Test Support

Version 8.14.0.1

July 16, 2024

Contents

1 Using Check Forms

```
(require test-engine/racket-tests)        package: htdp-lib
```

This module provides test forms for use in Racket programs, as well as parameters to configure the behavior of test reports.

Each check form may only occur at the top-level; results are collected and reported by the test function. Note that the check forms only register checks to be performed. The checks are actually run by the test function.

```
(check-expect expr expected-expr)
```

Checks whether the value of the *expr* expression is `equal?` to the value produced by the *expected-expr*.

It is an error for *expr* or *expected-expr* to produce a function value or an inexact number.

```
(check-random expr expected-expr)
```

Checks whether the value of the *expr* expression is `equal?` to the value produced by the *expected-expr*.

The form supplies the same random-number generator to both parts. If both parts request `random` numbers from the same interval in the same order, they receive the same random numbers.

Examples:

```
> (check-random (random 10) (random 10))
> (check-random
    (begin (random 100) (random 200))
    (begin (random 100) (random 200)))
> (test)
Both tests passed!
```

If the two parts call `random` for different intervals, they are likely to fail:

Examples:

```
> (check-random
    (begin (random 100) (random 200))
    (begin (random 200) (random 100)))
> (test)
Ran 1 test.
```

```
0 tests passed.
Check failures:

         Actual value │ 16 │ differs from │ 8 │, the expected
value.


at line 2, column 0
```

It is an error for *expr* or *expected-expr* to produce a function value or an inexact number.

```
(check-satisfied expr property?)
```

Checks whether the value of the *expr* expression satisfies the *property?* predicate (which must evaluate to a function of one argument).

Examples:

```
> (check-satisfied 1 odd?)
> (check-satisfied 1 even?)
> (test)
Ran 2 tests.
1 of the 2 tests failed.
Check failures:

         Actual value │ 1 │ does not satisfy even?.

at line 3, column 0
```

Changed in version 1.1 of package `htdp-lib`: allow the above examples to run in BSL and BSL+

```
(check-within expr expected-expr delta-expr)

  delta-expr : number?
```

Checks whether the value of the `test` expression is structurally equal to the value produced by the `expected` expression; every number in the first expression must be within `delta` of the corresponding number in the second expression.

It is an error for *expr* or *expected* to produce a function value.

```
(check-error expr)
(check-error expr msg-expr)

  msg-expr : string?
```

4

Checks that evaluating *expr* signals an error, where the error message matches the string (if any).

```
(check-member-of expr expected-expr ...)
```

Checks whether the value of the *expr* expression is `equal?` to any of the values produced by the *expected-expr*s.

It is an error for *expr* or any of the *expected-expr*s to produce a function value or an inexact number.

```
(check-range expr min-expr max-expr)

  expr : number?
  min-expr : number?
  max-expr : number?
```

Checks whether value of *expr* is between the values of *min-expr* and *max-expr* inclusive.

```
(test)
```

Runs all of the tests specified by check forms in the current module and reports the results. When using the gui module, the results are provided in a separate window, otherwise the results are printed to the current output port.

```
(test-silence) → boolean?
(test-silence silence?) → void?
  silence? : any/c
```

A parameter that stores a boolean, defaults to #f, that can be used to suppress the printed summary from test.

```
(test-execute) → boolean?
(test-execute execute?) → void?
  execute? : any/c
```

A parameter that stores a boolean, defaults to #t, that can be used to suppress evaluation of test expressions.

2 Running Tests and Inspecting Test Results

```
(require test-engine/test-engine)        package: htdp-lib
```

This module defines language-agnostic procedures for running test code to execute checks, and recording and inspecting their results.

A *test* is a piece of code run for testing, a *check* is a single assertion within that code: Typically the tests are first registered, then they are run, and then their results are inspected. Both tests and the results of failed checks are recorded in a data structure called a *test object*. There is always a current test object associated with the current namespace.

```
(struct test-object (tests
                     successful-tests
                     failed-checks
                     signature-violations))
  tests : (listof (-> boolean?))
  successful-tests : (listof (-> boolean?))
  failed-checks : (listof failed-check?)
  signature-violations : (listof signature-violation?)
```

The four components of a `test-object` are all in reverse order:

The first one is the list of tests (each represented by a thunk), the others are succeeded tests, failed checks and signature violations, respectively.

The thunks are expected to always run to completion. They shouöd return `#t` upon success, and `#f` upon failure.

```
(empty-test-object) → test-object?
```

Creates an empty test object.

```
(current-test-object) → test-object?
```

Returns the current test object.

```
(initialize-test-object!) → any
```

Initializes the test object. Note that this is not necessary before using `current-test-object` and the various other functions operating on it: These will automatically initialize as necessary. Use this function to reset the current test object.

```
(add-test! thunk) → any
  thunk : (-> boolean?)
```

Register a test, represented by a thunk. The thunk, when called, is expected to call `add-failed-check!` and `add-signature-violation!` as appropriate.

```
(add-failed-check! failed-check) → any
  failed-check : failed-check?
```

Record a test failure.

```
(add-signature-violation! violation) → any
  violation : signature-violation?
```

Record a signature violation.

```
(run-tests!) → test-object?
```

Run the tests, calling the thunks registered via `add-test!` in the order they were registered.

```
(struct failed-check (reason srcloc?))
  reason : fail-reason?
  srcloc? : (or/c #f srcloc?)
```

This is a description of a failed check. The source location, if present, is from an expression that may have caused the failure, possibly an exception.

```
(struct fail-reason (srcloc))
  srcloc : srcloc?
```

Common supertype of all objects describing a reason for a failed check. The `srcloc` is the source location of the check.

```
(struct unexpected-error fail-reason (srcloc expected exn))
  srcloc : srcloc?
  expected : any/c
  exn : exn?
```

An error happened instead of regular termination.

```
(struct unexpected-error/markup unexpected-error (srcloc
                                                  expected
                                                  exn
                                                  error-markup))
  srcloc : srcloc?
  expected : any/c
  exn : exn?
  error-markup : markup?
```

An error happened instead of regular termination. This also contains markup describing the error.

```
(struct unequal fail-reason (srcloc actual expected))
  srcloc : srcloc?
  actual : any/c
  expected : any/c
```

A value was supposed to be equal to another, but wasn't. Generated by `check-expect`.

```
(struct not-within fail-reason (srcloc actual expected range))
  srcloc : srcloc?
  actual : any/c
  expected : any/c
  range : real?
```

A value was supposed to be equal to another within a certain range, but wasn't. Generated by `check-within`.

```
(struct incorrect-error fail-reason (srcloc expected exn))
  srcloc : srcloc?
  expected : any/c
  exn : exn?
```

An exception was expected, but a different one occurred. Generated by `check-error`.

```
(struct incorrect-error/markup incorrect-error (srcloc
                                                expected
                                                exn
                                                error-markup))
  srcloc : srcloc?
  expected : any/c
  exn : exn?
  error-markup : markup?
```

An exception was expected, but a different one occurred. Also includes markup describing the error. Generated by `check-error`.

```
(struct expected-error fail-reason (srcloc message value))
  srcloc : srcloc?
  message : (or/c #f string?)
  value : any/c
```

An error was expected, but a value came out instead. Generated by `check-error`.

```
(struct not-mem fail-reason (srcloc actual set))
  srcloc : srcloc?
  actual : any/c
  set : (listof any/c)
```

The value produced was not part an the expected set. Generated by `check-member-of`.

```
(struct not-range fail-reason (srcloc actual min max))
  srcloc : srcloc?
  actual : real?
  min : real?
  max : real?
```

The value produced was not part an the expected range. Generated by `check-range`.

```
(struct satisfied-failed fail-reason (srcloc actual name))
  srcloc : srcloc?
  actual : any/c
  name : string?
```

The value produced did not satisfy a predicate. The `name` field is the name of the predicate. Generated by `check-satisfied`.

```
(struct unsatisfied-error fail-reason (srcloc name exn))
  srcloc : srcloc?
  name : string?
  exn : exn?
```

A value was supposed to satsify a predicate, but an error happened instead. The `name` field is the name of the predicate. Generated by `check-satisfied`.

```
(struct unsatisfied-error/markup unsatisfied-error (srcloc
                                                    name
                                                    exn
                                                    error-markup))
  srcloc : srcloc?
  name : string?
  exn : exn?
  error-markup : markup?
```

A value was supposed to satsify a predicate, but an error happened instead. The `name` field is the name of the predicate. Also includes markup describing the error. Generated by `check-satisfied`.

```
(struct violated-signature fail-reason (srcloc
                                        obj
                                        signature
                                        blame-srcloc))
  srcloc : srcloc?
  obj : any/c
  signature : signature?
  blame-srcloc : (or/c #f srcloc?)
```

A signature was violated, and this was communicated via an exception. Note that signature violations should really be (and usually are) communicated via add-signature-violation!.

```
(struct signature-got (value))
  value : any/c
```

The value that violated the signature.

```
(struct signature-violation (obj
                             signature
                             message
                             srcloc
                             blame-srcloc))
  obj : any/c
  signature : signature?
  message : (or/c string? signature-got?)
  srcloc : (or/c #f srcloc?)
  blame-srcloc : (or/c #f srcloc?)
```

Signature signature was violated by object obj. The srcloc field is the location of the signature. The optional blame-srcloc points at the source code to blame for the violation.

```
(struct property-fail fail-reason (srcloc result))
  srcloc : srcloc?
  result : check-result?
```

A counterexample for a property was found, described in the result field.

```
(struct property-error fail-reason (srcloc exn))
  srcloc : srcloc?
  exn : exn?
```

A property check produced an unexpected exception.

3 Printing Test Results

This module is responsible for output of test results: Where the output goes, and some aspects of the formatting can be customized via parameters.

```
(require test-engine/test-markup)        package: htdp-lib
```

```
(render-value-parameter) → (any/c . -> . string?)
(render-value-parameter render-value-proc) → void?
  render-value-proc : (any/c . -> . string?)
```

This parameter determines how `test-object->markup` renders a value for display in an error message in a language-specific way. The default is `(lambda (v) (format "~V" v))`.

```
(display-test-results-parameter) → (markup? . -> . any)
(display-test-results-parameter display-test-proc) → void?
  display-test-proc : (markup? . -> . any)
```

This parameter determines how to output the test results. The default prints to `(current-output-port)`.

```
(display-test-results! test-object) → any
  test-object : test-object?
```

This just calls the procedure bound to `display-test-results-parameter`.

```
(get-rewritten-error-message-parameter)
 → (exn? . -> . string?)
(get-rewritten-error-message-parameter get-rewritten-error-message-proc)
 → void?
  get-rewritten-error-message-proc : (exn? . -> . string?)
```

This parameter determines how to get an error message from an exception, possibly after reformulation and/or translation.

```
(get-rewritten-error-message exn) → string?
  exn : exn?
```

This just calls the procedure bound to `get-rewritten-error-message-parameter`.

```
(test-object->markup test-object) → markup?
  test-object : test-object?
```

This generates a test report as markup, using `render-value-parameter` and `get-rewritten-error-message-parameter`.