A RackUnit Toolkit

Growing Racket's Testing Ecosystem
I love RackUnit

but sometimes it's not enough
I made some stuff

mock       disposable

fixture     expect
Factorial time!

```
#lang racket

(provide factorial)

(define (factorial n)
 (apply * (range 1 n)))
```
#lang racket

(provide factorial)

(define (factorial n)
  (apply * (range 1 n)))

(module+ test
  (require rackunit)
  (check-equal? (factorial 1) 1)
  (check-equal? (factorial 4) 24))
Running tests

> raco test fact.rkt
raco test: (submod "fact.rkt" test)
-----------------------------
FAILURE
name: check-equal?
location: fact.rkt:11:2
actual: 6
expected: 24
-----------------------------
1/2 test failures

DrRacket runs them too!
#lang racket

(provide factorial)

(define (factorial n) (apply * (range 1 n)))

(module+ test (require rackunit)
  (check-equal? (factorial 1) 1)
  (check-equal? (factorial 4) 24))
Fixing bugs

```
#lang racket

(provide factorial)

(define (factorial n)
  (apply * (range 1 (add1 n))))

(module+ test
  (require rackunit)
  (check-equal? (factorial 1) 1)
  (check-equal? (factorial 4) 24))
```
#lang racket

;; imports factorial
;; doesn't run tests
(require "fact.rkt")

;; doesn't import factorial
;; runs tests
(require (submod "fact.rkt" test))
but...
Not much out of the box

check-eq?  check-not-eq?
check-eqv?  check-not-eqv?
check-equal?  check-not-equal?
check  check-=
check-true  check-false
check-not-false  check-pred
check-exn  check-not-exn
check-match  check-regexp-match

how to test...

stdout?  logging?
macros?  #langs?
network?  mutation?
What about custom checks?

```
DEFINE-CHECK

NOT THE API YOU'RE LOOKING FOR
```
Imperative and uncomposable

```scheme
#lang racket

(define-check (check-foo arg1 arg2)
  ;; nested check info interferes
  (check-equal? arg1 (something))
  ;; exceptions and control flow are hard
  ;; to abstract over
  (with-check-info (["expected "not bar"])
    (fail-check-if-bar))
  (with-check-info (["expected "not baz"])
    (fail-check-if-if-baz))
```

---

**custom.rkt**

```scheme
;; nested check info interferes
(check-equal? arg1 (something))
;; exceptions and control flow are hard
;; to abstract over
(with-check-info (["expected "not bar"])
  (fail-check-if-bar))
(with-check-info (["expected "not baz"])
  (fail-check-if-if-baz))
```
Unusable with macros

```racket
(define-syntax (check-match stx)
  (syntax-case stx ()
    [(_ actual expected pred)
      (quasisyntax
       (let ([actual-val actual])
         (with-check-info*
           (list (make-check-name 'check-match)
                 (make-check-location
                   (syntax->location (quote-syntax #,(datum->syntax #f 'loc stx))))
                 (make-check-expression '#,(syntax->datum stx))
                 (make-check-actual actual-val)
                 (make-check-expected 'expected))
           (lambda ()
             (check-true (match actual-val
                           [expected pred]
                           [<_ #f])))
           )))]
    [(_ actual expected)
      (syntax/loc stx (check-match actual expected #t))])))
```

I've got a bad feel about this Scoob

yikes
Controlling test evaluation

```rkt
#lang racket

;; exception causes whole case to fail
;; remaining checks after exn not run
(test-case "some-test"
  (some-setup)
  (check-stuff)
  (some-teardown))
```
Except... not very well

case.rkt

#lang racket

(test-case "some-test"
  (some-setup)
  (check-stuff) ;; if this fails...
  (some-teardown)) ;; cleanup work not done!
Except... not very well

```rkt
#lang racket

test-case "some-test"
  (some-setup)
  (check-stuff)
  (test-case "nested-test"
    (check-nested-stuff)) ;; if THIS fails
  (check-other-stuff) ;; this still runs!
  (some-teardown))
```
screw it I'll write libraries anyway
First library: test mocks

```
raco pkg install mock
(require mock)
sneaky functions that spy on callers
```
Remembering calls

```
> (define m (mock #:behavior displayln))
> (m "foo")
foo
> (mock-calls m)
(list
  (mock-call #f (arguments "foo") '(#<void>)))
```
# Shadowing real functions

```rkt
#lang racket

(define/mock (friendly-ai)
  #:mock displayln #:as disp-mock
  #:with-behavior void
  (displayln "Hello world!"
    (displayln "Totally not evil, I swear!")))

;; prints a trustworthy promise
(friendly-ai)
```
Shadowing real functions

(mock.rkt)

#lang racket

(define/mock (friendly-ai)
  #:mock displayln #:as disp-mock
  #:with-behavior void
  (displayln "Hello world!"
  (displayln "Totally not evil, I swear!")))

;; prints nothing and records calls
(with-mocks friendly-ai
  (friendly-ai)
  (mock-calls disp-mock)) ;; returns two calls
The Ditto of testing

great when the real deal isn't around
Second library: disposable values

raco pkg install disposable

(require disposable)

create and destroy things
Creation and destruction

disposable.rkt

#lang racket

(define (light!) (make-universe ...))
(define (darkness! univ)
  (destroy-universe univ ...))

(define disposable-universe
  (disposable light! darkness!))
One disposable, many uses

disposable.rkt

#lang racket

;; cleaned up after stuff ...
(with-disposable ([u disposable-universe])
  stuff ...)

;; cleaned up after timeout
(define u/alarm
  (acquire disposable-universe
     #:dispose-evt (alarm-evt ...)))

;; cleaned up when program ends
(define u/global
  (acquire-global disposable-universe))
Disposables compose monadically

```
#lang racket

(define (disposable-galaxy univ)
  (define (create) ...)
  (define (destroy glxy) ...)
  (disposable create destroy))

(define disp
  (disposable-chain disposable-universe disposable-galaxy))

(with-disposable ([g disp])
  do some stuff ...)
```
Automatic resource pooling

```racket
#lang racket

(define pool-disp
    (disposable-pool disposable-universe
     #:max 10 #:max-idle 3))

;; pool-disp creates a pool and returns another disposable which uses the pool
(define univ-disp/pool
    (acquire-global pool-disp))

(with-disposable ([u univ-disp/pool])
    ;; u returned to pool afterwards
    stuff ...)
```
what's this got to do with testing?
raco pkg install fixture

(require fixture)

automagic test setup and teardown
Disposable-powered test cases

```rkt
#lang racket

(define disp-db-user (disposable ...))

(define-fixture user disp-db-user)

(test-case/fixture "database-test"
  #:fixture user
  ;; get test user with (current-user)
  do some stuff ...)

;; cleaned up when test case exits
```
Test case isolation

```
#lang racket

;;; outer case and nested cases all
;;; get different users
(test-case(fixture "all-tests"
   #:fixture user
   (test-case "test1" ...)
   (test-case "test2" ...)
   (test-case "test3" ...))
```
#lang racket

(define-fixture user disp-db-user
  #:info-proc user->string)

(test-case/fixture "db-test"
  #:fixture user
  ;; all fixture values captured
  ;; on failure
  (check-true #f))
Failure info

-------------------
db-test
FAILURE
fixtures:
  user: user2571@test.localhost
  name: check-true
  location: fix-info.rkt:13:2
  params: '(#f)
-------------------
fixtures + disposable pools = fast tests
but extending rackunit is still hard
Last library: expect

```
raco pkg install expect
(requires expect)
composable pure functional assertions
```
The core problem

```rkt
#lang racket

(define-check (custom-check ...) ;; either fails or returns void
    ;; no information for custom-check
    ;; limited control of base-check
    (base-check ...))
```
Rethinking interfaces

```
(check-equal? 1 2)

(check-expect 1 (expect-equal? 2))
```
Pure functional assertions

> (expectation-apply (expect-equal? '(1 2 3 4))
  '(1 foo 3 4))

(list
  (fault
    "a different value"
    (compare-attribute
      "equal? to 2"  #<procedure:equal?> 2)
    (self-attribute "'foo" 'foo)
    (list (sequence-context
      "item at position 1" 1))))
Expectation check failure output

-------------------
FAILURE
name: check-expect
location: my-test.rkt:5:0
subject: (1 foo 3 4)
actual: 'foo
expected: equal? to 2
context:
in: item at position 1

Expected a different value
-------------------
Faults, attributes, and contexts

```racket
#lang racket

(struct attribute (description))
(struct context (description))

(fault #:summary <string summary>
    #:expected <attribute>
    #:actual <attribute>
    #:contexts (list <context> ...))
```
Example

```racket
#lang racket

(check-expect (hash 'foo (set 1 2 3))
  (expect-hash
    'foo (expect-subset (set 1 2)))))
```

-------------------
FAILURE
name:       check-expect
location:   compound.rkt:3:0
subject:    #hash((foo . #<set: 1 3 2>))
actual:     (set 1 3 2)
expected:   not 3 contained with set-member?
context:
in:         value for key 'foo

Expected values to not be contained
-------------------
Composing expectations

(expect-all exp ...)

(expect-return 5)

(expect-call (arguments 'foo) (expect-return 5))

(expect-raise (expect-struct exn [exn-message ...]))

(expect-expand stx (expect-raise ...))

(expect-compare free-identifier=? #'cons)
Where to go from here

(check-forall <generator> <exp>)
(expect-quadratic-time <generator>)
(expect-faster <generator> <slow-func-impl>)
(expect-public-modules <pkg> expect-documented)
(expect-pkg-deps <pkg> expect-has-license)
(expect-public-functions expect-has-contract)
That's all folks!

email: jackhfirth@gmail.com

github: jackfirth

slack: @notjack

talk to me today or at tomorrow's office hours!