```
;; This is a collection of tests from the ErrorSystem.NoteFile.  It tests Xerox extensions to the
CommonLisp ErrorSystem mostly dealing with proceed cases.  The individual test files for each of
the functions have been appended together in this big file to gain diagnostic information by
testing the functions in a particular order.  Nested proceed-cases use find-restart and so find
should come first.
;;
;; The source for the text file listing is the NoteCards database at
{eris}<lispcore>cml>test>ErrorSystem.NoteFile.  Changes are made only to the NoteFile.  The
listing is
;; Filed As:  {eris}<lispcore>cml>test>24-ErrorSystem.x
;;
;;
(do-test "define our-little-condition" (define-condition our-little-condition (condition)))
;; Definition To Be Tested: ignore-errors
;;
;; Source:        Xerox LIsp Manual
;; Chapter 24: ERROR SYSTEM    Handling Conditions
;;
;; Created By:    Kirk Kelley
;;
;; Creation Date:  21 November 86
;;
;; Last Update:   >> day month << 86
;;
;; Filed As:      {eris}<lispcore>cml>test>24-ignore-errors.x
;;
;;
;; Syntax: ignore-errors &body forms  [Macro]
;;
;; Function Description: Executes its body in a context which handles errors of type error by
returning control to this form. If no error is signalled, all values returned by the last form
are returned by ignore-errors. Otherwise, the form returns nil and the condition that was
signalled. Synonym for (condition-case (progn . forms) (error () nil)).
;;
;; Argument(s):  forms
;;
;; Returns: nil if error followed by the signalled condition, else value(s) of last form
;;
(do-test-group "ignore-errors"
           (do-test "ignore-errors with simple error" (not (ignore-errors (error))))
           (do-test "ignore-errors no error"
                 (and (string-equal "success" (ignore-errors "success"))
                     (ignore-errors (signal 'simple-condition))))
           (do-test "ignore-errors cerror" (not (ignore-errors (cerror))))
           (do-test "ignore-errors second return no error"
                 (second (multiple-value-list (ignore-errors (values-list (list nil t))))))
           (do-test "ignore-errors second return error"
                 (second (multiple-value-list (ignore-errors (error))))))
;; Definition To Be Tested: find-restart
;;
;; Source:        Xerox LIsp Manual
;; Chapter 24: ERROR SYSTEM    Proceeding from Conditions
;;
;; Created By:    Kirk Kelley
;;
;; Creation Date:  21 November 86
;;
;; Last Update:   >> day month << 86
;;
;; Filed As:      {eris}<lispcore>cml>test>24-find-restart.x
;;
;;
;; Syntax: find-restart name
;;
;; Function Description: Searches for a proceed case by the given name which is applicable to the
given condition in the current dynamic contour.  If name is a proceed function name, then the
innermost (ie, most recently established) proceed case with that function name that matches the
given condition is returned.  nil is returned if no such proceed case is found.  If name is a
proceed case object, then it is simply returned unless it is not currently valid for use.  In
that case, nil is returned.
;;
;; Argument(s):  name -- a proceed function name or
;;                               a proceed case object
;;
;; Returns: proceed-case, proceed case object, or nil
;;
;; The simple tests for this fall out of compute-proceed-cases.
;;
;;
;;
(do-test-group "find-restart"
```

```
(do-test "find-restart nil 1" (not (find-restart 'none)))
(do-test "find-restart nil 2"
      (not (or (find-restart 'none)
               (restart-case (find-restart 'none) (use-food)))))
(do-test "find-restart nil 3"
      (not (restart-case (find-restart 'proceed) (use-food))))
(do-test "find-restart nil switched"
      (not (restart-case (find-restart 'use-food) (proceed))))
(do-test "typep find-restart"
      (restart-case (typep (find-restart 'use-food) 'restart) (use-food)))
(do-test "restart-case signal positive "
    (restart-case
        (condition-case (signal (make-condition 'our-little-condition))
                    (our-little-condition nil (find-restart 'use-food)))
         (use-food)))
(do-test "find-restart nil :condition"
      (not (restart-case (find-restart 'use-food)
                     (use-food nil :condition our-little-condition nil))))
(do-test "find-restart nested inner"
    (define-proceed-function use-food :report "Select this food.")
    (restart-case
        (restart-case
            (and (setq our-restart-case
                       (find-restart 'use-food))
                 (typep our-restart-case 'restart)
                 (string-equal "The inner case."
                               (princ-to-string our-restart-case)))
            (use-food nil :report "The inner case." t)) (use-food)))
(do-test "find-restart nested outer"
    (restart-case
        (progn (and (test-setq our-restart-case
                               (find-restart 'use-food))
                    (typep our-restart-case 'restart)
                    (string-equal "Select this food."
                                  (princ-to-string our-restart-case)))
               (restart-case (find-restart 'use-food)
                             (use-food nil :report "The inner case." t))
               (and (test-setq our-restart-case
                               (find-restart 'use-food))
                    (typep our-restart-case 'restart)
                    (string-equal "Select this food."
                                  (princ-to-string our-restart-case))))
        (use-food))))
(do-test-group "old-style find-restart"
    (do-test "old style find-restart nil 2"
      (not (or (find-restart 'none)
               (proceed-case (find-restart 'none) (use-food)))))
    (do-test "old style find-restart nil 3"
      (not (proceed-case (find-restart 'proceed) (use-food))))
    (do-test "old style find-restart nil switched"
      (not (proceed-case (find-restart 'use-food) (proceed))))
    (do-test "old style find-restart positive"
      (proceed-case (find-restart 'use-food) (use-food)))
    (do-test "proceed-case signal positive "
        (proceed-case
            (condition-case (signal (make-condition 'our-little-condition))
                        (our-little-condition nil (find-restart 'use-food)))
             (use-food)))
    (do-test "old style find-restart nil :condition"
      (not (proceed-case (find-restart 'use-food)
                     (use-food nil :condition our-little-condition nil))))
    (do-test "old style find-restart nested inner"
        (define-proceed-function use-food :report "Select this food.")
        (proceed-case
            (proceed-case
                (and (setq our-proceed-case
                           (find-restart 'use-food))
                     (typep our-proceed-case 'restart)
                     (string-equal "The inner case."
                                   (princ-to-string our-proceed-case)))
                (use-food nil :report "The inner case." t)) (use-food)))
    (do-test "old style find-restart nested outer"
        (proceed-case
            (progn (and (test-setq our-proceed-case
                                   (find-restart 'use-food))
                        (typep our-proceed-case 'restart)
                        (string-equal "Select this food."
                                      (princ-to-string our-proceed-case)))
                   (proceed-case (find-restart 'use-food)
                                 (use-food nil :report "The inner case." t))
```

```
                (and (test-setq our-proceed-case
                           (find-restart 'use-food))
                     (typep our-proceed-case 'restart)
                     (string-equal "Select this food."
                                   (princ-to-string our-proceed-case))))
            (use-food)))) )
;; Definition To Be Tested: proceed-case
;;
;; Source:        Xerox LIsp Manual
;; Chapter 24: ERROR SYSTEM    Proceeding from Conditions
;;
;; Created By:    Kirk Kelley
;;
;; Creation Date:  21 November 86
;;
;; Last Update:   >> day month << 86
;;
;; Filed As:      {eris}<lispcore>cml>test>24-proceed-case.ux
;;
;;
;; Syntax: proceed-case form &rest clauses    [Macro]
;;
;; Function Description: The form is evaluated in a dynamic context where the clauses have
special meanings as points to which control may be transferred in the event that a condition is
is signalled.  If form runs to completion and eturns any values, all values returned by theform
are simply returned by the proceed-case from.  If a condition is signalled while form is running,
a handler may transfer control to one of the clauses. If a transfer to a clause occurs, the forms
in the body of that clause will be evaluated and any values returned by the last such form will
be returned by the proceed-case form.  See the documentation for further information.
;;
;; Argument(s):  form
;;                    clauses -- (proceed-function-name arglist [keyword value]* [body-form]*)
;;                            valid keyword/value pairs are:
;;                                  :filter-function expression
;;                                  :filter form
;;                                  :condition type
;;                                  :report-function exp
;;                                  :report form
;;
;; Returns: value of last form or handled form
;;
(do-test-group "restart-case :filter-function"
    (do-test "restart-case :filter-function simple positive"
          (restart-case (find-restart 'use-food)
                    (use-food nil :filter-function
                            (lambda ()
                                t))))
    (do-test "restart-case :filter-function simple negative"
          (restart-case (not (find-restart 'use-food))
                    (use-food nil :filter-function
                            (lambda ()
                                nil))))
    (do-test "restart-case :filter-function simple negative 2"
          (restart-case (not (find-restart 'use-food))
                    (use-food nil :filter-function
                            (lambda ()
                                (typep *current-condition*
                                     'our-little-condition)))))
    (do-test "restart-case *cur-cond* :filter-function positive"
        (restart-case
            (let ((*current-condition* (make-condition 'our-little-condition)))
                (find-restart 'use-food))
            (use-food nil :filter-function
                    (lambda ()
                        (typep *current-condition* 'our-little-condition)))))
    (do-test "restart-case :filter simple positive"
          (restart-case (find-restart 'use-food)
                    (use-food nil :filter t)))
    (do-test "restart-case :filter simple negative"
          (restart-case (not (find-restart 'use-food))
                    (use-food nil :filter nil)))
    (do-test "restart-case :condition negative"
        (restart-case
            (not (let ((*current-condition* (make-condition 'our-little-condition)))
                   (find-restart 'use-food)))
            (use-food nil :condition error)))
    (do-test "restart-case :condition positive"
        (restart-case
            (let ((*current-condition* (make-condition 'our-little-condition)))
                (find-restart 'use-food))
```

```
            (use-food nil :condition our-little-condition)))
    (do-test "restart-case :filter and :condition error"
          (expect-errors (simple-error)
                    (restart-case (find-restart 'use-food)
                              (use-food nil :condition our-little-condition
                                        :filter t))))
    (do-test "restart-case :filter and :filter-function error"
        (expect-errors (simple-error)
            (restart-case (find-restart 'use-food)
                      (use-food nil :filter t :filter-function
                                (lambda ()
                                    (typep *current-condition*
                                           'our-little-condition)))))))
(do-test-group "restart-case :report-function"
    (do-test "restart-case :report-function"
        (restart-case
            (string-equal "Select this."
                      (princ-to-string (find-restart 'use-food)))
            (use-food nil :report-function
                    (lambda (restart-case *standard-output*)
                        (write-string "Select this." *standard-output*)))))
    (do-test "restart-case :report-function 2"
        (restart-case
            (string-equal "Select this."
                      (princ-to-string (find-restart 'use-food)))
            (use-food nil :report-function
                    (lambda (ignore stream)
                        (write-string "Select this." stream)))))
    (do-test "restart-case :report"
        (restart-case
            (string-equal "Select this."
                      (princ-to-string (find-restart 'use-food)))
            (use-food nil :report "Select this.")))
    (do-test "restart-case :report"
        (restart-case
            (string-equal "Select this."
                      (princ-to-string (find-restart 'use-food)))
            (use-food nil :report (write-string "Select this." *standard-output*))))
    (do-test "restart-case :report and :report-function error"
        (expect-errors (simple-error)
            (restart-case
                (string-equal "Select this."
                          (princ-to-string (find-restart 'use-food)))
                (use-food nil :report
                        (write-string "Select this." *standard-output*)
                        :report-function
                        (lambda (ignore stream)
                            (write-string "Select this." stream)))))))
(do-test-group "nested restart-case inner catch and throw"
    (do-test "nested restart-case catch and throw"
        (restart-case
            (catch 'test-throw
                (restart-case
                    (block test-throw
                        (throw 'test-throw
                            (string-equal "Select this."
                                (princ-to-string
                                    (find-restart 'proceed)))) nil)
                    (proceed nil :report "Select this." nil)))
            (proceed nil :report "Don't Select this." nil)))
    (do-test "throw restart-case"
        (catch 'test-throw
            (throw 'test-throw
                (restart-case
                    (string-equal "Select this."
                        (princ-to-string (find-restart 'proceed)))
                    (proceed nil :report "Select this.")))))
    (do-test "throw nested restart-case"
        (catch 'test-throw
            (throw 'test-throw
                (restart-case
                    (restart-case
                        (string-equal "Select this."
                            (princ-to-string (find-restart 'proceed)))
                        (proceed nil :report "Select this." nil))
                    (proceed nil :report "Don't Select this." nil)))))
    (do-test "nested restart-case outer catch and inner throw"
        (catch 'test-throw
            (restart-case
                (restart-case
```

```
                    (progn
                        (throw 'test-throw
                            (string-equal "Select this."
                                (princ-to-string
                                    (find-restart 'proceed
                                        (make-condition
                                            'simple-condition)))))
                            nil)
                        (proceed nil :report "Select this." nil))
                (proceed nil :report "Don't Select this." nil)) nil))
    (do-test "internested restart-case nested catch and throw"
        (not  (catch 'test-throw
                (restart-case
                    (catch 'test-throw
                        (restart-case
                            (progn
                                (throw 'test-throw
                                    (string-equal "Select this."
                                        (princ-to-string
                                            (find-restart
                                                'proceed
                                                (make-condition
                                                    'simple-condition)))))
                                nil)
                            (proceed nil :report "Select this." nil)))
                    (proceed nil :report "Don't Select this." nil)) nil))))
(do-test "restart-case: dynamic environment"
    (let ((x t))
        (declare (special x))
        (restart-case
            (let ((x nil))
                (declare (special x))
                (invoke-restart (find-restart 'use-food)))
            (use-food nil :report "Select this." x))))
;; Definition To Be Tested: define-proceed-function
;;
;; Source:        Xerox LIsp Manual
;; Chapter 24: ERROR SYSTEM    Proceeding from conditions
;;
;; Created By:    Kirk Kelley
;;
;; Creation Date:  21 November 86
;;
;; Last Update:   >> day month << 86
;;
;; Filed As:      {eris}<lispcore>cml>test>24-define-proceed-function.test
;;
;;
;; Syntax: define-proceed-function name [keyword value]* &rest variables   [Macro]
;;
;; Function Description: Defines a function called name which will proceed an error in a typed
way.  The only thing that a proceed function really does is collect values to be passed on to a
proceed-case clause.  Valid keyword/value pairs are the same as those which are defined for the
proceed-case special form.  That is, :test, :condition, :report-funciton, and :report.  The test
and report functions specified in a define-proceed-function form will be used for proceed-case
clauses with the same name that do not specify their own test or report functions, respectively.
See the documentation for further information.
;;
;; Argument(s):  name (of function to be defined)
;;                   keyword/value pairs:
;;                                :test function
;;                                :condition type
;;                                :report-function exp
;;                                :report form
;;                &optional variables
;;                     each variable has the form
;;                         variable-name or
;;                        (variable-name initial-value)
;;
;; Returns: value of function or handled proceed clause
;;
(do-test "define-proceed-function" (fmakunbound 'test-fn)
      (and (define-proceed-function test-fn :report "our little report")
          (fboundp 'test-fn)
          (proceed-case (string-equal "our little report"
                                (default-proceed-report 'test-fn))
                    (test-fn nil t))))
(do-test-group "define-proceed-function default parameter collection"
    (do-test "define-proceed-function test-fn" (fmakunbound 'test-fn)
          (define-proceed-function test-fn :report "Select this food." (y t)))
```

```
    (do-test "define-proceed-function find test"
          (proceed-case (find-restart 'test-fn) (test-fn)))
    (do-test "define-proceed-function default parameter collection"
          (proceed-case (invoke-proceed-case (find-restart 'test-fn))
                        (test-fn (y) y))))
;; Definition To Be Tested: compute-proceed-cases
;;
;; Source:        Xerox LIsp Manual
;; Chapter 24: ERROR SYSTEM    Proceeding from Conditions
;;
;; Created By:    Kirk Kelley
;;
;; Creation Date:  21 November 86
;;
;; Last Update:   >> day month << 86
;;
;; Filed As:      {eris}<lispcore>cml>test>24-compute-proceed-cases.x
;;
;;
;; Syntax: compute-proceed-cases condition        [Function]
;;
;; Function Description: Uses the dynamic state of the program to compute a list of proceed cases
which may be used with the given condition.  See the documentation for more information.
;;
;; Argument(s):  condition
;;
;; Returns: list of proceed cases
;;
(do-test-group "compute-restart-cases"
    (do-test "compute-restart-cases single"
          (restart-case (member-if #'(lambda (case)
                                       (eq (restart-case-name case)
                                           'proctest))
                                   (compute-restart-cases)) (proctest))))
(do-test "compute-restart-cases multiple" (fmakunbound 'test-fn)
    (define-proceed-function test-fn :report "Select this food." (y t))
    (restart-case
        (restart-case (and (member-if #'(lambda (case)
                                          (equal (restart-case-name case)
                                                 'test-fn))
                                      (compute-restart-cases))
                           (member-if #'(lambda (case)
                                          (equal (restart-case-name case)
                                                 'proceed))
                                      (compute-restart-cases)))
                      (test-fn nil t)) (proceed)))
;; Definition To Be Tested: restart-case-name
;;
;; Source:        Xerox LIsp Manual
;; Chapter 24: ERROR SYSTEM    Proceeding from Conditions
;;
;; Created By:    Kirk Kelley
;;
;; Creation Date:  21 November 86
;;
;; Last Update:   >> day month << 86
;;
;; Filed As:      {eris}<lispcore>cml>test>24-restart-case-name.test
;;
;;
;; Syntax: restart-case-name restart-case
;;
;; Function Description: Returns the name of the given restart-case, or nil if it is not named.
;;
;; Argument(s):  restart-case
;;
;; Returns: name or nil
;;
(do-test "restart-case-name named"
      (restart-case (equalp (restart-case-name (find-restart 'proceed))
                            'proceed) (proceed)))
;; Definition To Be Tested: default-proceed-test
;;
;; Source:        Xerox LIsp Manual
;; Chapter 24: ERROR SYSTEM    Proceeding from Conditions
;;
;; Created By:    Kirk Kelley
;;
;; Creation Date:  21 November 86
;;
```

```
;; Last Update:   >> day month << 86
;;
;; Filed As:      {eris}<lispcore>cml>test>24-default-proceed-test.x
;;
;;
;; Syntax: default-proceed-test restart-case-name
;;
;; Function Description: Returns the default test function for proceed cases with the given name.
May be used with setf to change it. [This is a Xerox Lisp extension.]
;;
;; Argument(s):  restart-case-name
;;
;; Returns: function
;;
(do-test-group "default-proceed-test"
    (do-test "default-proceed-test simple"
          (functionp (default-proceed-test 'proceed)))
    (do-test "default-proceed-test override" (fmakunbound 'test-fn)
        (define-proceed-function test-fn :report "our little report" :filter t)
        (setq testfn (default-proceed-test 'test-fn))
        ;;begin test
        (restart-case (equalp (default-proceed-test 'test-fn) testfn)
                  (test-fn nil :filter-function #'nil))))
;; Definition To Be Tested: default-proceed-test
;;
;; Source:        Xerox LIsp Manual
;; Chapter 24: ERROR SYSTEM    Proceeding from Conditions
;;
;; Created By:    Kirk Kelley
;;
;; Creation Date:  21 November 86
;;
;; Last Update:   >> day month << 86
;;
;; Filed As:      {eris}<lispcore>cml>test>24-default-proceed-test.x
;;
;;
;; Syntax: default-proceed-test restart-case-name
;;
;; Function Description: Returns the default report function for proceed cases with the given
name.  May be used with setf to change it. [A Xerox Lisp extension.]
;;
;; Argument(s):  restart-case-name
;;
;; Returns: function
;;
(do-test-group
    ("default-proceed-report" :before
     (fmakunbound 'test-fn
                (define-proceed-function test-fn :condition simple-condition
                                         :report "Select this food.")))
    (do-test "default-proceed-report simple"
          (string-equal (default-proceed-report 'test-fn) "Select this food."))
    (do-test "default-proceed-report override"
          (restart-case (string-equal (default-proceed-report 'test-fn)
                                      "Select this food.")
                    (test-fn nil :report "A different report."))))
;; Definition To Be Tested: invoke-restart
;;
;; Source:        Xerox LIsp Manual
;; Chapter 24: ERROR SYSTEM    Handling Conditions
;;
;; Created By:    Kirk Kelley
;;
;; Creation Date:  21 November 86
;;
;; Last Update:   >> day month << 86
;;
;; Filed As:      {eris}<lispcore>cml>test>24-invoke-restart.x
;;
;;
;; Syntax: invoke-restart restart-case &rest values   [Function]
;;
;; Function Description: Transfers control to the given restart-case, passing it the given
values.  The restart-case must be a proceed case object or the name of a proceed function which
is valid in the current dynamic context.  If the argument is not valid, the error bad-restart-
case will be signalled.  If the argument is a named proceed case that has a corresponding proceed
function, invoke-restart will do the optional argument resolution specified by that function
before transferring control to the proceed case. [The CL error proposal does not specify a
required second argument.]
```

```
;;
;; Argument(s):  restart-case -- object or name
;;                   condition
;;                   optional values -- for the restart-case
;;
;; Returns: can abort, does not return
;;
(do-test "invoke-restart single"
      (restart-case (invoke-restart 'test-proccase)
                    (test-proccase nil t)))
(do-test "invoke-restart multiple" (fmakunbound 'test-fn)
      (define-proceed-function test-fn :report "Select this food.")
      (and (restart-case (invoke-restart 'test-fn)
                   (proceed nil nil)
                   (test-fn nil t))
           (restart-case (invoke-restart 'proceed)
                   (proceed nil t)
                   (test-fn nil nil))))
;; Definition To Be Tested: catch-abort
;;
;; Source:        Xerox LIsp Manual
;; Chapter 24: ERROR SYSTEM    Proceeding from Conditions
;;
;; Created By:    Kirk Kelley
;;
;; Creation Date:  21 November 86
;;
;; Last Update:   >> day month << 86
;;
;; Filed As:      {eris}<lispcore>cml>test>24-catch-abort.x
;;
;;
;; Syntax: catch-abort print-form &body forms
;;
;; Function Description: Sets up a restart-case context for the proceed function abort.  If no
abort is done while execinting forms and they return normally all values returned by the last
form in forms are returned.  If an abort transfers control to this catch-abort, two values are
returned: nil and the condition which was given to abort (or nil if none was given).
;;
;; Argument(s):  print-form -- e.g. string / format
;;                   forms
;;
;; Returns: values of last form or nil and a condition.
;;
(do-test "simple catch-abort" (not (catch-abort "it worked" (abort))))
(do-test "catch-abort nested"
      (catch-abort "level 1" (not (catch-abort "level 2" (abort)))))
;; Definition To Be Tested: abort
;;
;; Source:        Xerox LIsp Manual
;; Chapter 24: ERROR SYSTEM    Proceeding from Conditions
;;
;; Created By:    Kirk Kelley
;;
;; Creation Date:  21 November 86
;;
;; Last Update:   >> day month << 86
;;
;; Filed As:      {eris}<lispcore>cml>test>24-abort.x
;;
;;
;; Syntax: abort &optional condition
;;
;; Function Description: Transfers control to the innermost (dynamic) catch-abort form, causing
it to return nil immediately.
;;
;; Argument(s):  optional condition
;;
;; Returns: never
;;simple abort is tested in catch-abort
(do-test-group "abort with condition"
    (do-test "abort with condition"
        (multiple-value-bind (result acondition)
            (catch-abort "test" (abort (make-condition 'simple-condition)))
            (and (not result)
                (typep acondition 'simple-condition))))
    (do-test "abort with condition 2"
        ;;the proceed case below should be ignored, so we return
        ;;t if this proceed case is seen.  Normal return from
        ;;catch-abort is nil.
```

```
        (multiple-value-bind (result acondition)
            (catch-abort "test"
                (restart-case
                    (progn (abort (make-condition 'simple-condition)) t)
                    (abort (condition) :filter-function
                         (lambda ()
                             nil) t)))
            (and (not result)
                (typep acondition 'simple-condition)))))
;; Definition To Be Tested: proceed
;;
;; Source:        Xerox LIsp Manual
;; Chapter 24: ERROR SYSTEM    Proceeding from Conditions
;;
;; Created By:    Kirk Kelley
;;
;; Creation Date:  21 November 86
;;
;; Last Update:   >> day month << 86
;;
;; Filed As:      {eris}<lispcore>cml>test>24-proceed.x
;;
;;
;; Syntax: proceed &optional condition
;;
;; Function Description: This is a predefined proceed function.  It is used by such functions as
break, cerror, etc.
;;
;; Argument(s):  optional condition
;;
;; Returns: nil
;;
(do-test-group "proceed"
    (do-test "proceed simple"
          (restart-case (find-restart 'proceed) (proceed)))
    (do-test "proceed body"
        (and (not  (restart-case
                     (invoke-restart (find-restart 'proceed))
                     (proceed nil nil)))
            (restart-case (invoke-restart (find-restart 'proceed))
                       (proceed nil t))))
    (do-test "proceed filter"
          (restart-case (not (find-restart 'proceed))
                       (proceed nil :filter nil)))
    (do-test "proceed report"
        (restart-case (string-equal "Select this."
                               (princ-to-string (find-restart 'proceed)))
                     (proceed nil :report "Select this."))))
(do-test-group "proceed nested"
    (do-test "proceed nested inner"
        (restart-case
            (restart-case (invoke-restart (find-restart 'proceed))
                       (proceed nil t))
            (proceed nil nil)))
    (do-test "proceed nested outer"
        (restart-case
            (progn (restart-case (restart-case nil (proceed nil nil)))
                 (invoke-restart (find-restart 'proceed))
                 (restart-case (restart-case nil (proceed nil nil))))
            (proceed nil t)))
    (do-test "proceed nested both"
        (restart-case
            (progn
                (restart-case
                    (invoke-restart (find-restart 'proceed))
                    (proceed nil nil))
                (invoke-restart (find-restart 'proceed)))
            (proceed nil t))))
(do-test-group "proceed bindings"
    (do-test "proceed closure"
        (eq   'x
            (let ((val 'x))
                (restart-case
                    (invoke-restart (find-restart 'proceed))
                    (proceed nil val))))))
;; Definition To Be Tested: use-value
;;
;; Source:        Xerox LIsp Manual
;; Chapter 24: ERROR SYSTEM    Proceeding from Conditions
;;
```

```
;; Created By:    Kirk Kelley
;;
;; Creation Date:  21 November 86
;;
;; Last Update:   >> day month << 86
;;
;; Filed As:      {eris}<lispcore>cml>test>24-use-value.ux
;;
;;
;; Syntax: use-value &optional new-value
;;
;; Function Description: This is a predefined proceed function.  It is intended to be used for
supplying an alternate value to be used in a compuatation.  If new-value is not provided, use-
value will prompt the user for one.
;;
;; Argument(s):  optional value
;;
;; Returns: n/a
;;
(do-test "use-value"
      (and (not (restart-case (invoke-restart 'use-value)
                        (use-value 'simple-condition nil)))
           (restart-case (invoke-restart 'use-value)
                      (use-value 'simple-condition t))))
;; Definition To Be Tested: store-value
;;
;; Source:        Xerox LIsp Manual
;; Chapter 24: ERROR SYSTEM    Proceeding from Conditions
;;
;; Created By:    Kirk Kelley
;;
;; Creation Date:  21 November 86
;;
;; Last Update:   >> day month << 86
;;
;; Filed As:      {eris}<lispcore>cml>test>24-use-value.x
;;
;;
;; Syntax: store-value &optional new-value
;;
;; Function Description: This is a predefined proceed function.  It is intended to be used for
supplying an alternate value to be stored in some location as a way of proceeding from an error.
store-value does not actually store the new vlaue anywhere: it is up to proceed case to take care
of that.  If new-value is not provided, store-value will prompt the user for one.  store-value is
used by such forms as check-type and cerror.
;;
;; Argument(s):  optional value
;;
;; Returns: n/a
;;
(do-test-group "store-value"
           (do-test "store-value"
                 (and (not (restart-case (invoke-restart 'store-value)
                                    (store-value 'simple-condition nil)))
                      (restart-case (invoke-restart 'store-value)
                                 (store-value 'simple-condition t)))))
                                 STOP
```