Supporting the Regression Testing of Lisp Programs

Richard C. Waters

MIT AI Laboratory
545 Technology Sq.
Cambridge MA 02139

Dick@AI.MIT.EDU

How often have you made a change in a system to fix a bug or add a feature and been totally sure that the change did not affect anything else, only to discover weeks or months later that the change broke something?

In my personal experience, the single most valuable software maintenance tool is a *regression tester*, which maintains a suite of tests for a system and can run them automatically when the system is changed. The term "regression testing" is used, because each version of the system being tested is compared with the previous version to make sure that the new version has not *regressed* by losing any of the tested capabilities. The more comprehensive the test suite is, the more valuable this comparison becomes.

Creating a comprehensive test suite for a system requires significant effort, and running a test suite can require significant amounts of computer time. However, given a comprehensive test suite, regression testing detects an impressive number of bugs with remarkably little human effort.

The RT regression tester presented here supports the regression testing of systems written in Common Lisp. In addition to being a valuable tool, RT is an interesting example of the power of Lisp.

The unified nature of the Lisp programming environment and the fact that Lisp programs can be manipulated as data allows RT to be implemented in two pages of code. Merely implementing a batch-mode regression tester using an Algol-like language in a typical programming environment would require much more code. Implementing a highly interactive system like RT would be a major undertaking.

User's Manual for RT

The functions, macros, and variables that make up the RT regression tester are in a package called "RT". The ten exported symbols are documented below. If you want to refer to these symbols without a package prefix, you have to 'use' the package.

The basic unit of concern of RT is the *test*. Each test has an identifying name and a body that specifies the action of the test. Functions are provided for defining, redefining, removing, and performing individual tests and the test suite as a whole. In addition, information is maintained about which tests have succeeded and which have failed.

- **deftest** *name form* **&rest** *values*

Individual tests are defined using the macro **deftest**. The identifying *name* is typically a number or symbol, but can be any Lisp form. If the test suite already contains a test with the same (**equal**) *name*, then this test is redefined and a warning message printed. (This warning is important to alert the user when a test suite definition file contains two tests with the same name.) When the test is a new one, it is added to the end of the suite. In either case, *name* is returned as the value of **deftest** and stored in the variable ***test***.

```
(deftest t-1 (floor 15/7) 2 1/7) ⇒ t-1
(deftest (t 2) (list 1) (1)) ⇒ (t 2)
(deftest bad (1+ 1) 1) ⇒ bad
(deftest good (1+ 1) 2) ⇒ good
```

The *form* can be any kind of Lisp form. The zero or more *values* can be any kind of Lisp

objects. The test is performed by evaluating *form* and comparing the results with the *values*. The test succeeds if and only if *form* produces the correct number of results and each one is `equal` to the corresponding *value*.

- `*test*` *name-of-current-test*

  The variable `*test*` contains the name of the test most recently defined or performed. It is set by `deftest` and `do-test`.

- `do-test &optional` (*name* `*test*`)

  The function `do-test` performs the test identified by *name*, which defaults to `*test*`. Before running the test, `do-test` stores *name* in the variable `*test*`. If the test succeeds, `do-test` returns *name* as its value. If the test fails, `do-test` returns `nil`, after printing an error report on `*standard-output*`. The following examples show the results of performing two of the tests defined above.

```
(do-test '(t 2)) ⇒ (t 2)
(do-test 'bad) ⇒ nil ; after printing:
Test BAD failed
Form: (1+ 1)
Expected value: 1
Actual value: 2.
```

- `*do-tests-when-defined*` default value `nil`

  If the value of this variable is non-null, each test is performed at the moment that it is defined. This is helpful when interactively constructing a suite of tests. However, when loading a test suite for later use, performing tests as they are defined is not liable to be helpful.

- `get-test &optional` (*name* `*test*`)

  This function returns the *name*, *form*, and *values* of the specified test.

```
(get-test '(t 2)) ⇒ ((t 2) (list 1) (1))
```

- `rem-test &optional` (*name* `*test*`)

  If the indicated test is in the test suite, this function removes it and returns *name*. Otherwise, `nil` is returned.

- `rem-all-tests`

  This function reinitializes RT by removing every test from the test suite and returns `nil`. Generally, it is advisable for the whole test suite to apply to some one system. When switching from testing one system to testing another, it is wise to remove all the old tests before beginning to define new ones.

- `do-tests &optional` (*out* `*standard-output*`)

  This function uses `do-test` to run each of the tests in the test suite and prints a report of the results on *out*, which can either be an output stream or the name of a file. If *out* is omitted, it defaults to `*standard-output*`. `Do-tests` returns `t` if every test succeeded and `nil` if any test failed.

  As illustrated below, the first line of the report produced by `do-tests` shows how many tests need to be performed. The last line shows how many tests failed and lists their names. While the tests are being performed, `do-tests` prints the names of the successful tests and the error reports from the unsuccessful tests.

```
(do-tests "report.txt") ⇒ nil
; the file "report.txt" contains:
Doing 4 pending tests of 4 tests total.
 T-1 (T 2)
Test BAD failed
Form: (1+ 1)
Expected value: 1
Actual value: 2.
 GOOD
1 out of 4 total tests failed: BAD.
```

  It is best if the individual tests in the suite are totally independent of each other. However, should the need arise for some interdependence, you can rely on the fact that `do-tests` will run tests in the order they were originally defined.

- `pending-tests`

  When a test is defined or redefined, it is marked as *pending*. In addition, `do-test` marks the test to be run as pending before running it and `do-tests` marks every test as pending before running any of them. The only time a test is marked as not pending is when it completes successfully. The function `pending-tests` returns a list of the names of the currently pending tests.

```
(pending-tests) ⇒ (bad)
```

• **continue-testing**

This function is identical to **do-tests** except that it only runs the tests that are pending and always writes its output on ***standard-output***.

```
(continue-testing) ⇒ nil ; after printing:
Doing 1 pending test out of 4 total tests.
Test BAD failed
Form: (1+ 1)
Expected value: 1
Actual value: 2.
1 out of 4 total tests failed: BAD.
```

Continue-testing has a special meaning if called at a breakpoint generated while a test is being performed. The failure of a test to return the correct value does not trigger an error break. However, there are many kinds of things that can go wrong while a test is being performed (e.g., dividing by zero) that will cause breaks.

If **continue-testing** is evaluated in a break generated during testing, it aborts the current test (which remains pending) and forces the processing of tests to continue. Note that in such a breakpoint, ***test*** is bound to the name of the test being performed and **(get-test)** can be used to look at the test.

When building a system, it is advisable to start constructing a test suite for it as soon as possible. Since individual tests are rather weak, a comprehensive test suite requires large numbers of tests. However, these can be accumulated over time. In particular, whenever a bug is found by some means other than testing, it is wise to add a test that would have found the bug and therefore will ensure that the bug will not reappear.

Every time the system is changed, the entire test suite should be run to make sure that no unintended changes have occurred. Typically, some tests will fail. Sometimes, this merely means that tests have to be changed to reflect changes in the system's specification. Other times, it indicates bugs that have to be tracked down and fixed. During this phase, **continue-testing** is useful for focusing on the tests that are failing. However, for safety sake, it is always wise to reinitialize RT, redefine the entire test suite, and run **do-tests** one more time after you think all of the tests are working.

How RT Works

The code for RT is shown in Figures 1 & 2. The first figure shows the functions for maintaining the suite of tests. For the most part, the code is self explanatory. However, several points are worthy of note.

The test suite is represented as a list of *test entries* stored in the variable ***entries***. The list begins with a dummy entry of **nil** so that insertion and deletion of entries can be done by side-effect without having to handle an empty test suite as a special case. Each test entry contains five pieces of information:

- **pend** A flag that is non-null when the test is pending.
- **name** The name of the test represented by the test entry.
- **form** The form to evaluate when performing the test.
- **vals** The values specifying what the form should return.
- **defn** A list containing the **name**, **form**, and **vals**.

For efficiency, the entry data structure is represented as a list where the **pend**, **name**, and **form** fields are defined in the normal way, and the **vals** and **defn** fields are overlapping tails of the list.

Get-entry is broken out as a separate function, rather than being part of **get-test**, because it is a called by **do-test** as well.

The reason why **deftest** is a macro instead of a function is to allow tests to be defined without explicitly quoting the various parts of the definition.

The **copy-list** in **add-entry** is needed to ensure that evaluating a **deftest** a second time creates a fresh entry.

A desire to keep the entries on ***entries*** in the order that the tests are initially defined makes the main loop in **add-entry** somewhat complex. The loop searches through ***entries*** to see if there is a pre-existing test with the same name as the one being defined. If there is, the entry is replaced. If not, the new entry is placed at the end of ***entries***.

The error reporting done by **get-entry** and

```
(in-package "RT" :use '("LISP"))

(provide "RT")

(export
  '(deftest get-test do-test rem-test
    rem-all-tests do-tests pending-tests
    continue-testing *test*
    *do-tests-when-defined*))

(defvar *test* nil "Current test name")
(defvar *do-tests-when-defined* nil)
(defvar *entries* '(nil) "Test database")
(defvar *in-test* nil "Used by TEST")
(defvar *debug* nil "For debugging")

(defstruct (entry (:conc-name nil)
                  (:type list))
  pend name form)

(defmacro vals (entry) `(cdddr ,entry))

(defmacro defn (entry) `(cdr ,entry))

(defun pending-tests ()
  (do ((l (cdr *entries*) (cdr l))
       (r nil))
      ((null l) (nreverse r))
    (when (pend (car l))
      (push (name (car l)) r))))

(defun rem-all-tests ()
  (setq *entries* (list nil))
  nil)

(defun rem-test (&optional (name *test*))
  (do ((l *entries* (cdr l)))
      ((null (cdr l)) nil)
    (when (equal (name (cadr l)) name)
      (setf (cdr l) (cddr l))
      (return name))))
```

```
(defun get-test (&optional (name *test*))
  (defn (get-entry name)))

(defun get-entry (name)
  (let ((entry (find name (cdr *entries*)
                     :key #'name
                     :test #'equal)))
    (when (null entry)
      (report-error t
        "~%No test with name ~:@(~S~)."
        name))
    entry))

(defmacro deftest (name form &rest values)
  `(add-entry '(t ,name ,form .,values)))

(defun add-entry (entry)
  (setq entry (copy-list entry))
  (do ((l *entries* (cdr l))) (nil)
    (when (null (cdr l))
      (setf (cdr l) (list entry))
      (return nil))
    (when (equal (name (cadr l))
                 (name entry))
      (setf (cadr l) entry)
      (report-error nil
        "Redefining test ~@:(~S~)"
        (name entry))
      (return nil)))
  (when *do-tests-when-defined*
    (do-entry entry))
  (setq *test* (name entry)))

(defun report-error (error? &rest args)
  (cond (*debug*
         (apply #'format t args)
         (if error? (throw '*debug* nil)))
        (error? (apply #'error args))
        (t (apply #'warn args))))
```

Figure 1: The code for the part of RT that maintains the test suite.

`add-entry` is broken out into the separate function `report-error` to provide greater uniformity and facilitating the testing of RT.

It is often advisable to insert a few hooks in a system that facilitate testing. As illustrated in the next section, the use of the variable `*debug*` and the associated `throw` makes it possible to test the error checking done by RT without causing error breaks at testing time.

Figure 2 shows the code for running tests. Except for the `format` control strings—which, as always, are convenient but inscrutable—most of the code is self explanatory. Nevertheless, a couple of points are interesting.

The `catch` set up by `do-entry` is used by `continue-testing` to abort out of a test that has caused an error break. The variable `*in-test*` is used as an interlock to make sure that the function `continue-testing` will only do a `throw` when the appropriate `catch` exists. The way `do-entry` first sets the `pend` field of the entry to `t` and then resets it to reflect whether the test has succeeded causes the `pend` field to remain `t` when a test is aborted.

Because it does a lot of output, `do-entries` looks complex. However, it actually does little more than call `do-entry` on each pending test.

It was decided that `Continue-testing` did not need to have a stream argument, because `continue-testing` is only useful when using RT interactively.

One might be moved to say that the code in Figures 1 & 2 is too trivial to be an impressive example of the power of Lisp. However, this

```
(defun do-test (&optional (name *test*))
  (do-entry (get-entry name)))

(defun do-entry (entry &optional
                 (s *standard-output*))
  (catch '*in-test*
    (setq *test* (name entry))
    (setf (pend entry) t)
    (let* ((*in-test* t)
           (*break-on-warnings* t)
           (r (multiple-value-list
               (eval (form entry)))))
      (setf (pend entry)
            (not (equal r (vals entry))))
      (when (pend entry)
        (format s "~&Test ~:@(~S~) failed~
                   ~%Form: ~S~
                   ~%Expected value~P: ~
                     ~{~S~^~%~17t~}~
                   ~%Actual value~P: ~
                     ~{~S~^~%~15t~}.~%"
                *test* (form entry)
                (length (vals entry))
                (vals entry)
                (length r) r))))
      (when (not (pend entry)) *test*))

(defun continue-testing ()
  (if *in-test*
      (throw '*in-test* nil)
      (do-entries *standard-output*)))
```

```
(defun do-tests (&optional
                 (out *standard-output*))
  (dolist (entry (cdr *entries*))
    (setf (pend entry) t))
  (if (streamp out)
      (do-entries out)
      (with-open-file
          (stream out :direction :output)
        (do-entries stream))))

(defun do-entries (s)
  (format s "~&Doing ~A pending test~:P ~
             of ~A tests total.~%"
          (count t (cdr *entries*)
                 :key #'pend)
          (length (cdr *entries*)))
  (dolist (entry (cdr *entries*))
    (when (pend entry)
      (format s "~@[~<~%~:; ~:@(~S~)~>~]"
              (do-entry entry s))))
  (let ((pending (pending-tests)))
    (if (null pending)
        (format s "~&No tests failed.")
        (format s "~&~A out of ~A ~
                   total tests failed: ~
                   ~:@(~{~<~%   ~1:;~S~>~
                         ~^, ~}~)."
                (length pending)
                (length (cdr *entries*))
                pending))
    (null pending)))
```

Figure 2: The code for the part of RT that performs tests.

would be taking too narrow a view. The impressive thing about Figures 1 & 2 is not what they contain, but what the do not have to contain. In particular, most of what you would have to write to implement RT in other languages is provided by the Lisp environment and does not have to be written at all.

Consider what it would be like to implement RT in a language such as Ada. Because of the strong typing in Ada, one would probably be prevented from taking the simple approach of storing each test as a combination of a testing function to call and a group of data values. Rather, one would probably have to define each test as a separate function of no arguments. This would allow you to use the standard Ada compiler to prepare the tests for execution; however, you would have to write some amount of code outside of Ada (e.g., shell scripts in a UNIX system) to manage the process of defining and running tests.

For an Ada implementation to support the interactive running of individual test cases and reporting of the results, a user-interface module would have to be written. To go beyond this and allow the interactive (re)definition of tests, some escape to the surrounding operating system would be required to access the compiler. To take the final step of allowing the testing of a system to be intermixed with debugging, the implementation would have to be built as an extension to an interactive programming environment. Like any Lisp system, RT gets the benefit of this at no cost to the implementor whatever.

An Example Test Suite

Returning to the question of how RT can best be used, consider Figure 3, which shows the beginnings of a test suite for RT itself. There is a bit of gratuitous complexity because the system is being used to test itself. Nevertheless, the figure is a good example of what a typical test suite looks like. The first three lines of the

```
(in-package "USER")
(require "RT")
(use-package "RT")

(defmacro setup (&rest body)
  `(do-setup '(progn ., body)))

(defun do-setup (form)
  (let ((*test* nil)
        (*do-tests-when-defined* nil)
        (rt::*entries* (list nil))
        (rt::*debug* t)
        result)
    (deftest t1 4 4)
    (deftest (t 2) 4 3)
    (values
      (normalize
        (with-output-to-string
            (*standard-output*)
          (setq result
                (catch 'rt::*debug*
                  (eval form)))))
      result)))

(defun normalize (string)
  (let ((l nil))
    (with-input-from-string (s string)
      (loop (push (read-line s nil s) l)
            (when (eq (car l) s)
              (setq l (nreverse (cdr l)))
              (return nil))))
      (delete "" l :test #'equal)))

(rem-all-tests)

(deftest get-test-1
  (setup (get-test 't1))
  () (t1 4 4))
(deftest get-test-2
  (setup (get-test 't1) *test*)
  () (t 2))
(deftest get-test-3
  (setup (get-test '(t 2)))
  () ((t 2) 4 3))
(deftest get-test-4
  (setup (let ((*test* 't1)) (get-test)))
  () (t1 4 4))
(deftest get-test-5
  (setup (get-test 't0))
  ("No test with name T0.") nil)

(deftest do-test-1
  (setup (do-test 't1))
  () t1)
(deftest do-test-2
  (setup (do-test 't1) *test*)
  () t1)
(deftest do-test-3
  (setup (do-test '(t 2)))
  ("Test (T 2) failed"
   "Form: 4"
   "Expected value: 3"
   "Actual value: 4.")
  nil)
```

Figure 3: Some tests of RT itself.

figure specify that the test suite is in the "USER" package and prepare RT for use.

The function **setup** is used by the tests to create a safe environment where experiments can be performed without affecting the overall test suite in the figure. In preparation for these experiments, **setup** defines two example tests (**t1** and **(t 2)**). **Setup** captures any output created by **form** in a string and returns a list of the lines of output as its first value. **Setup** binds **rt::*debug*** to **t** (see Figure 1) and includes an appropriate **catch** so that the error checking done by RT can be tested.

The function **normalize** overcomes a minor problem in the portability of Common Lisp. Several of the format control strings in **do-entry** and **do-entries** use the control code **~&** (see Figure 2). Unfortunately, while this is better than **~%** in many situations, it is not guaranteed to behave differently, and Common Lisp implementations vary widely in what they do. **Normalize** removes any blank lines that result from **~&** acting like **~%**.

The first five tests in Figure 3 test the function **get-test**. Even for this trivial function, several tests are required to get reasonable coverage of its capabilities. **Get-test-5**, checks that **get-test** reports an error when given the name of a non-existent test.

The last three tests in Figure 3 test the function **do-test**. The full test suite for RT contains several more tests of **get-test** and **do-tests**, and some twenty more tests overall.

Acknowledgments

The concept of regression testing is an old one, and many (if not most) large programming organizations have regression testers. RT is the result of ten years of practical use and evolution. Many of the ideas in it came from conversations with Charles Rich and Kent Pitman, who implemented similar systems.

This paper describes research done at the MIT AI Laboratory. Support was provided by DARPA, NSF, IBM, NYNEX, Siemens, Sperry, and MCC. The views and conclusions presented here are those of the author and should not be interpreted as representing the policies, expressed or implied, of these organizations.

Obtaining RT

RT is written in portable Common Lisp and has been tested in several different Common Lisp implementations. The complete source for RT is shown in Figures 1–2. In addition, the source can be obtained over the INTERNET by using FTP. Connection should be made to the `AI.MIT.EDU` machine (INTERNET number 128.52.32.81). Login as "anonymous" and copy the files shown below. It is advisable to run the tests in `rt-test.lisp` after compiling RT for the first time on a new system.

In the directory `/pub/lptrs/`

`rt.lisp`	source code
`rt-test.lisp`	test suite
`re-doc.txt`	brief documentation