

Some Useful Lisp Algorithms: Part 1

Richard C. Waters

TR91-04 December 1991

Abstract

Richard C. Waters Chapter 3 Implementing Queues in Lisp (co-authored by P. Norvig) presents several different algorithms for implementing queues in Lisp. It discusses why the obvious list-based implementation of queues is inefficient and the particular situations where more complex implementations are appropriate.

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Mitsubishi Electric Research Laboratories, Inc.; an acknowledgment of the authors and individual contributions to the work; and all applicable portions of the copyright notice. Copying, reproduction, or republishing for any other purpose shall require a license with payment of fee to Mitsubishi Electric Research Laboratories, Inc. All rights reserved.

Mitsubishi Electric Research Laboratories

Technical Report 91-04

December 15, 1991

Some Useful Lisp Algorithms: Part 1

by

Richard C. Waters

Abstract

This technical report gathers together three papers that were written during 1991 and submitted for publication in *ACM Lisp Pointers*. Each paper describes a useful Lisp algorithm.

Chapter 1 “Supporting the Regression Testing of Lisp Programs” presents a system called RT that maintains a database of tests and automatically runs them when requested. This can take a lot of computer time, but does not take any of the programmer’s time. As a result, any bugs found by running the tests—and this is a lot more bugs than you might think—are essentially found for free.

Chapter 2 “Determining the Coverage of a Test Suite” presents a system called COVER that can help assess the coverage of a suite of test cases. When a suite of test cases for a program is run in conjunction with COVER, statistics are kept on which conditions in the code for the program are exercised and which are not. Based on this information, COVER can print a report of what has been missed. By devising tests that exercise these conditions, a programmer can extend the test suite so that it has more complete coverage.

Chapter 3 “Implementing Queues in Lisp” (co-authored by P. Norvig) presents several different algorithms for implementing queues in Lisp. It discusses why the obvious list-based implementation of queues is inefficient and the particular situations where more complex implementations are appropriate.

Submitted to *ACM Lisp Pointers*, January, November, and December 1991.

201 Broadway
Cambridge Massachusetts 02139

Publication History:-

1. First printing, TR 91-04, December 1991
2. Chapter 1 published as “Supporting the Regression Testing of Lisp Programs”,
ACM Lisp Pointers, 4(2):47–53, June 1991.

Copyright © Mitsubishi Electric Research Laboratories, 1991
201 Broadway; Cambridge Massachusetts 02139

This work may not be copied or reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for nonprofit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of Mitsubishi Electric Research Laboratories of Cambridge, Massachusetts; an acknowledgment of the authors and individual contributions to the work; and all applicable portions of the copyright notice. Copying, reproduction, or republishing for any other purpose shall require a license with payment of fee to Mitsubishi Electric Research Laboratories. All rights reserved.

1. Supporting the Regression Testing of Lisp Programs

Richard C. Waters

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 `def-test` 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 called by `do-test` as well.

The reason why `defest` 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 `defest` 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) '(cddddr ,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) (cddddr 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 (stream-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 (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 they 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 `FTP.AI.MIT.EDU` machine (INTERNET number 128.52.32.6). 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
```

The contents of Figures 1 & 2 and the files above are copyright 1990 by the Massachusetts Institute of Technology, Cambridge MA. Permission to use, copy, modify, and distribute this software for any purpose and without fee is hereby granted, provided that this copyright and permission notice appear in all copies and supporting documentation, and that the names of MIT and/or the author are not used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. MIT and the author make no representations about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty.

MIT and the author disclaim all warranties with regard to this software, including all implied warranties of merchantability and fitness, in no event shall MIT or the author be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

2. Determining the Coverage of a Test Suite

Richard C. Waters

The value of a suite of test cases depends critically on its *coverage*. Ideally a suite should test every facet of the specification for a program and every facet of the algorithms used to implement the specification. Unfortunately, there is no practical way to be sure that complete coverage has been achieved. However, something should be done to assess the coverage of a test suite, because a test suite with poor coverage has little value.

A traditional approximate method of assessing the coverage of a test suite is to check that every condition tested by the program is exercised. For every predicate in the program, there should be at least one test case that causes the predicate to be true and one that causes it to be false. Consider the function `my*` in Figure 4, which uses a convoluted algorithm to compute the product of two numbers.

The function `my*` contains two predicates, `(minusp x)` and `(minusp y)`, which lead to four conditions: `x` is negative, `x` is not negative, `y` is negative, and `y` is not negative. To be at all thorough, a test suite must contain tests exercising all four of these conditions. For instance,

```
(defun my* (x y)
  (let ((sign 1))
    (when (minusp x)
      (setq sign (- sign))
      (setq x (- x)))
    (when (minusp y)
      (setq sign (- sign))
      (setq y (- x)))
    (* sign x y)))
```

Figure 4: An example program.

any test suite that fails to exercise the condition where `y` is negative will fail to detect the bug in the next to last line of the function.

(As an example of the fact that covering all the conditions in a program does not guarantee that every facet of either the algorithm or the specification will be covered, consider the fact that the two test cases `(my* 2.1 3)` and `(my* -1/2 -1/2)` cover all four conditions. However, they do not detect the bug on the next to last line and they do not detect the fact that `my*` fails to work on complex numbers.)

The `COVER` system determines which conditions tested by a program are exercised by a given test suite. This is no substitute for thinking hard about the coverage of the test suite. However, it provides a useful starting point and can indicate some areas where additional test cases should be devised.

User's Manual for `COVER`

The functions, macros, and variables that make up the `COVER` system are in a package called `"COVER"`. The six exported symbols are documented below.

- `cover:annotate t-or-nil`

Evaluating `(cover:annotate t)` triggers the processing of function and macro definitions by the `COVER` system. Each subsequent instance of `defun` or `defmacro` is altered by adding annotation that maintains information about the various conditions tested in the body.

Evaluating `(cover:annotate nil)` stops the

special processing of function and macro definitions. Subsequent definitions are not annotated. However, if a function or macro that is currently annotated is redefined, the new definition is annotated as well.

The macro `cover:annotate` should only be used as a top-level form. When annotation is triggered, a warning message is printed, and `t` is returned. Otherwise, `nil` is returned.

```
(cover:annotate t) => t ; after printing:
;;; Warning: Coverage annotation applied.
```

- `cover:forget-all`

This function, which always returns `t`, has the effect of removing all coverage annotation from every function and macro. It is appropriate to do this before completely recompiling the system being tested or before switching to a different system to be tested.

- `cover:reset`

Each condition tested by an annotated function and macro is associated with a flag that trips when the condition is exercised. The function `cover:reset` resets all these flags, and returns `t`. It is appropriate to do this before re-running a test suite to reevaluate its coverage.

- `cover:report &key fn out all`

This function displays the information maintained by COVER, returning no values. *Fn* must be the name of an annotated function or macro. If *fn* is specified, a report is printed showing information about that function or macro only. Otherwise, reports are printed about every annotated function and macro.

Out, which defaults to `*standard-output*`, must either be an output stream or the name of a file. It specifies where the reports should be printed.

If *all*, which defaults to `nil`, is non-null then the reports printed contain information about every condition. Otherwise, the reports are abbreviated to highlight key conditions that have not been exercised.

- `cover:*line-limit*` default value 75

The output produced by `cover:report` is

```
(setq cover:*line-limit* 43) => 43
(cover:reset) => T
(cover:report) => ; after printing:
;- :REACH (DEFUN MY* (X Y)) <1>
(my* 2 2) => 4
(cover:report) => ; after printing:
;+ :REACH (DEFUN MY* (X Y)) <1>
;+ :REACH (WHEN (MINUSP X) (SETQ S <2>
; - :NON-NULL (MINUSP X) <4>
;+ :REACH (WHEN (MINUSP Y) (SETQ S <6>
; - :NON-NULL (MINUSP Y) <8>
(my* -2 2) => -4
(cover:report) => ; after printing:
;+ :REACH (DEFUN MY* (X Y)) <1>
;+ :REACH (WHEN (MINUSP Y) (SETQ S <6>
; - :NON-NULL (MINUSP Y) <8>
(cover:report :all t) => ; after printing:
;+ :REACH (DEFUN MY* (X Y)) <1>
;+ :REACH (WHEN (MINUSP X) (SETQ S <2>
;+ :NON-NULL (MINUSP X) <4>
;+ :NULL (MINUSP X) <5>
;+ :REACH (WHEN (MINUSP Y) (SETQ S <6>
; - :NON-NULL (MINUSP Y) <8>
;+ :NULL (MINUSP Y) <9>
```

Figure 5: Example COVER reports.

truncated to ensure that it is no wider than `cover:*line-limit*`.

An example. Suppose that the function `my*` in Figure 4 has been annotated and that no other functions or macros have been annotated. Figure 5 illustrates the operation of COVER and the reports printed by `cover:report`.

Each line in a report contains three pieces of information about a point in a definition: `+/-` specifying that the point either has (+) or has not (-) been exercised, a message indicating the physical and logical placement of the point in the definition, and in angle brackets `< >`, an integer that is a unique identifier for the point. Indentation is used to indicate that some points are subordinate to others in the sense that the subordinate points cannot be exercised without also exercising their superiors. The order of the lines of the report is the same as the order of the points in the definition.

Each message contains a label (e.g., `:REACH`, `:NULL`) and a piece of code. There is a point labeled `:REACH` corresponding to each definition as

a whole and each conditional form within each definition. Subordinate points corresponding to the conditions a conditional form tests are grouped under the point corresponding to the form. As discussed in detail in the next subsection, the messages for the subordinate points describe the situations in which the conditions are exercised. Lines that would otherwise be too long to fit on one line have their messages truncated (e.g., points <2> and <6> in Figure 5).

The first three reports in Figure 5 are abbreviated based on two principles. First, if a point p and all of its subordinates have been exercised, then p and all of its subordinates are omitted from the report. This is done to focus the user's attention on the points that have not been exercised.

Second, if a point p has not been exercised, then all of the points subordinate to it are omitted from the report. This reflects the fact that it is not possible for any of these subordinate points to have been exercised and one cannot devise a test case that exercises any of the subordinate points without first figuring out how to exercise p .

An additional complicating factor is that COVER operates in an incremental fashion and does not, in general, have full information about the subordinates of points that have not been exercised. As a result, it is not always possible to present a complete report. However, one can have total confidence that if the report says that every point has been exercised, this statement is based on complete information.

The first report in Figure 5 shows that none of the points within `my*` has been exercised. The second report displays most of the points in `my*`, to set the context for the two points that have not been exercised. The third report omits <2> and its subordinates, since they have all been exercised. The fourth report shows a complete report corresponding to the third abbreviated report.

- `cover:forget &rest ids`

This function gives the user greater control over the reports produced by `cover:report`. Each *id* must be an integer identifying a point.

All information about the specified points (and their subordinates) is forgotten. From the point of view of `cover:report`, the effect is as if the points never existed. (A forgotten point can be retrieved by reevaluating or recompiling the function or macro definition containing it.) The example below, which follows on after the end of Figure 5, shows the action of `cover:forget`.

```
(cover:forget 6) => T
(cover:report :all t) => ; after printing:
;+ :REACH (DEFUN MY* (X Y)) <1>
; + :REACH (WHEN (MINUSP X) (SETQ S <2>
; + :NON-NULL (MINUSP X) <4>
; + :NULL (MINUSP X) <5>
(cover:report) => ; after printing
;All points exercised.
```

The abbreviated report above does not describe any points, because every point in `my*` that has not been forgotten has been exercised. It is appropriate to forget a point if there is some reason that no test case can possibly exercise the point. However, it is much better to write your code so that every condition can be tested.

(Point numbers are assigned based on the order in which points are entered into COVER's database. In general, whenever a definition is reevaluated or recompiled, the numbers of the points within it change.)

The way conditionals are annotated. Figure 6 shows a file that makes use of COVER. Figure 7 shows the kind of report that might be produced by loading the file. Because, `maybe-` and `g` are the only definitions that have been annotated, these are the only definitions that are reported on. The order of the reports is the same as the order in which the definitions were compiled. The report on `g` indicates that the tests performed by `run-tests` exercise most of the conditions tested by `g`. However, they do not exercise the situation in which the `case` statement is reached, but neither of its clauses is selected.

There are no points within `maybe-`, because the code for `maybe-` does not contain any conditional forms. It is interesting to consider the precise points that COVER includes for `g`.

```
(in-package "USER")
(require "COVER" ...)
(defmacro maybe+ (x y)
  '(if (numberp ,x) (+ ,x ,y)))
(cover:annotate t)
(defmacro maybe- (x y)
  '(if (numberp ,x) (- ,x ,y)))
(defun g (x y)
  (cond ((and (null x) y) y)
        (y (case y
              (1 (maybe- x y))
              (2 (maybe+ x y))))))
(cover:annotate nil)
(defun h (x y) ...)
(cover:reset)
(run-tests)
(cover:report :out "report" :all t)
```

Figure 6: Example of a file using COVER.

When COVER processes a definition, a cluster of points is generated corresponding to each conditional form (i.e., `if`, `when`, `until`, `cond`, `case`, `typecase`, `and`, and `or`) that is literally present in the program. In addition, points are generated corresponding to conditional forms that are produced by macros that are annotated (e.g., the `if` produced by the `maybe-` in the first `case` clause in `g`). However, annotation is not applied to conditionals that come from other sources (e.g., from macros that are defined outside of the system being tested). These conditionals are omitted, because there is no reasonable way for the user to know how they relate to the code, and therefore there is no reasonable way for the user to devise a test case that will exercise them.

The messages associated with a point's subordinates describe the situations under which the subordinates are exercised. The pattern of messages associated with `case` and `typecase` is illustrated by the portion (reproduced below) of Figure 7 that describes the `case` in `g`.

```
; + :REACH (CASE Y (1 (MAYBE- X Y <13>
;   + :SELECT 1 <15>
;   + :SELECT 2 <16>
; - :SELECT-NONE <17>
```

```
; + :REACH (DEFMACRO MAYBE- (X Y)) <1>
; + :REACH (DEFUN G (X Y)) <2>
; + :REACH (COND ((AND # Y) Y) (Y ( <3>
;   + :REACH (AND (NULL X) Y) <9>
;   + :FIRST-NONE (NULL X) <11>
;   + :EVAL-ALL Y <12>
;   + :FIRST-NON-NONE (AND (NULL X) <5>
;   + :FIRST-NON-NONE Y <7>
;   + :REACH (CASE Y (1 (MAYBE- X Y <13>
;     + :SELECT 1 <15>
;     + :REACH (IF (NUMBERP X) (- X <18>
;       + :NON-NONE (NUMBERP X) <20>
;       + :NULL (NUMBERP X) <21>
;     + :SELECT 2 <16>
;     - :SELECT-NONE <17>
;   + :ALL-NONE <8>
```

Figure 7: The report created by Figure 6.

There are two subpoints corresponding to the two clauses of the `case`. In addition, since the last clause does not begin with `t` or `otherwise`, there is an additional point corresponding to the situation where none of the clauses of the `case` are executed.

The pattern of messages associated with a `cond` is illustrated by the portion (reproduced below) of Figure 7 that describes the `cond` in `g`.

```
; + :REACH (COND ((AND # Y) Y) (Y ( <3>
;   + :REACH (AND (NULL X) Y) <9>
;   + :FIRST-NON-NONE (AND (NULL X) <5>
;   + :FIRST-NON-NONE Y <7>
;   + :ALL-NONE <8>
```

There are subordinate points corresponding to the two clauses and the situation where neither clause is executed. There is also a point `<9>` corresponding to the `and` that is the predicate of the first `cond` clause. This point is placed directly under `<3>`, because it is not subordinate to any of the individual `cond` clauses.

The treatment of `and` (and `or`) is particularly interesting. Sometimes `and` is used as a control construct on a par with `cond`. In that situation, it is clear that `and` should be treated analogously to `cond`. However, at other times, `and` is used to compute a value that is tested by another conditional form. In that situation, COVER could choose to treat `and` as a simple function. However, it is nevertheless still reasonable to think of an `and` as having conditional points that correspond to different reasons why

the `and` returns a true or false value. It is wise to include tests corresponding to each of these different reasons.

The pattern of messages associated with an `and` is illustrated by the portion (reproduced below) of Figure 7 that describes the `and` in `g`.

```
(cover:report :all t)
; + :REACH (AND (NULL X) Y) <9>
; + :FIRST-NULL (NULL X) <11>
; + :EVAL-ALL Y <12>
```

The final subpoint corresponds to the situation where all of the arguments of the `and` have been evaluated. The `and` then returns whatever the final argument returned.

Figure 6 illustrates a batch-oriented use of `COVER`. However, `COVER` is most effectively used in an interactive way. It is recommended that you first create as comprehensive a test suite as you can and capture it using a tool such as `RT` [1]. The tests should then be run in conjunction with `COVER` and repeated reports from `COVER` generated as additional tests are created until complete coverage of conditions has been achieved. To robustly support this mode of operation, `COVER` has been carefully designed so that it will work with batch-compiled definitions, incrementally-compiled definitions, and interpreted definitions.

How `COVER` Works

The code for `COVER` is shown in Figures 8, 10, 11, and 13. Figure 8 shows the definition of the primary data structure used by `COVER` and some of the central operations. A `point` structure contains five pieces of information about a position in the code for a definition.

`hit` Flag indicating whether the point has been exercised.
`id` Unique integer identifier.
`status` Flag that controls reporting.
`name` Logical name.
`subs` List of subordinate points.

The `hit` flag operates as a “time stamp”. When a point is exercised, this is recorded by storing the current value of the variable `*hit*`

in the `hit` field of the point. This method of operation makes it possible to reset the `hit` flags of all the points currently in existence without visiting any of them (see the definition of `cover:reset`).

The `id` is printed in reports and used to identify points when calling `cover:forget`. The variable `*count*` is used to generate the values.

The `status` controls the reporting of a point. It is either `:SHOW` (shown in reports), `:HIDDEN` (not shown in reports, but its subordinates may be), or `:FORGOTTEN` (neither it nor its subordinates are shown in reports). (`cover:forget` changes the status of the indicated points to `:FORGOTTEN`.)

The `name` of a point `p` describes its position in the definition containing it. A `name` has the form: *(label code . superior-name)* where *label* is an explanatory label such as `:REACH` or `:NULL`, *code* is a piece of code, and *superior-name* is the `name` of the point containing `p` (if any). Taken together, the *label* and *code* indicate the position of `p` in a definition and the condition under which it is exercised (see the discussion of Figure 7).

At any given moment, the variable `*points*` contains a list of points corresponding to the annotated definitions known to `COVER`. (The function `cover:forget-all` resets `*points*` to `nil`.) As an illustration of the point data structure, Figure 9 shows the contents of `*points*` corresponding to the second report in Figure 5. It is assumed that `*hit*` has the value 1.

The function `add-top-point` adds a new top-level point corresponding to a definition to the list `*points*`. If there is already a point for the definition, the new point is put in the same place in the list.

The function `record-hit` records the fact that a point has been exercised. This may require locating the point in `*points*` using `locate` or adding the point into `*points*` using `add-point`. `record-hit` is optimized so that it is extremely fast when the point has already been exercised. This allows `COVER` to run with relatively little overhead. (The details of the way `record-hit` and `add-point` operate are discussed further in conjunction with Figure 13.)

```

(in-package "COVER" :use '("LISP"))
(provide "COVER")
(shadow '(defun defmacro))
(export '(annotate report reset forget
         forget-all *line-limit*))
(defstruct (point (:conc-name nil)
                 (:type list))
  (hit 0)
  (id nil)
  (status :show)
  (name nil)
  (subs nil))
(defvar *count* 0)
(defvar *hit* 1)
(defvar *points* nil)
(defvar *annotating* nil)
(defvar *testing* nil)
(lisp:defun forget (&rest ids)
  (forget1 ids *points*)
  t)
(lisp:defun forget1 (names ps)
  (dolist (p ps)
    (when (member (id p) names)
      (setf (status p) :forgotten)
      (forget1 names (subs p)))))
(lisp:defun forget-all ()
  (setq *points* nil)
  (setq *hit* 1)
  (setq *count* 0)
  t)
(lisp:defun reset () (incf *hit*) t)

(lisp:defun add-top-point (p)
  (setq p (copy-tree p))
  (let ((old (find (fn-name p) *points*
                  :key #'fn-name)))
    (cond (old (setf (id p) (id old)
                    (nsubstitute p old *points*)))
          (t (setf (id p) (incf *count*)
                  (setq *points*
                       (nconc *points*
                              (list p)))))))
  (list p))))
(lisp:defun record-hit (p)
  (unless (= (hit p) *hit*)
    (setf (hit p) *hit*)
    (let ((old (locate (name p))))
      (if old
          (setf (hit old) *hit*)
          (add-point p)))))
  (list p)))
(lisp:defun locate (name)
  (find name
        (if (not (cdr name))
            *points*
            (let ((p (locate (cdr name))))
              (if p (subs p))))
        :key #'name :test #'equal))
  (list p)))
(lisp:defun add-point (p)
  (let ((sup (locate (cdr (name p)))))
    (when sup
      (setq p (copy-tree p))
      (setf (subs sup)
            (nconc (subs sup) (list p)))
      (setf (id p) (incf *count*))
      (dolist (p (subs p))
        (setf (id p) (incf *count*))))))
  (list p)))

```

Figure 8: The basic data structure used by COVER.

```

((1 :SHOW 1 (#1=( :REACH (DEFUN MY* (X Y))))
  ((2 :SHOW 1 (#2=( :REACH (WHEN (MINUSP X) (SETQ SIGN (- SIGN)) (SETQ X (- X)))) #1#)
    ((3 :HIDDEN 1 ((:REACH (MINUSP X)) #2# #1#) NIL)
      (4 :SHOW 0 ((:NON-NULL (MINUSP X)) #2# #1#) NIL)
      (5 :SHOW 1 ((:NULL (MINUSP X)) #2# #1#) NIL))))
  (6 :SHOW 1 (#6=( :REACH (WHEN (MINUSP Y) (SETQ SIGN (- SIGN)) (SETQ Y (- X)))) #1#)
    ((7 :HIDDEN 1 ((:REACH (MINUSP Y)) #6# #1#) NIL)
      (8 :SHOW 0 ((:NON-NULL (MINUSP Y)) #6# #1#) NIL)
      (9 :SHOW 1 ((:NULL (MINUSP Y)) #6# #1#) NIL))))))

```

Figure 9: The contents of `*points*` corresponding to the second report in Figure 5.

Figure 10 shows the code that prints reports. As can be seen by a comparison of Figures 5 and 9, reports are a relatively straightforward printout of parts of `*points*` with nesting indicated by indentation and only the first part of each point's name shown. The function `report2` supports the abbreviation described in conjunction with Figure 5.

Annotating definitions. Figure 11 shows the code that controls the annotation of definitions by COVER. The first time `cover:annotate` is called, it uses `shadowing-import` to install new definitions for `defun` and `defmacro`. Whether or not annotation is in effect is recorded in the variable `*annotate*`. The variable `*testing*` is used to make it easier to test COVER using

```

(defvar *line-limit* 75)
(proclaim '(special *depth* *all*
                  *out* *done*))
(lisp:defun report
  (&key (fn nil)
        (out *standard-output*)
        (all nil))
  (let (p)
    (cond
     ((not (stream-p out))
      (with-open-file
       (s out :direction :output)
       (report :fn fn :all all :out s)))
     ((null *points*)
      (format out
              "~%No definitions annotated."))
     ((not fn)
      (report1 *points* all out))
     ((setq p (find fn *points*
                   :key #'fn-name))
      (report1 (list p) all out))
     (t (format out "~%~A is not annotated."
                  fn))))
  (values))
(lisp:defun fn-name (p)
  (let ((form (cadr (car (name p)))))
    (and (consp form)
         (consp (cdr form))
         (cadr form))))
(lisp:defun report1 (ps *all* *out*)
  (let ((*depth* 0) (*done* t))
    (mapc #'report2 ps)
    (when *done*
      (format *out*
              "~%;All points exercised."))))
(lisp:defun report2 (p)
  (case (status p)
    (:forgotten nil)
    (:hidden (mapc #'report2 (subs p)))
    (:show
     (cond ((reportable-subs p)
            (report3 p)
            (let ((*depth* (1+ *depth*)))
              (mapc #'report2 (subs p))))
           ((reportable p)
            (report3 p))))))
(lisp:defun reportable (p)
  (and (eq (status p) :show)
       (or *all*
           (not (= (hit p) *hit*)))))
(lisp:defun reportable-subs (p)
  (and (not (eq (status p) :forgotten))
       (or *all* (not (reportable p))
           (some #'(lambda (s)
                    (or (reportable s)
                        (reportable-subs s)))
                (subs p))))))
(lisp:defun report3 (p)
  (setq *done* nil)
  (let* ((*print-pretty* nil)
         (*print-level* 3)
         (*print-length* nil)
         (m (format nil
                    ";~V@T~:[-~;+~]~{ ~S~}"
                    *depth*
                    (= (hit p) *hit*)
                    (car (name p))))
         (limit (- *line-limit* 8)))
    (when (> (length m) limit)
      (setq m (subseq m 0 limit)))
    (format *out* "~%~A <~S>" m (id p))))

```

Figure 10: The code for the part of COVER that prints reports.

RT [1].

Redefining `defun` and `defmacro` is a convenient approach to use for supporting COVER, however, it is in general a rather dangerous thing to do. One problem is that for COVER to operate correctly, `cover:annotate` must be executed before any of the definitions you wish to annotate are read. For instance, Figure 6 would not work if an `eval-when` were wrapped around the top-level forms as a group.

When annotation is in effect, the new definitions of `defun` and `defmacro` use `sublis` to replace every instance of `if`, `cond`, etc. with special macros `c-if`, `c-cond`, etc. (see Figure 13). Defining forms created by the user (e.g., `def` in Figure 13) are typically macros that expand

into `defmacro`. They are indirectly supported by COVER, as long as their definitions are read after `cover:annotate` has been evaluated.

On the face of it, it is not correct to use `sublis` to rename forms in code, because every instance of the indicated symbols is changed, whether or not they are actually uses of the indicated forms and whether or not they are in quoted lists. Nevertheless, COVER uses `sublis` for two reasons.

First, in contrast to a code walker, `sublis` is very simple. (The only understanding of Lisp structure that COVER needs is how to separate the declarations from the body of a definition, see the function `parse-body`.)

Most problems can easily be avoided by resist-

```

(lisp:defmacro annotate (t-or-nil)
  '(eval-when (eval load compile)
    (annotate1 ,t-or-nil)))
(lisp:defun annotate1 (flag)
  (shadowing-import
   (set-difference '(defun defmacro)
    (package-shadowing-symbols *package*)))
  (when (and flag (not *testing*))
    (warn "Coverage annotation applied.")
    (setq *annotating* (not (null flag)))))
(lisp:defmacro defun (n argl &body b)
  (process 'defun 'lisp:defun n argl b))
(lisp:defmacro defmacro (n a &body b)
  (process 'defmacro 'lisp:defmacro n a b))
(lisp:defun parse-body (body)
  (let ((decls nil))
    (when (stringp (car body))
      (push (pop body) decls))
    (loop (unless (and (consp (car body))
                       (eq (caar body)
                           'declare))
            (return nil))
          (push (pop body) decls))
    (values (nreverse decls) body)))

(defvar *check*
  '((or . c-or) (and . c-and)
    (if . c-if) (when . c-when)
    (unless . c-unless)
    (cond . c-cond) (case . c-case)
    (typecase . c-typecase)))
(lisp:defun process (cdef def fn argl b)
  (if (not (or *annotating*
              (find fn
                   *points*
                   :key #'fn-name)))
      '(,def ,fn ,argl ., b)
      (multiple-value-bind (decls b)
        (parse-body b)
        (setq b (sublis *check* b))
        (let ((name
              '(:reach
                (,cdef ,fn ,argl))))
          '(eval-when (eval load compile)
            (add-top-point
             ',(make-point :name name)
             ,(def ,fn ,argl ,@ decls
                  ,(c0 (make-point :name
                                   name)
                       name b))))))))))

```

Figure 11: The code for the part of COVER that annotates definitions.

```

(EVAL-WHEN (EVAL LOAD COMPILE)
  (COVER::ADD-TOP-POINT '(NIL :SHOW 0 (#1=( :REACH (DEFUN MY* (X Y)))) NIL))
(LISP:DEFUN MY* (X Y)
  (COVER::RECORD-HIT '(NIL :SHOW 0 (#1#) NIL))
  (LET ((SIGN 1))
    (COVER::RECORD-HIT
     '(NIL :SHOW 0 (#2=( :REACH (WHEN (MINUSP X) (SETQ SIGN (- SIGN)) (SETQ X (- X)))) #1#)
       ((NIL :HIDDEN 0 ((:REACH (MINUSP X)) #2# #1#) NIL)
        (NIL :SHOW 0 ((:NON-NULL (MINUSP X)) #2# #1#) NIL)
        (NIL :SHOW 0 ((:NULL (MINUSP X)) #2# #1#) NIL))))))
    (IF (PROGN (COVER::RECORD-HIT '(NIL :HIDDEN 0 ((:REACH (MINUSP X)) #2# #1#) NIL))
          (MINUSP X))
        (PROGN (COVER::RECORD-HIT '(NIL :SHOW 0 ((:NON-NULL (MINUSP X)) #2# #1#) NIL))
              (SETQ SIGN (- SIGN)) (SETQ X (- X)))
        (PROGN (COVER::RECORD-HIT '(NIL :SHOW 0 ((:NULL (MINUSP X)) #2# #1#) NIL))
              NIL))
    ...)))

```

Figure 12: Part of the annotated definition of `my*` from Figure 4.

ing the temptation to use `if`, `cond`, etc. as variable names. Any remaining difficulties can be tolerated because COVER is merely part of scaffolding for testing a system rather than part of the system to be delivered. A subtle difficulty concerns `and` and `or`. They are used as type specifiers as well as conditional forms. This difficulty is partly overcome by the type definitions at the end of Figure 13.

Second, the use of `sublis` supports two key features of COVER that would be very difficult to support using a code walker. It insures that only conditional forms that literally appear in the definition are annotated (as opposed to ones that come from macro expansions), and yet, conditionals that come from the expansion of annotated macros are annotated. (Note that the literals that turn into conditionals in the

```

(defvar *fix*
  '((c-or . or) (c-and . and) (c-if . if)
    (c-when . when) (c-unless . unless)
    (c-cond . cond) (c-case . case)
    (c-typecase . typecase)))

(proclaim '(special *subs* *sup*))

(lisp:defmacro sup-mac () nil)

(lisp:defmacro def (name args form)
  '(lisp:defmacro ,name (&whole w ,@ args
    &environment env)
    (let* ((*subs* nil)
           (*sup*
            '(:reach ,(sublis *fix* w)
              .,(macroexpand-1
                 (list 'sup-mac) env))))
      (p (make-point :name *sup*))
        (form ,form))
      (setf (subs p) (nreverse *subs*))
      (c0 p *sup* (list form))))))

(lisp:defmacro c (body &rest msg)
  (c1 '(list ,body) msg :show))

(lisp:defmacro c-hide (b)
  (c1 '(list ,b) (list :reach b) :hidden))

(eval-when (eval load compile)

(lisp:defun c1 (b m s)
  '(let ((n (cons (sublis *fix*
                     (list .,m)
                     *sup*)))
        (push (make-point :name n :status ,s)
              *subs*))
      (c0 (make-point :name n :status ,s)
          n ,b)))

(lisp:defun c0 (p sup b)
  '(macrolet ((sup-mac () ',sup))
    (record-hit ',p
      .,b)) )

(def c-case (key &rest cs)
  '(case ,(c-hide key)
    .,(c-case0 cs)))

(def c-typecase (key &rest cs)
  '(typecase ,(c-hide key)
    .,(c-case0 cs)))

(lisp:defun c-case0 (cs)
  (let ((stuff (mapcar #'c-case1 cs)))
    (when (not (member (caar (last cs))
                       '(t otherwise)))
      (setq stuff
        (nconc stuff
          '((t ,(c nil :select-none))))))
      stuff))

(lisp:defun c-case1 (clause)
  '(,(car clause)
    ,(c '(progn ., (cdr clause)) :select
      (car clause))))

(def c-if (pred then &optional (else nil))
  '(if ,(c-hide pred)
    ,(c then :non-null pred)
    ,(c else :null pred)))

(def c-when (pred &rest actions)
  '(if ,(c-hide pred)
    ,(c '(progn ., actions)
      :non-null pred)
    ,(c nil :null pred)))

(def c-unless (pred &rest actions)
  '(if (not ,(c-hide pred))
    ,(c '(progn ., actions) :null pred)
    ,(c nil :non-null pred)))

(def c-cond (&rest cs)
  (c-cond0 (gensym) cs))

(lisp:defun c-cond0 (var cs)
  (cond ((null cs) (c nil :all-null))
        ((eq (caar cs) t)
         (c (if (cdar cs)
                '(progn .,(cdar cs))
                t)
            :first-non-null t))
        ((cdar cs)
         '(if ,(c-hide (caar cs))
           ,(c '(progn .,(cdar cs))
             :first-non-null
             (caar cs))
           ,(c-cond0 var (cdr cs))))))
  (t '(let ((,var
            ,(c-hide (caar cs))))
      (if ,var
        ,(c var :first-non-null
          (caar cs))
        ,(c-cond0 var
          (cdr cs)))))))

(def c-or (&rest ps) (c-or0 ps))

(lisp:defun c-or0 (ps)
  (if (null (cdr ps))
    (c (car ps) :eval-all (car ps))
    (let ((var (gensym)))
      '(let ((,var ,(c-hide (car ps))))
        (if ,var
          ,(c var :first-non-null
            (car ps))
          ,(c-or0 (cdr ps)))))))

(def c-and (&rest ps)
  '(cond .,(maplist #'c-and0
    (or ps (list t))))

(lisp:defun c-and0 (ps)
  (if (null (cdr ps))
    '(t ,(c (car ps) :eval-all (car ps)))
    '((not ,(c-hide (car ps)))
      ,(c nil :first-null (car ps))))))

(deftype c-and (&rest b) '(and ., b))

(deftype c-or (&rest b) '(or ., b))

```

Figure 13: The code for the part of COVER that annotates conditionals.

code generated by a macro are quoted in the body of the macro.)

Figure 12 shows part of the results of annotating the function `my*` from Figure 4. The annotated definition is preceded by a call on `add-top-point`, which enters a point describing the definition into `*points*`. Within the definition, calls on `record-hit` are introduced at strategic locations. Each call contains a quoted point that is essentially a template for what should be introduced into `*points*`. The first `when` in `my*` is converted into an `if` that has cases corresponding to the success and failure of the predicate tested by the `when`. The call on `record-hit` that precedes this `if` contains a point with subpoints that establishes the cases of the `if`. This ensures that both cases of the `if` will be present in `*points*` as soon as the `if` is exercised, even if only one of the cases is exercised.

The hidden point associated with the predicate tested by the `when` establishes an appropriate context for points within the predicate itself. It is unnecessary in this example, because there are no such points. In the `cond` in the function `g` in Figure 6, a similar hidden point associated with the first predicate tested serves to correctly position the points associated with the `and` (see Figure 7).

For the most part, the macros in Figure 13 operate in straightforward ways to generate annotated conditionals. However, `def`, `c`, `c1`, and `c0` interact in a somewhat subtle way using `macrolet` to communicate the name of a superior point to its subordinates. This could have been done more simply with `compiler-let`; however, `compiler-let` is slated to be removed from Common Lisp.

Underlying approach. The annotation scheme used by `COVER` is designed to meet two goals. First, it must introduce as little overhead as possible when the annotated function runs. (It does not matter if the process of inserting annotation is expensive and it does not matter if the process of printing reports is expensive. It does not even matter if processing is relatively expensive the first time a point is ex-

ercised. However, it is essential that processing be very fast when an exercised point is exercised a second time.)

Second, the scheme must work reliably with interpreted code, with compiled code loaded from files, and with code that is incrementally compiled on the fly. This introduces a number of strong constraints. In particular, you cannot depend on using some descriptive data structure built up during compilation, because you cannot assume that compilation will occur. On the other hand, if you use quoted data structures as in Figure 12, you cannot make any assumptions about what sharing will exist or whether they will be copied, because some Lisp compilers feel free to make major changes in quoted lists.

To achieve high efficiency, `record-hit` (see Figure 8) alters its argument by side-effect to mark it exercised. Side-effecting a compiled constant is inherently dangerous, but is relatively safe here, because the changed value is an integer, and the point data structure cannot be shared with any other point data structure, because no two points can have the same `name`.

The first time a given call on `record-hit` is encountered, it enters the point which is its argument into `*points*`. This is done by first looking to see if the point is already there (e.g., because it was entered by an `add-top-point` or is a subordinate point that was explicitly entered as part of its superior point). If it is not there, it is copied and inserted as a subordinate point of the appropriate superior point. (By this process, `*points*` is dynamically built up in exactly the same way when executing interpreted and compiled code.) If the superior point cannot be found, nothing is done. (This can only happen when the annotation of the currently executing function has been forgotten.)

The second time a call on `record-hit` is encountered the only thing it has to do is check that the point has been exercised. If it has, nothing needs to be done. If a `cover:reset` has been done, then the check will fail, and `record-hit` relocates the point in `*points*`, and sets the `hit` flag. (This second lookup could be avoided

if the quoted point had been directly inserted into `*points*` instead of copied. However, this is unsafe for two reasons. First, the sharing would mean that side-effects to `*points*` would translate into side-effects to compiled list constants. This will cause many Lisp systems to blow up in unexpected ways. Second, in some Lisp systems compiling an interpreted function can cause the quoted lists in it to be copied. As a result, you cannot depend that any sharing set up between a global data structure and quoted constants will be preserved.)

The operation of `COVER` requires that each point be given a unique identifying name. The naming scheme used assumes that a given conditional form will not have two predicates that are `equal` and that a chunk of straightline code will not contain two conditional forms that are `equal`. If this assumption is violated, `COVER` will merge the two resulting points into one.

The power of Lisp. `COVER` is a good example of the power of Lisp as a tool for building programming environments. Because Lisp contains a simple representation for Lisp programs, it is easy to write systems that convert programs into other programs. Because Lisp encompasses both the language definition and the run-time environment, it is easy to write systems that both manipulate the language and extend the run-time environment. Systems like `COVER` are regularly written for C and other Algol-like languages; however, this is much harder to do than in Lisp.

Acknowledgments

The concept of code coverage is an old one, which is used by many (if not most) large programming organizations. `COVER` is the result of several years of practical use and evolution.

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 COVER

`COVER` is written in portable Common Lisp and has been tested in several different Common Lisp implementations. The full source for `COVER` is shown in Figures 8, 10, 11, and 13. In addition, the source can be obtained over the INTERNET by using FTP. Connection should be made to `FTP.AI.MIT.EDU` (INTERNET number 128.52.32.6). Login as “anonymous” and copy the files shown below.

```
In the directory /pub/lptrs/
cover.lisp           source code
cover-test.lisp     test suite
cover-doc.txt       brief documentation
```

The contents of Figures 8, 10, 11, and 13 and the files above are copyright 1991 by the Massachusetts Institute of Technology, Cambridge MA. Permission to use, copy, modify, and distribute this software for any purpose and without fee is hereby granted, provided that this copyright and permission notice appear in all copies and supporting documentation, and that the names of MIT and/or the author are not used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. MIT and the author make no representations about the suitability of this software for any purpose. It is provided “as is” without express or implied warranty.

MIT and the author disclaim all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall MIT or the author be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

References

- [1] R.C. Waters, “Supporting the Regression Testing of Lisp Programs”, *ACM Lisp Pointers*, 4(2):47–53, June 1991.

3. Implementing Queues in Lisp

Richard C. Waters and Peter Norvig

A queue is a data structure where items are entered one at a time and removed one at a time in the same order—i.e., first in first out. They are the same as stacks except that in a stack, items are removed in the reverse of the order they are entered—i.e., last in first out. Queues are most precisely described by the functions that act on them:

- (**make-queue**) Creates and returns a new empty queue.
- (**queue-elements** *queue*) Returns a list of the elements in *queue* with the oldest element first. The list returned may share structure with *queue* and therefore may be altered by subsequent calls on **enqueue** and/or **dequeue**.
- (**empty-queue-p** *queue*) Returns **t** if *queue* does not contain any elements and **nil** otherwise.
- (**queue-front** *queue*) Returns the oldest element in *queue* (i.e., the element that has been in the *queue* the longest). When *queue* is empty, the results are undefined.
- (**dequeue** *queue*) *Queue* is altered (by side-effect) by removing the oldest element in *queue*. The removed element is returned. When *queue* is empty, the results are undefined.
- (**enqueue** *queue* *item*) *Queue* is altered (by side-effect) by adding the element *item* into *queue*. The return value (if any) is undefined.

```
(empty-queue-p (setq q (make-queue))) => t
(progn (enqueue q 'a) (enqueue q 'b) (queue-front q)) => a
(progn (enqueue q 'c) (enqueue q 'd) (dequeue q)) => a
(queue-elements q) => (b c d)
```

Having **enqueue** and **dequeue** alter *queue* by side-effect is convenient for most uses of queues and allows for efficient implementations. However, it means that care must be taken when queues are manipulated. For instance, if the output of **queue-elements** must be preserved beyond a subsequent use of **enqueue** or **dequeue** it must be copied (e.g., with **copy-list**).

Queues Implemented With Lists

Lisp's eponymous data structure, the list, can be used to represent a wide variety of data structures including queues. The implementation of queues in Figure 14 represents a queue as a cons cell whose car is a list of the elements in the queue, ordered with the oldest first.

The implementation in Figure 14 is simple and easy to understand. The close similarity of queues and stacks is highlighted by the fact that **dequeue** is implemented using **pop** and **enqueue** is implemented in a way that is very similar to **push**.

The one thing that may not be immediately clear about the implementation in Figure 14 is the reason why a header cell is necessary, instead of just using the list of elements in the queue to represent the queue. The header cell is needed so that an element can be added into an empty queue (and the last element removed from a one-element queue) purely by side-effect. For this to work, an empty queue must be some kind of mutable structure that can be pointed to (e.g., not

```

(defun make-queue () (list nil))
(defun queue-elements (q) (car q))
(defun empty-queue-p (q) (null (car q)))
(defun queue-front (q) (caar q))
(defun dequeue (q) (pop (car q)))
(defun enqueue (q item) (setf (car q) (nconc (car q) (list item))))
(setq q (make-queue)) => (nil)
(progn (enqueue q 'a) (enqueue q 'b) (enqueue q 'c) q) => ((a b c))

```

		;space	time
		; 2	2
		; 2	2
		; 4	4
		; 4	O(n)

Figure 14: Queue implementation using `nconc`.

just `nil`).

The functions in Figure 14 are divided into two groups to reflect the fact that the last four functions are called much more often than the first two. As a result, it is more important that they be efficient.

The first column of numbers on the right of Figure 14 shows the size of the code required if the corresponding function is compiled in line at the point of use. The size is specified as the number of primitive operations (`car`, `cdr`, `cons`, `list`, `null`, `rplaca`, `rplacd`, `setq`, branching, generating a constant `nil`, and calling an out-of-line function) that are necessary. For instance, `dequeue` requires 4 basic operations (a `car`, two `cdrs` and a `rplacd`).

The space numbers cannot be taken as exactly reflecting any particular Lisp implementation, because a given Lisp compiler may create code that performs unnecessary operations, and a given hardware platform may require multiple instructions to support some of the primitive operations. However, this does not matter a great deal, because the relative code size of functions is the key thing that is important in the context of this paper. (The validity of the numbers in Figure 14 as a basis for this kind of comparison has been verified by looking at the code produced by the compilers for the TI Explorer and the Symbolics Lisp Machine.)

An important virtue of the implementation of queues in Figure 14 is that the functions are coded compactly enough that it is practical to compile all of them in line (i.e., by declaring them `inline`). In most Common Lisp implementations, this is significantly more efficient than using out-of-line function calls.

The second column of numbers on the right of Figure 14 shows the number of basic operations that have to be executed at run time. If there is any branching required, the number reflects the control path that is most likely to be taken. These numbers reveal that there is a problem with the implementation. Most of the functions have small fixed costs that are independent of the size of the queue. However, the time required to perform the `nconc` in `enqueue` is proportional to the size of the queue.

Keeping a Pointer to the End of the Queue

The problem with `nconc` is not that it makes an expensive change (it merely performs one `rplacd`), but that it has to search down the entire list to locate the `cons` cell containing the last queue element. This inefficiency can be overcome by maintaining a pointer to the end of the list of queue elements.

In particular, BBN Lisp supported a queue data structure exactly like the one in Figure 14 except that the `cdr` of the header cell was used as a pointer to the list cell containing the last element in the queue (if any). Using this pointer, the six queue functions can be supported as shown in Figure 15. (In BBN Lisp, the function `enqueue` was called `tconc`.)

The only difference between Figure 15 and Figure 14 is in the implementation of `enqueue`. It

```

(defun make-queue () (list nil))
(defun queue-elements (q) (car q))
(defun empty-queue-p (q) (null (car q)))
(defun queue-front (q) (caar q))
(defun dequeue (q) (pop (car q)))
(defun enqueue (q item)
  (let ((new-last (list item)))
    (if (null (car q))
        (setf (car q) new-last)
        (setf (caddr q) new-last))
      (setf (cdr q) new-last)))
(setq q (make-queue)) ⇒ (nil)
(progn (enqueue q 'a) (enqueue q 'b) (enqueue q 'c) q) ⇒ ((a b . #1=(c)) . #1#)
;space time
; 2 2
; 2 2
; 4 4
; 9 8

```

Figure 15: Simple queue implementation using an end pointer.

is transformed into a constant-time operation and is therefore very much faster. Unfortunately, `enqueue` is now too large to be comfortably compiled in line.

The implementation of `enqueue` in Figure 15 is larger than in Figure 14 primarily because it has to test for a special boundary condition. When the input queue is empty, `enqueue` has to do a `rplaca` to insert the (one element) list of queue elements in the car of the header cell; otherwise it has to do a `rplacd` to extend the list of queue elements.

Moving the Boundary Test to a Better Place

It is possible to remove the boundary test from `enqueue` by rearranging the queue data structure as follows. First, the two components of the header cell are interchanged, putting the pointer to the end of the queue in the car. Second, a convention can be adopted that an empty queue's end pointer points to the queue itself. These two changes allow the same code to be used for inserting an element into a queue whether or not the queue is empty, see Figure 16.

Unfortunately, while the two changes above simplify `enqueue`, they make it more difficult to implement `dequeue`. The problem is that `dequeue` now has a special boundary condition to test for—if the queue becomes empty, the queue's last pointer has to be made to point to the queue itself. However, because this is a simpler special case than the one in `enqueue` in Figure 15, it does not lead to as much overhead. Also, since some applications do significantly more `enqueues` than `dequeues` and no application does more `dequeues`, the trade-off is worthwhile.

The implementation approach in Figure 16 takes subtle advantage of the typeless nature of

```

(defun make-queue () (let ((q (list nil))) (setf (car q) q)))
(defun queue-elements (q) (cdr q))
(defun empty-queue-p (q) (null (cdr q)))
(defun queue-front (q) (cadr q))
(defun dequeue (q)
  (let ((elements (cdr q)))
    (unless (setf (cdr q) (cdr elements))
      (setf (car q) q))
    (car elements)))
(defun enqueue (q item) (setf (car q) (setf (cdar q) (list item)))) ; 4 4
(setq q (make-queue)) ⇒ #1=(#1#)
(progn (enqueue q 'a) (enqueue q 'b) (enqueue q 'c) q) ⇒ (#1=(c) a b . #1#)
;space time
; 2 2
; 2 2
; 7 6

```

Figure 16: A compact and efficient queue implementation.

```

(defun make-queue () (let ((q (list nil))) (cons q q)))
(defun queue-elements (q) (cdar q))
(defun empty-queue-p (q) (null (cdar q)))
(defun queue-front (q) (cadar q))
(defun dequeue (q) (car (setf (car q) (cdar q))))
(defun enqueue (q item) (setf (cdr q) (setf (caddr q) (list item))))
;space time
; 3 3
; 3 3
; 4 4
; 4 4
(setq q (make-queue)) => (#1=(nil) . #1)
(progn (enqueue q 'a) (enqueue q 'b) (enqueue q 'c) q) => ((nil a b . #1=(c)) . #1#)

```

Figure 17: Another compact and efficient queue implementation.

Lisp. In most other languages, the header cell for a queue would be a different type of structure from the cells forming the linked list of queue elements. This would block `enqueue` from treating the `cdr` of the header cell the same as the `cdr` of a linked list cell. (In some languages, this problem could be overcome by judicious use of type unioning or type-check bypassing.)

Eliminating the Boundary Test by Adding a Cell

A different way to improve on Figure 15 is to eliminate the need for any boundary tests at all, by adding a dummy cell into the list holding the elements in the queue as shown in Figure 17. This allows `enqueue` and `dequeue` to operate essentially as if the queue were never empty. However, the other functions have to be adjusted to skip over the dummy cell, and therefore become a bit longer.

Whether or not the implementation in Figure 17 is better than the one in Figure 16 depends on the details of your Lisp implementation and which queue operations you use most. For instance, if calls on `dequeue` are particularly infrequent (e.g., because a list of the items queued is the primary result desired), then the implementation in Figure 16 is better. In contrast, if the Lisp Implementation has special hardware support for following chains of pointers through cons cells (e.g., the TI Explorer), Figure 17 is better.

Queues Implemented With Vectors

Lists are a convenient basis for queues. In particular, the interaction of `cons` and garbage collection provides support for queues of unbounded length without any special provisions having to be made. However, list-based implementations are wasteful of memory, because an entire cons cell has to be used to store each element in the queue, and as elements are enqueued and dequeued, new cons cells continually have to be allocated.

Memory efficient implementations of queues are possible using vectors. This approach is often taken in other languages. Figure 18 shows an implementation like those usually shown in introductory data-structure texts. The basic approach is to store the elements of a queue as a section of a vector treated as a ring. The elements are stored in reverse order in the vector so that a comparison with zero can be used to detect when either the front or end pointers reach the edge of the vector.

The primary advantage of a vector-based implementation is that it requires only about half the memory to store the contents of the queue. If the queue elements are shorter than a word (e.g., characters or bits) even more savings are possible. In addition, enqueueing and dequeuing elements does not generate any garbage at all (unless the queue size gets so large that an enlarged vector has to be allocated).

The primary disadvantage of a vector-based implementation is that it is more complicated. In particular, it has to do all its own memory management. This means that the queue still takes up a lot of space even when it is empty. In addition, provision has to be made for extending the vector holding the queue if it becomes full. (In figure 18, this is supported by the function `extend-queue`

```

(defstruct q front end size elements)
(defun make-queue (&optional (size 20))
  (make-q :front (1- size) :end (1- size) :size size
         :elements (make-sequence 'simple-vector size)))
(defun queue-elements (q)
  (when (not (empty-queue-p q))
    (do ((i (1+ (q-end q)) (1+ i))
        (result nil)
        (nil)
        (when (= i (q-size q)) (setq i 0))
        (push (svref (q-elements q) i) result)
        (when (= i (q-front q)) (return result))))))
;space time
(defun empty-queue-p (q) (= (q-front q) (q-end q))) ; 3 3
(defun queue-front (q) (svref (q-elements q) (q-front q))) ; 3 3
(defun dequeue (q) ; 7 6
  (let ((front (q-front q)))
    (prog1 (svref (q-elements q) front)
      (when (zerop front) (setq front (q-size q)))
      (setf (q-front q) (1- front)))))
(defun enqueue (q item) ; 10 8
  (let ((end (q-end q)))
    (setf (svref (q-elements q) end) item)
    (when (zerop end) (setq end (q-size q)))
    (when (= (setf (q-end q) (1- end)) (q-front q)) (extend-queue q))))
(defun extend-queue (q)
  (let* ((elements (q-elements q))
        (size (q-size q))
        (new-size (* 2 size))
        (divide (1+ (q-front q)))
        (new-end (+ divide size -1))
        (new (make-sequence 'simple-vector new-size)))
    (replace new elements :end2 divide)
    (replace new elements :start1 (1+ new-end) :start2 divide)
    (setf (q-elements q) new)
    (setf (q-end q) new-end)
    (setf (q-size q) new-size)))
(progn (setq q (make-queue))
  (dotimes (i 17) (enqueue q '-) (dequeue q))
  (dotimes (i 5) (enqueue q i) q)
  => #S(queue front 17 end 2 size 20 elements
        #(2 1 0 - - - - - 4 3))

```

Figure 18: A traditional vector-based queue implementation.

and a fullness test in `enqueue`.)

Whenever possible, it is good to start the queue at a size that is sufficient to hold the maximum expected size, rather than starting at an arbitrary size like 20. For this reason the function `make-queue` is extended by giving it an optional size argument. Given firm maximum-size information one could go further and dispense with `extend-queue` and the fullness test in `enqueue`. However, this is a dangerous practice and saves relatively little.

It is worthy of note that it would be a mistake to use an adjustable array in the queue data structure. This would make extending the array a little bit easier, but would slow up all of the other operations on the vector. Adjustable arrays are only helpful when there may be many pointers directly to the array that has to be extended. Whenever, as here, there is known to be only one pointer, it is much better to change the pointer to point to a new array, than to extend the array itself.

```

(defstruct q front end size elements)
(defun make-queue (&optional (size 20))
  (make-q :front (- size 1) :end (- size 1) :size size
         :elements (make-sequence 'simple-vector size)))
(defun queue-elements (q)
  (do ((i (1+ (q-end q)) (1+ i))
      (result nil))
      ((> i (q-front q)) result)
      (push (svref (q-elements q) i) result)))
(defun empty-queue-p (q) (= (q-front q) (q-end q)))
(defun queue-front (q) (svref (q-elements q) (q-front q)))
(defun dequeue (q)
  (progn (svref (q-elements q) (q-front q)) (decf (q-front q))))
(defun enqueue (q item)
  (setf (svref (q-elements q) (q-end q)) item)
  (when (minusp (decf (q-end q))) (shift-queue q)))
(defun shift-queue (q)
  (let* ((elements (q-elements q))
        (new elements))
    (when (> (q-front q) (/ (q-size q) 2))
      (setq new (make-sequence 'simple-vector (* 2 (q-size q))))
      (setf (q-elements q) new)
      (setf (q-size q) (* 2 (q-size q))))
    (setf (q-end q) (- (q-size q) 2 (q-front q)))
    (replace new elements :start1 (1+ (q-end q)))
    (setf (q-front q) (1- (q-size q)))))
(progn (setq q (make-queue))
  (dotimes (i 17) (enqueue q '-) (dequeue q))
  (dotimes (i 5) (enqueue q i) q))
⇒ #S(queue front 19 end 14 size 20 elements
    #(2 1 0 - - - - - - - - - - 4 3 2 1 0))

```

Figure 19: A faster vector-based queue implementation.

Another problem is that `queue-elements` becomes an $O(n)$ operation, since it has to copy the queue contents into a list. If you want to be able to easily get a list of the elements in a queue, it is better to start with a list-based implementation.

A final problem with Figure 18 is the inefficiency of some of the key operations. The functions `empty-queue` and `queue-front` are small and could be coded in line. However, `dequeue` is on the borderline in size and `enqueue` is quite large.

Shifting Is Better Than Using a Ring

Figure 19 shows the kind of improvements that can be obtained using a little ingenuity. The key difference between Figure 19 and Figure 18 is that the implementation does not treat the vector as a ring. Rather, whenever the queue reaches the end of the vector, it is shifted over (by the function `shift-queue`, which also extends the vector if necessary).

One might well imagine that operating on the vector as a ring had to be better than shifting everything over every time the queue reaches the edge of the vector. However, as long as the queue is significantly shorter than the vector (say only $2/3$ the length or less) then shifting does not have to occur very often, and performing occasional shifts ends up being cheaper than complex decrementing of the pointers all of the time. `Dequeue` and `enqueue` both become significantly more efficient, and `dequeue` becomes short enough to easily code in line.

All in all, except for the fact that the queue structure has to be a bit bigger for things to work

out efficiently, the implementation in Figure 19 is better than the one in Figure 18 in all respects. Given its memory efficiency and quite reasonable speed, it is worth considering Figure 19 as an alternative to a list-based implementation in any situation where the function `queue-elements` is not used.

Conclusion

Lisp provides an all-purpose data structure—the list—which is often adequate for rapid prototyping. But when an efficient solution is required, Lisp programmers must choose their data structures carefully. Figures 16–19 show two efficient list based implementations of queues and two efficient vector-based implementations. Which is appropriate to use depends on the details of the exact situation in question.

The various implementations presented above illustrate several general issues to keep in mind when seeking efficient algorithms. Introducing alignments of components can often eliminate special cases (e.g., the way the queue data structure is rearranged in Figure 16). Sometimes a computation can be moved from an expensive context to a less expensive one (e.g., moving the boundary test from `enqueue` to `dequeue` in Figure 16). Many times, it is better to do a little extra work all the time, then do an expensive check to determine when extra work is really needed (e.g., indexing through the extra cell in Figure 17 is better in many situations than testing for whether the list is empty). Other times, it is better to introduce extra work some of the time to eliminate a steady background of work (e.g., occasional wholesale shifting in Figure 19 is better than continual performing complex pointer stepping). Slimming functions down to in-line-able size can pay big pragmatic dividends. Above all, the only way to get a really efficient algorithm is to experiment with many alternatives.