance of `exn:test:check` with information detailing the failure.

Although checks are implemented as macros, which is necessary to grab source location, they are conceptually functions. This means, for instance, checks always evaluate their arguments. You can use check as first class functions, though you will lose precision in the reported source locations if you do so.

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

```
(check op v1 v2 [message]) → any
  op : (-> any any any)
  v1 : any
  v2 : any
  message : string? = ""
```

The simplest 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. If the check succeeds, the value returned by *op* is the value returned by the check.

For example, the following check succeeds:

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

```
(check-eq? v1 v2 [message]) → #t
  v1 : any
  v2 : any
  message : string? = ""
(check-not-eq? v1 v2 [message]) → #t
  v1 : any
  v2 : any
  message : string? = ""
(check-eqv? v1 v2 [message]) → #t
  v1 : any
  v2 : any
  message : string? = ""
(check-not-eqv? v1 v2 [message]) → #t
  v1 : any
  v2 : any
  message : string? = ""
(check-equal? v1 v2 [message]) → #t
  v1 : any
  v2 : any
  message : string? = ""
(check-not-equal? v1 v2 [message]) → #t
  v1 : any
  v2 : any
  message : string? = ""
```

Checks that *v1* is (not) `eq?`, `eqv?`, or `equal?` to *v2*. 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?")
(check-not-eq? 1 1 "integers are eq?")
(check-eqv? 1 1.0 "not eqv?")
(check-not-eqv? 1 1 "integers are eqv?")
(check-equal? 1 1.0 "not equal?")
(check-not-equal? (list 1) (list 1) "equal?")
```

```
(check-pred pred v [message]) → #t
  pred : (-> any any)
  v : any
  message : string? = ""
```

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*.

Here's an example that passes and an example that fails:

```
(check-pred string? "I work")
(check-pred number? "I fail")
```

```
(check-= v1 v2 epsilon [message]) → #t
  v1 : any
  v2 : any
  epsilon : number?
  message : string? = ""
```

Checks that *v1* and *v2* are within *epsilon* of one another. The optional *message* is included in the output if the check fails.

Here's an example that passes and an example that fails:

```
(check-= 1.0 1.01 0.01 "I work")
(check-= 1.0 1.01 0.005 "I fail")
```

```
(check-true v [message]) → #t
  v : any
  message : string? = ""
(check-false v [message]) → #t
  v : any
  message : string? = ""
(check-not-false v [message]) → #t
  v : any
  message : string? = ""
```

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

For example, the following checks all fail:

```
(check-true 1)
(check-false 1)
(check-not-false #f)
```

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

Checks that *thunk* raises an exception for which *exn-predicate* returns #t. 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.

Here are two example, one showing a test that succeeds, and one showing a common error:

```
(check-exn exn?
           (lambda ()
             (raise (make-exn "Hi there"
                              (current-continuation-marks)))))
; Forgot to wrap the expression in a thunk. Don't do this!
(check-exn exn?
           (raise (make-exn "Hi there"
                            (current-continuation-marks))))
```

```
(check-not-exn thunk [message]) → #t
  thunk : (-> any)
  message : string? = ""
```

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

```
(fail [message]) → #t
  message : string? = ""
```

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

```
(check-regexp-match regexp string) → #t
  regexp : regexp?
  string : string?
```

Checks that *regexp* matches the *string*.

The following check will succeed:

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

This check will fail:

```
(check-regexp-match "a+bba" "aaaabbba")
```

3.2.1 Augmenting Information on Check Failure

When an check fails it stores information including the name of the check, the location and message (if available), the expression the check is called with, and the parameters to the check. 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)
  name : symbol?
  value : any
```

A check-info structure stores information associated with the context of execution of an check.

The are several predefined functions that create check information 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)
(make-check-location loc) → check-info?
  loc : (list/c any (or/c number? #f) (or/c number? #f)
                    (or/c number? #f) (or/c number? #f))
(make-check-expression msg) → check-info?
  msg : any
(make-check-message msg) → check-info?
  msg : string?
(make-check-actual param) → check-info?
  param : any
(make-check-expected param) → check-info?
  param : any
```

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

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

Example:

```
  (with-check-info*
```

```
(list (make-check-info 'time (current-seconds)))
(lambda () (check = 1 2)))
```

When this check fails the message

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

will be printed along with the usual information on an check failure.

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

The `with-check-info` macro stores the given information in the check information stack for the duration of the execution of the body expressions. `Name` is a quoted symbol and *val* is any value.

Example:

```
(for-each
 (lambda (elt)
   (with-check-info
    (('current-element elt))
    (check-pred odd? elt)))
 (list 1 3 5 7 8))
```

When this test fails the message

```
current-element: 8
```

will be displayed along with the usual information on an check failure.

3.2.2 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.

Firstly, 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 evalute 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.

It is also useful to understand how the check information stack operates. The stack is stored in a parameter and the `with-check-info` forms evaluate to calls to `parameterize`. Hence check information has lexical scope. For this reason simple checks (see below) cannot usefully contain calls to `with-check-info` to report additional information. All checks cre-

ated using `define-simple-check` or `define-check` grab some information by default: the name of the checks and the values of the parameters. Additionally the macro forms of checks grab location information and the expressions passed as parameters.

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

The `define-simple-check` macro constructs a check called *name* that takes the params and an optional message as arguments and evaluates the *expr*s. The check fails if the result of the *expr*s is `#f`. Otherwise the check succeeds. Note that simple checks cannot report extra information using `with-check-info`.

Example:

To define a check `check-odd?`

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

We can use these checks in the usual way:

```
(check-odd? 3)  ; Success
(check-odd? 2)  ; Failure
```

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

The `define-binary-check` macro constructs a check that tests a binary predicate. It's benefit over `define-simple-check` is in better reporting on check failure. The first form of the macro accepts a binary predicate and tests if the predicate holds for the given values. The second form tests if *expr* non-false.

Examples:

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 a-port) #\a)
```

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

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

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

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

```
(fail-check)
```

The `fail-check` macro raises an `exn:test:check` with the contents of the check information stack.

3.2.3 The Check Evaluation Context

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

Parameter containing the function that handles exceptions raised by check failures. The default value is `raise`.

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

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)`.

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 expr ...)`

Like a `test-begin` except a name is associated with the group of *expr*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

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

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 §3.7 "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 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 §3.7 "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

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 for is just like `define-test-suite`, and in addition it `provides` the test suite.

3.3.3 Compound Testing Evaluation Context

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/c) any/c)
(current-test-case-around *handler*) → void?
  handler : (-> (-> any/c) any/c)

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/c
  thunk : (-> any/c)

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.

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 expr1 expr2 ...*)

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))
```

3.6 User Interfaces

RackUnit provides a textual and a graphical user interface

3.6.1 Textual User Interface

```
(require rackunit/text-ui)
```

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. The output is compatable 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)
```

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

```
(test/gui test ...) → any
  test : (or/c test-case? test-suite?)
```

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

```
(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.

3.7 Programmatically Running Tests and Inspecting Results

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

3.7.1 Result Types

```
(struct exn:test exn ()
        #: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.

3.7.2 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) → (R = (listof (or/c test-result 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 can. Hence it should be used in preference to foldts.

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

to a new seed

Seed is any value

Test is a test-case or test-suite

Run is a function from a test case name (string) and action (thunk) to any values.

FDown is a function from a test suite name (string) and the seed, to a new seed

FUp 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 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?)
```

Foldts 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).

Fdown 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).

Fup 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).

Fhere 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:

```
(define (fold-test-results suite-fn case-fn seed test)
  (foldts
   (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 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.

If you've made it this far you truly are a master RackUnit hacker. As a bonus prize we'll just mention that the code in hash-monad.rkt and monad.rkt might be of interest for constructing user interfaces. The API is still in flux, so isn't documented here. However, do look at the implementation of `run-tests` for examples of use.

4 Release Notes

4.1 Version 3.4

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

There are also miscellaneous Scribble fixes.

4.2 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.

5 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 Hanchow 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