MAGS Test Runners

Karissa R. McKelvey <krmckelv@indiana.edu>
Aaron W. Hsu <arcfide@sacrideo.us>

2 March 2011

Contents

Library Definition: (mags runners)

This library houses custom test-runners for (mags grade). These test-runners are based upon the framework of SRFI :64. More about customizing `test-runners` can be found on the SRFI-64 Documentation[1].

Exports:

`test-runner-quiet test-runner-verbose`

Imports:

`(chezscheme) (srfi :64) (mags sandbox)`

1. Overview. In test-runners, for each event, there is a specific callback function that is called. This gives us a lot of control over the output of the test runner. For more specifics about writing your own test runners, visit the (SRFI :64) documentation noted in the beginning of this library. Let's look at the various test-runners defined for `mags`, and discuss each one.

`test-runner-verbose`

`test-runner-quiet` The test runner quiet is like the test-runner-verbose, except it does not report passed tests.

2. Test Runners: test-runner-verbose. This test runner, named `test-runner-verbose`, uses two ports, `s-port`, for student output and `p-port` a port for professor output. The ports must be strings or open ports. The test runner will output to both ports in the following way:

To the student: A printing of the simple result of each test (either FAILED or PASSED), with any exceptions or conditions printed as well as the overall tally.

To the professor: In addition to the above, the specific tests which failed as well as the expected and actual values of the test cases. **3.** The `test-runner-verbose` assumes that each problem set will be contained within a test group. For example,

```
(test-group "Assignment 10"

    (test-group
     (test-group "lists"
       (test-equal '(3 2 1) (reverse '(1 2 3)))
       (test-equal '() (error 'foo "bar"))))
    (test-group
     (test-group "insert"
     (test-equal "3 into list of size 1" '(1 3) (insert 3 '(1)))))
    )
```

Output for `student-port`:

```
Results for Assignment 10

1:lists: Incorrect
2:insert: Passed

Your assignment has been successfully loaded and autograded.
```

Output for `professor-port`:

```
Results for Assignment 10

1:lists:
FAILED
  Tested: (reverse '(1 2 3))
```

[1] <http://srfi.schemers.org/srfi-64/srfi-64.html>

```
  Expected: (3 2 1)
  Actual: (2 1 3)

FAILED
  Tested: '(1)
  Error: Exception in foo: bar

2:insert: PASSED

Test Results
  Passed: 1
  Failed: 1
  Missing: 0
```

4. A `test-runner-verbose` is defined here. It takes two arguments, both assumed to be either strings or open ports. One represents the output to the professor and the other to the student. Test runners have callback functions that are used on certain cases.

⟨Define test-runner-verbose⟩ ≡

```
(define (test-runner-verbose pp sp)
  (let ([port ⟨get-port 23⟩]
        [student-port ⟨get-port 23⟩]
        [runner (test-runner-null)]
        [passed-count 0]
        [failed-count 0]
        [missing-count 0])
    ⟨On-group-begin verbose 10⟩
    ⟨On-test-end verbose 8⟩
    ⟨On-bad-count 21⟩
    ⟨On-bad-end-name 22⟩
    ⟨On-group-end verbose 9⟩
    ⟨On-final verbose 11⟩
    runner))
```

Exports: `test-runner-verbose`

5. Test Runners: test-runner-quiet. This test runner, named `test-runner-quiet`, is of the same format as the above `test-runner-verbose`, except passes are unreported.

6. A `test-runner-quiet` is defined here. It takes two arguments, both assumed to be either strings or open ports. One represents the output to the professor and the other to the student have callback functions that cases.

⟨Define test-runner-quiet⟩ ≡

```
(define (test-runner-quiet pp sp)
  (let ([port ⟨get-port 23⟩]
        [student-port ⟨get-port 23⟩]
        [runner (test-runner-null)]
        [passed-count 0]
        [failed-count 0]
        [missing-count 0]
        [resulted?  #f]
        [failed?  #f])
```

⟨On-group-begin quiet 16⟩
⟨On-test-end quiet 14⟩
⟨On-bad-count 21⟩
⟨On-bad-end-name 22⟩
⟨On-group-end quiet 15⟩
⟨On-final quiet 17⟩
`runner))`

Exports: `test-runner-quiet`

7. Test-runner-verbose implementation. Following are the implementations of the above test-runners

8. The result of each test is handled in the `test-runner-verbose` by checking to see which type of result occured: an error, a fail, or a pass. If it errors, it prints the error using the printing convention `Print error`, else it prints the resutl with any errors. It uses the `test-runner-aux-value` of the runner to store this information in order to ensure only one pass or fail is recorded within a test-group.

When no name is provided to a test-¡equality-pred¿ test, `test-runner-verbose` pretty-prints the full tested expression.

⟨On-test-end verbose⟩ ≡

```
(test-runner-on-test-end!
  runner
  (lambda (runner)
    (let ([result-kind (test-result-kind runner)]
          [was-error?  (test-result-ref runner 'was-error?)]
          [expected (test-result-ref runner 'expected-value)]
          [actual (test-result-ref runner 'actual-value)]
          [test-name ⟨Truncate test-name 24⟩])
      (cond
        [was-error?
         (unless (equal?  (test-runner-aux-value runner) 'error)
           (begin
             ⟨Print error 24⟩
             (unless (equal?  (test-runner-aux-value runner) 'fail)
               (format student-port " Incorrect ~%"))
             (test-runner-aux-value!  runner 'error)))]
        [(or (equal?  result-kind 'xpass) (equal?  result-kind 'fail))
         (begin
           (format port
             "~%FAILED:~%   Test:          ~d~%   Expected:  ~d~%   Actual:      ~d~%"
             test-name expected actual)
           (test-runner-aux-value!  runner 'fail))]
        [(test-passed?)
         (if (equal?  (test-runner-aux-value runner) 'pass)
             (format port ".")
             (begin
               (test-runner-aux-value!  runner 'pass)
               (format port " PASSED")))]))))
```

Captures: `runner port student-port failed-count missing-count`

9. At the end of each group, we must update the passed and failed count as well as reset the `test-runner-aux-value` so that the next test-group can be recorded correctly as a problem set.

⟨On-group-end verbose⟩ ≡

```
(test-runner-on-group-end!
```

```
runner
(lambda (runner)
  (when (equal?  (test-runner-aux-value runner) 'pass)
    (begin
      (format student-port " Passed ~%")
      (set!  passed-count (add1 passed-count))
      (test-runner-aux-value!  runner #f)))
  (unless (test-result-ref runner 'was-error?)
    (when (equal?  (test-runner-aux-value runner) 'fail)
      (begin
        (format student-port " Incorrect ~%")
        (set!  failed-count (add1 failed-count))
        (test-runner-aux-value!  runner #f))))
  (when (equal?  (test-runner-aux-value runner) 'error)
    (begin
      (set!  failed-count (add1 failed-count))
      (test-runner-aux-value!  runner #f)))
  (unless (equal?  (test-runner-aux-value runner) 'error)
    (format port "~%"))))
```

Captures: `runner` `port` `student-port` `passed-count` `failed-count`

10. When the group begins, we need to print the proper headers. If its top-level, which means that this is the first test-group, it prints the name of the test-group, otherwise it prints the group name.

⟨On-group-begin verbose⟩ ≡

```
(test-runner-on-group-begin!
  runner
  (lambda (runner suite-name count)
    (if ⟨top-level 23⟩
        (begin
          (format port "Results for ~d ~%" suite-name)
          (format student-port "Results for ~d ~%" suite-name))
        (begin
          ⟨Print Group 24⟩
          ⟨Print Group 24⟩))))
```

Captures: `runner` `port` `student-port` `suite-name`

11. Finally, the runner pritns the results and closes the ports.

⟨On-final verbose⟩ ≡

```
(test-runner-on-final!
  runner
  (lambda (runner)
    ⟨Print End Results 24⟩
    ⟨Print Successful Load 24⟩
    (close-output-port student-port)
    (close-output-port port)))
```

Captures: `runner` `port` `student-port` `passed-count` `failed-count` `missing-count`

12. Push test-runner-verbose to the top level

⟨∗⟩ ≡

⟨Define test-runner-verbose 4⟩

13. Implementation of `test-runner-quiet`. Each of the following subsections give a specific part of the implementation of the test-runner.

14. The result of each test is handled in the `test-runner-quiet` by checking to see which type of result occured: an error, a fail, or a pass. It is very similar to the `test-runner-verbose`, except nothing is printed in the result of a pass.

When no name is provided to a test-¡equality-pred¿ test, `test-runner-quiet` pretty-prints the full tested expression.

⟨On-test-end quiet⟩ ≡

```
(test-runner-on-test-end!
  runner
  (lambda (runner)
    (let ([result-kind (test-result-kind runner)]
          [was-error?  (test-result-ref runner 'was-error?)]
          [expected (test-result-ref runner 'expected-value)]
          [actual (test-result-ref runner 'actual-value)]
          [test-name ⟨Truncate test-name 24⟩])
      (cond
        [was-error?
         (unless failed?
           ⟨Print result header 24⟩
           ⟨Print full problem 24⟩
           ⟨Print error 24⟩
           (set!  failed?  #t))]
        [(or (equal?  result-kind 'xpass) (equal?  result-kind 'fail))
         (unless failed?
           ⟨Print result header 24⟩
           ⟨Print full problem 24⟩
           (format port
             " FAILED ~%   Test:          ~d~%   Expected:  ~d~%   Actual:      ~d~%"
             test-name expected actual)
           (test-runner-aux-value!  runner 'fail)
           (set!  failed?  #t))]
        [(test-passed?)  (test-runner-aux-value!  runner 'pass)])))))
```

Captures: `runner port student-port failed-count missing-count resulted? failed?`

15. At the end of each group, we must update the passed and failed count as well as reset the `test-runner-aux-value` so that the next test-group can be recorded correctly as a problem set.

⟨On-group-end quiet⟩ ≡

```
(test-runner-on-group-end!
  runner
  (lambda (runner)
    (cond
      [failed?  (set!  failed-count (add1 failed-count))]
      [(equal?  (test-runner-aux-value runner) 'pass)
       (set!  passed-count (add1 passed-count))])
    (test-runner-aux-value!  runner #f)
    (set!  failed?  #f)))
```

Captures: `runner passed-count failed-count failed?`

16. When the group begins, we need to print the proper headers. If its top-level, which means that this is the first test-group, it prints 'Results for ¡group name¿', otherwise it simply prints the group name.
⟨On-group-begin quiet⟩ ≡

```
(test-runner-on-group-begin!
  runner
  (lambda (runner suite-name count)
    (if ⟨top-level 23⟩
        (begin (format port "Results for ~d ~%" suite-name)))))
```

Captures: `runner port student-port suite-name`

17. This chunk prints the end results and closes the ports, at the final call of the runner.
⟨On-final quiet⟩ ≡

```
(test-runner-on-final!
  runner
  (lambda (runner)
    ⟨Print End Results 24⟩
    (if (and (zero?  failed-count) (zero?  missing-count))
        (format student-port "All programs passed all tests."))
    (close-output-port student-port)
    (close-output-port port)))
```

Captures: `runner port student-port passed-count failed-count missing-count`

18. Push test-runner-quiet to the top level
⟨∗⟩ ≡

```
⟨Define test-runner-quiet 6⟩
```

19. The `max-name-length` is a parameter you can use to control how long the maximum printed name of test cases can be before truncation. This defaults to 80. A value of 0 denotes unlimited length.
⟨∗⟩ ≡

```
(define max-name-length
  (make-parameter
    80
    (lambda (int) (assert (integer?  int)) int)))
```

20. Common call-back functions. These call-back functions are common to all test runners. **21.** There are times when the runner will find a bad count of tests within a group. This is usually due to mismatched parens.
⟨On-bad-count⟩ ≡

```
(test-runner-on-bad-count!
  runner
  (lambda (runner)
    (format
      port
      "Bad number of tests have been recorded.  Please check your test suite for typos
or mismatched parens near test ~d."
      (test-runner-test-name runner))))
```

Captures: `runner port`

22. The runner may find a misspelled name in one of the `test-end` clauses. This can be avoided by using `test-group`. The runner will print a message stating the fact.
⟨On-bad-end-name⟩ ≡

```
(test-runner-on-bad-end-name!
```

```
runner
(lambda (runner)
  (format
    port
    "Please check your test cases.  There is a mispelled end name near test ~d."
    (test-runner-test-name runner))))
```

Captures: `runner` `port`

23. Helpers. These are helpers. `top-level` tells us if we are at the top level of the group stack. `get-port` turns a port into the appropriate file output port, or throws an error if the port is invalid.

⟨top-level⟩ ≡
```
(null?  (test-runner-group-stack runner))
```

Captures: `runner`

⟨get-port⟩ ≡
```
(cond
  [(port?  p) p]
  [(string?  p)
   (open-file-output-port
     p
     (file-options no-fail)
     (buffer-mode block)
     (native-transcoder))]
  [else
   (error 'grade
     "unexpected output type, expected file or port"
     p)])
```

Captures: `p`

24. Printing Conventions. These are conventions we can use when printing to a given port. This makes printing easier by keeping them as conventions which we can reuse.

⟨Print result header⟩ ≡
```
(if (and (or (equal?  result 'error) (equal?  result 'fail))
         (not resulted?))
    (begin
      (format
        port
        "The following programs failed one or more tests:~%")
      (set!  resulted?  #t)))
```

Captures: `port` `result` `resulted?`

⟨Print error⟩ ≡
```
(let ([actual (test-result-ref runner 'actual-value)]
      [test-name ⟨Truncate test-name 24⟩])
  (unless (equal?  (test-runner-aux-value runner) 'error)
    (cond
      [(timeout?  actual)
       (begin
         (format port " FAILED~%   Test:    ~d~%" test-name)
```

```
            (format port "   Error:  Probable Infinite Loop~%"))]
          [(unbound-term?  actual)
           (begin
             (test-runner-aux-value!  runner 'missing)
             (set!  missing-count (add1 missing-count))
             (set!  failed-count (sub1 failed-count))
             (format port " MISSING~%"))]
          [else
           (begin
             (format port " FAILED~%   Test:    ~d~%" test-name)
             (format port "   Error:  ")
             (display-condition actual port)
             (format port "~%"))])
        (test-runner-aux-value!  runner 'error)))
```

Captures: `port runner failed-count missing-count`

⟨Print Group⟩ ≡

```
(format port "~d:" output)
```

Captures: `port output`

⟨Print full problem⟩ ≡

```
(when (not (null?  (test-runner-group-stack runner)))
  (begin
    ⟨Print to two ports 24⟩
    (when (not (null?  (cdr (test-runner-group-stack runner))))
      (format student-port "  Problem ")
      ⟨Print Group 24⟩
      ⟨Print Group 24⟩
      (format student-port " ")
      (format port " "))
    ⟨Print Group simple 24⟩
    ⟨Print Group 24⟩))
```

Captures: `runner port student-port resulted?`

⟨Print Group simple⟩ ≡

```
(format port "~d" output)
```

Captures: `port output`

⟨Print End Results⟩ ≡

```
(format port
  "~%Test Results~%   Passed:  ~d.~%   Failed:  ~d.  ~%   Missing:  ~d.  ~%"
  passed-count failed-count missing-count)
```

Captures: `runner port passed-count failed-count missing-count`

⟨Print to two ports⟩ ≡

```
(format port1 output)
(format port2 output)
```

Captures: `port1` `port2` `output`

⟨Print Successful Load⟩ ≡

```
(format
  port
  "~%Your submission has successfully been loaded and autograded.")
```

Captures: `port`

⟨Truncate test-name⟩ ≡

```
(begin
  (if (and (string?  name)
           (> (string-length name) (max-name-length)))
      (begin
        (string-truncate!  name (max-name-length))
        (string-append name "..."))
      (if (char=?
            #\newline
            (string-ref name (sub1 (string-length name))))
          (substring name 0 (sub1 (string-length name)))
          name)))
```

Captures: `name`

This concludes the definition of (mags runners).