APPENDIX E. ERROR SYSTEM

This appendix replaces Chapter 24, Error System, of *Common Lisp Implementation Notes,* Lyric Release*,* which replaced most of Chapter 24, Errors, of *Common Lisp, the Language*. Text shown with ~~StrikeThru~~ is that text from the Lyric release that no longer applies in Medley. Enhancements added in Medley are indicated with revision bars in the right margin.

The XCL error system has been updated to reflect the current ANSI Common Lisp error system proposal. This version seems to be gaining wide use in other Common Lisp implementations, so no further major changes are anticipated.

The Common Lisp error system is based on proposal number 18 for the Common Lisp error system. Deviations from this proposal are noted. Since the Common Lisp error system has not yet been standardized, this system may change in future releases to accommodate the final version of the Common Lisp error system.

If you have access to the ARPANet, a copy of this proposal may be retrieved from MIT-AI.ARPA as the file "COMMON;COND18 TXT".

All symbols described in the error system proposal that are not already in the "LISP" package are exported from the "CONDITIONS" package. In addition, the "XEROX-COMMON-LISP" package exports these symbols, so you can make them available either by using "XCL" or using "CONDITIONS", whichever is appropriate to your application. The distinction is made so that XCL extensions of the Common Lisp error system will be clear. All unqualified symbols are assumed to be in the "LISP" package.

Summary of Error System Changes

The semantics of HANDLER-BIND where multiple bindings are set up or mutiple condition types are being handled are slightly different. Old code that used this will probably not behave as expected.

HANDLER-BIND and HANDLER-CASE (a.k.a. CONDITION-CASE) now always take a typespec instead of a list of condition types to indicate the conditions to be handled. Old code that uses this will only handle the first condition type in the list. The function, CONDITIONS::CONVERT-HANDLER-CASE is provided to aid in converting old code. It may be used as a mutation function in SEdit.

HANDLER-CASE now supports a :NO-ERROR option that is executed if none of the other clauses are taken. This is handy for writing code that depends on the normal completion of some operation, for example, creating auxilliary files if a particular stream is successfully opened.

SERIOUS-CONDITION no longer forces entry to the debugger. The function used to signal the condition now determines what happens if the condition is not handled. This means that SERIOUS-CONDITION has no more interesting properties and is likely to be removed in the final version of the error standard.

Several new condition types have been defined. Others have moved in the hierarchy. For example, ILLEGAL-GO is now a subtype of PROGRAM-ERROR.

No standard condition type has a default handler.

The standard debugger entry point is now called INVOKE-DEBUGGER instead of DEBUG.

The syntax of DEFINE-CONDITION has been changed to make it more like CLOS' DEFCLASS. The function CONDITIONS::CONVERT-OLD-DEFINE-CONDITION is provided to aid in converting old code. It may be used as a mutation function in SEdit.

Several DEFINE-CONDITION options have been merged, while others have been removed. In particular, there are no more "instant variables."

PROCEED-CASE has been replaced by RESTART-CASE. The semantics of restarts have been cleaned up and several new features added. Related functions, such as COMPUTE-PROCEED-CASES, have been renamed appropriately.

INVOKE-PROCEED-CASE has been renamed to INVOKE-RESTART.

DEFINE-PROCEED-FUNCTION has been removed, although XCL will continue to support it for compatibility.

The arguments to a restart's report function are different. Old code that used something other than a string for the report method will not work correctly.

A distinction is now made between invoking a restart interactively and simply invoking one. To this end, there is the function INVOKE-RESTART-INTERACTIVELY and the :INTERACTIVE option to RESTART-CASE.

RESTART-BIND, in analogy to HANDLER-BIND, has been added.

A new variable, *BREAK-ON-SIGNALS* exists to aid in debugging. It is a generalization of *BREAK-ON-WARNINGS*. The latter has been retained for compatibility.

The proceed function PROCEED has been changed to CONTINUE.

Old compiled code will continue to work except in the following cases, some of which have been mentioned above:

A proceed case's report function was not a simple string. Such code can cause stack overflow trying to report the condition (*STANDARD-OUTPUT* ends up being bound to NIL). Such code should be rewritten.

A handler binding is made to a list of condition types. Only the first type in the list will be handled.

Multiple handler bindings were created by the same HANDLER-BIND or HANDLER-CASE. Such code will work as expected, but if recompiled in Medley, will not. To get the effect of the current semantics, you must use nested HANDLER-BINDs.

Under the new error system, `use-value` and `store-value` no longer prompt for a value.

Introduction to Error System Terminology

condition A *condition* is a kind of object which is created when an exceptional situation arises in order to represent the relevant features of that situation.

signal, handlers Once a condition is created, it is common to *signal* it. When a condition is signaled, a set of *handlers* are tried in some pre-defined order until one decides to *handle* the condition or until no more handlers are found. A condition is said to have been handled if a handler performs a non-local transfer of control to exit the signalling process.

restart Although such transfers of control may be done directly using traditional Lisp mechanisms such as `catch` and `throw`, `block` and `return`, or `tagbody` and `go`, the condition system also provides a more structured way to *restart* a computation. Among other things, the use of these structured primitives for restarting allows a better and more integrated relationship between the user program and the interactive debugger.

~~*serious conditions*~~ ~~It is not necessary that all conditions be handled. Some conditions are trivial enough that a failure to handle them may be disregarded. Others, which we will call *serious conditions* must be handled in order to assure correct program behavior. If a serious condition is signalled but no handler is found, the debugger will be entered so that the user may interactively specify how to proceed.~~

errors conditions which result from incorrect programs or data are called *errors*. Not all conditions are errors, however. Storage conditions are examples of conditions that are not errors. For example, the control stack may legitimately overflow without a program being in error. Even though a stack overflow is not necessarily a program error, it is serious enough to warrant entry to the debugger if the condition goes unhandled.

Some types of conditions are predefined by the system. All types of conditions are subtypes of `conditions:condition`. That is,

```
(typep c 'conditions:condition)
```

is true if `c` is a condition.

creating conditions

The only standard way to define a new condition type is `conditions:define-condition`. The only standard way to instantiate a condition is `conditions:make-condition`.

When a condition object is created, the most common operation to be performed upon it is to *signal* it (although there may be applications in which this does not happen, or does not happen immediately).

When a condition is signaled, the system tries to locate the most appropriate handler for the condition and invoke that handler. Handlers are located according to the following rules:

bound

- Check for locally defined (ie, *bound*) handlers.
- If no appropriate bound handler is found, check first for the default handler of the signaled type and then of each of its superiors.

decline

If an appropriate handler is found, the handler may *decline* by simply returning without performing a non-local transfer of control. In such cases, the search for an appropriate handler is picked up where it left off, as if the called handler had never been present. When a handler is running, the "handler binding stack" is popped back to just below the binding that caused that handler to be invoked. This is done to avoid infinite recursion in the case that a handler also signals a condition.

`conditions:handler-bind`

When a condition is signaled, handlers are searched for in the dynamic environment of the signaller. Handlers can be established within a dynamic context by use of `conditions:handler-bind` and other forms based on it.

handler

A *handler* is a function of one argument, the condition to be handled. The handler may inspect the object (using primitives described in another section) to be sure it is interested in handling the condition. After inspecting the condition, the handler must take one of the following actions:

- It may decline to handle the condition by simply returning. When this happens, any returned values are ignored and the effect on the signaling process is the same as if the handler had not run. The next handler in line will be tried, or if no such handler exists, the default action for the given condition will be taken. A default handler may also decline, in which case the condition will go unhandled. What happens then depends on which function was used to signal the condition (`xcl:signal`, `error`, `cerror`, `warn`).
- It may perform some non-local transfer of control using `go`, `return`, `throw`, `abort`, or `conditions:invoke-restart`.
- It may signal another condition.
- It may invoke the debugger.

`conditions:restart-case` When a condition is signalled, a facility is available for use by handlers to transfer control to an outer dynamic contour of the program. The form which creates contours that may be returned to is `conditions:restart-case`. Each contour is set up by a `conditions:restart-case` clause, and is called a *restart*. The function that transfers control to a restart is `conditions:invoke-restart`.

~~*proceed function* Also, control may be transferred along with parameters to a named `xcl:proceed-case` clause by invoking a *proceed function* of that name.~~

~~*Proceed functions* are created with the macro `xcl:define-proceed-function`.~~

restart type A restart with a particular name is sometimes called a *restart type*.

report In some cases, it may be useful to *report* a condition or a restart to a user or a log file of some sort. When the printer is invoked on a condition or proceed case and `*print-escape*` is `nil`, the report function for that object is invoked. In particular, this means that an expression like

```
(princ condition)
```

will invoke `condition`'s report function. Because of this, no special function is provided for invoking the report function of a condition or a restart.

Program Interface to the Condition System

Defining and Creating Conditions

conditions:define-condition *name* (*parent-type*) [({*slot*}*) {*option*}*]

[Macro]

Defines a new condition type with the given *name*, making it a subtype of the given *parent-type*.

Except as otherwise noted, the arguments are not evaluated.

The valid *option*s are:

`(:documentation` *doc-string*`)`

doc-string should be a string which describes the purpose of the condition type or `NIL`. If this option is omitted, `NIL` is assumed. `(documentation` *name* `'type)` will retrieve this information.

`(:conc-name` *symbol-or-string*`)`

As in `defstruct`, this sets up automatic prefixing of the names of slot accessors. Also as in `defstruct` if no prefix is specified the default behavior for automatic prefixing is to use the name of the new type followed by a hyphen interned in the

package which is current at the time that the `conditions:define-condition` is processed.

~~`:report-function` *expression*~~

~~*expression* should be a suitable argument to the `function` special form, e.g., a symbol or a lambda expression. It designates a function of two arguments, a condition and a stream, which prints the condition to the stream when `*print-escape*` is `nil`.~~

~~The `:report-function` describes the condition in a human-sensible form. This item is somewhat different than a structure's `:print-function` in that it is only used if `*print-escape*` is `nil`.~~

`(:report` *exp*`)`

This option specifies the report function for this condition type. Report function are inherited, so if a particular condition type does not have one, the report function of its parent will be used.

If *exp* is a string, it is a shorthand for

```
(:report (lambda (condition stream)
        (declare (ignore conditions))
         (princ exp stream)))
```

If *exp* is not a string, `(function` *exp*`)` will be evaluated in the current lexical environment. This should return a function of two arguments, a condition and a stream. It will be called when a condition of this type is to be printed and `*print-escape*` is `nil`. The report function will be called with the condition to be reported and the stream to which the report is to be made.

~~`:handler-function` *expression*~~

~~*expression* should be a suitable argument to the `function` special form. It designates a function of one argument, a condition, which may handle that condition if no dynamically-bound handler did.~~

`(:handle` *exp*`)`

This option specifies a default handler for conditions of this type. `(function` *exp*`)` will be evaluated in the current lexical context. This should result in a function of one argument, a condition, to be used as the default handler for this condition type.

Each *slot* is a `defstruct` *slot-description*. In addition to those specified, the slots of the *parent-type* are also available. No slot-options are allowed, only an optional default-value expression. Condition objects are immutable, i.e., all of their slots are automatically declared to be `:read-only`.

`conditions:make-condition` will accept keywords with the same name as any of the slots, and will initialize the corresponding slots in conditions it creates.

Accessors are created according to the same rules as used by `defstruct`. For example:

```
(conditions:define-condition bad-food-color (food-lossage)
(food color)
      (:report (lambda (c s) (format s "The food ~A was ~A"
                        (bad-food-color-food c) (bad-food-
color-color c)))))
```

defines a condition of type `bad-food-color` which inherits from the `food-lossage` condition type. The new type has slots `food` and `color` so that `conditions:make-condition` will accept `:food` and `:color` keywords and accessors `bad-food-color-food` and `bad-food-color-color` will apply to objects of this type.

The report function for a condition will be implicitly called any time a condition is printed with `*print-escape*` being `nil`. Hence,

```
(princ condition)
```

is a way to invoke the condition's report function.

Here are some examples of defining condition types. This form defines a condition called `machine-error` which inherits from `error`:

```
(conditions:define-condition machine-error (error) (machine-
name)
      (:report (lambda (c s) (format s
                  "There is a problem with ~A."
                  (machine-error-machine-name c))))
   )
```

The following defines a new error condition (a subtype of `machine-error`) for use when machines are not available:

```
(conditions:define-condition machine-not-available-error
                     (machine-error) (machine-name)
    (:report (lambda (c s) (format s
               "The machine ~A is not available."
               (machine-error-machine-name c))))
   )
```

The following defines a still more specific condition, built upon `machine-not-available-error`, which provides a default for `machine-name` but which does not provide any new slots:

```
(conditions:define-condition
          my-favorite-machine-not-available-error
          (machine-not-available-error)
   ((machine-name "Tesuji:AISDev")))
```

This gives the `machine-name` slot a default initialization. Since no `:report` clause was given, the information supplied in the definition of `machine-not-available-error` will be used if a condition of this type is printed while `*print-escape*` is `nil`.

xcl:condition-reporter *type* [Macro]

Returns the object used to report conditions of the given *type*. This will be either a string, a function of two arguments (condition and stream) or `nil` if there is no report function. `setf` may be used with this form to change the report function for a condition type.

xcl:condition-handler *type* [Macro]

Returns the default handler for conditions of the given *type*. This will be a function of one argument or `nil` if there is no default handler. `setf` may be used with this form to change the default handler for a condition type.

conditions:make-condition *type* &**rest** *slot-initializations* [Function]

Calls the appropriate constructor function for the given type, passing along the given slot initializations to the constructor, and returning an instantiated condition.

The *slot-initializations* are given in alternating keyword/value pairs. eg,

```
(conditions:make-condition 'bad-food-color
   :food my-food
   :color my-color)
```

This function is provided mainly for writing subroutines that manufacture a condition to be signaled. Since all of the condition signalling functions can take a *type* and *slot-initializations*, it is usually easier to call them directly.

Signalling Conditions

xcl:*current-condition* [Variable]

This variable is bound by condition-signalling forms (`conditions:signal`, `error`, `cerror`, and `warn`) to the condition being signaled. This is especially useful in restart filters. The top-level value of `xcl:*current-condition*` is `nil`.

conditions:signal *datum* &**rest** *arguments* [Function]

Invokes the signal facility on a condition. If the condition is not handled, `conditions:signal` returns the condition object that was signaled.

If *datum* is a condition then that condition is used directly. In this case, it is an error for *arguments* to be non-`nil`.

If *datum* is a condition type, then the condition used is the result of doing

```
(apply #'conditions:make-condition
       datum arguments)
```

If *datum* is a string, then the condition used is the result of doing

```
(conditions:make-condition
   'conditions:simple-condition
   :format-string datum
   :format-arguments arguments).
```

~~If the condition is of type `xcl:serious-condition`, then `xcl:signal` will behave exactly like `error`, i.e., it will call~~

~~xcl:debug if the condition isn't handled, and will never return to its caller.~~

If (typep *condition* conditions:*break-on-signals*) is true, then the debugger will be entered prior to the signalling process. This is true for all other functions and macros that signal conditions, such as warn, error, cerror, assert and check-type.

conditions:*break-on-signals* [Variable]

This flag is primarily for use when debugging programs that do signaling. Its value is a type specifier.

When (typep *condition* conditions:*break-on-signals*) is true, then calls to conditions:signal and other functions that implicitly call conditions:signal will enter the debugger prior to signalling the condition. The conditions:continue restart may be used to continue with the normal signalling process.

The default value of this variable is nil.

Note: the variable *break-on-warnings* continues to be supported for compatibility, but conditions:*break-on-signals* offers that power and more. New code should not use *break-on-warnings*.

error *datum* &**rest** *arguments* [Function]

Like conditions:signal except if the condition is not handled, the debugger is called with the given condition, and error never returns.

datum is treated as in conditions:signal. If *datum* is a string, a conditon of type conditions:simple-error is made. This form is compatible with that described in Steele's *Common Lisp, the Language*.

cerror *proceed-format-string datum* &**rest** *arguments* [Function]

Like error, if the condition is not handled the debugger is called with the given condition. However, cerror enables the restart conditions:continue, which will simply return the condition being signalled from cerror.

datum is treated as in error. If *datum* is a condition, then that condition is used directly. In this case, *arguments* will be used only with the *proceed-format-string* and will not be used to initialize *datum*.

The *proceed-format-string* must be a string. Note that if *datum* is not a string, then the format arguments used by the *proceed-format-string* will still be the *arguments* (in the keyword format as specified). In this case, some care may be necessary to set up the *proceed-format-string* correctly. The format directive ~* may be particularly useful in this situation.

The value returned by cerror is the condition which was signaled.

See Steele's *Common Lisp, the Language,* page 430 for examples of the use of `cerror`.

warn *datum* `&`**rest** *arguments* [Function]

Invokes the signal facility on a condition. If the condition is not handled, then the text of the warning is printed on `*error-output*`. If the variable `*break-on-warnings*` is true, then in addition to printing the warning, the debugger is entered using the function `break`. The value returned by `warn` is the condition that was signalled.

If *datum* is a condition, then that condition is used directly. In this case, if the condition is not of type `conditions:warning` or *arguments* is non-null, then an error of type `conditions:type-error` is signalled.

If *datum* is a condition type, then the condition used is the result of doing `(apply #'conditions:make-conditions` *datum arguments*`)`. This result must be of type `conditions:warning` or an error of type `conditions:type-error` is signalled.

If *datum* is a string, then the condition used is the result of `(conditions:make-conditions 'conditions:simple-warning :format-string` *datum* `:format-arguments` *arguments*`)`.

The precise mechanism for warning is as follows:

1) If `*break-on-warnings*` is true, the debugger will be entered. This feature is primarily for compatibility with old code: use of `conditions:*break-on-signals*` is preferred. If the break is continued using the `conditions:continue` restart, `warn` proceeds with step 2.

2) The warning condition is signalled. While it is being signalled, the `conditions:muffle-warning` restart is established for use by a handler to bypass further action by `warn`, i.e., to cause `warn` to immediately return.

3) The warning condition is reported to `*error-output*` by the `warn` function. Note that `warn` will indicate that the condition being signalled is a warning when it reports it, so there is no need for the condition to do so in its report method.

break-on-warnings [Variable]

check-type [Macro]

ecase [Macro]

ccase [Macro]

etypecase [Macro]

ctypecase [Macro]

assert [Macro]

All of the above behave as described in *Common Lisp: the Language*. The default clauses of `ecase` and `ccase` forms signal `conditions:simple-error` conditions. The default clauses of

`etypecase` and `ctypecase` forms signal `conditions:type-error` conditions. `assert` signals the `xcl:assertion-failed` condition. `ccase` and `ctypecase` set up a `conditions:store-value` restart.

Handling Conditions

conditions:handler-bind *bindings* `&`**rest** *forms* [Macro]

Executes the forms in a dynamic context where the given local handler *bindings* are in effect. The elements of *bindings* must take the form `(`*type-spec handler*`)`. The handlers are bound in the order they are given, i.e., when searching for a handler, the error system will consider the leftmost binding in a particular `conditions:handler-bind` form first. However, while one of these handlers is running, none of the bindings established by the `conditions:handler-bind` will be in effect.

type must be a type specifier. To make a binding for several condition types, use `(or` *type1 type2* ...`)`.

handler should evaluate to a function of one argument, a condition, to be used to handle a signalled condition during execution of the *forms*.

An example of the use of `conditions:handler-bind` appears at the end of the `conditions:restart-case` macro description.

conditions:handler-case *form* `&`**rest** *cases* [Macro]

xcl:condition-case *form* `&`**rest** *cases* [Macro]

Executes the given *form*. Each *case* has the form

`(`*type* `(`[*var*]`)` **.** *body*`)`

If a condition is signalled (and not handled by an intervening handler) during the execution of the form, and there is an appropriate clause—i.e., one for which

`(typep` *condition* **'** *type*`)`

is true—then control is transferred to the body of the relevant clause, binding *var*, if present, to the condition that was signaled. If no condition is signalled, then the values resulting from the *form* are returned by the `xcl:condition-case`. If the condition is not needed, `var` may be omitted.

Earlier clauses will be considered first by the error system. I.e.,

```
(xcl:condition-case form
   (cond1 ...)
   (cond2 ...))
```

is equivalent to

```
(xcl:condition-case
        (xcl:condition-case form
           (cond1 ...))
   (cond2 ...))
```

~~*type* may also be a list of types, in which case it will catch conditions of any of the specified types.~~

One may also specify an action to be taken if execution of *form* completes normally. This may be done by specifying a clause that has `:no-error` as its type. Such a clause, if provided, must be last. A `:no-error` clause looks like:

`(:no-error` *lambda-list* `.` *body*`)`

If execution of the form completes normally and there is a `:no-error` clause, the values produced by the form will be bound to variables in the clause's *lambda-list* and the *body* will be executed with none of the handler bindings in effect. In this case the value of the `xcl:condition-case` form is the value returned by the last form of the *body* of its `:no-error` clause. Having a :no-error clause is equivalent to wrapping `(mutiple-value-call #'(lambda` *lambda-list* `.` *body*`) ...)` around the `xcl:condition-case` form.

`conditions:handler-case` is synonymous with `xcl:condition-case`.

Examples:

```
(xcl:condition-case (/ x y)
   (division-by-zero () nil))

(xcl:condition-case (open *the-file*
                          :direction :input)
   (file-error (condition)
      (format t "~&Open failed: ~A~%" condition)))

(xcl:condition-case (some-user-function)
   (file-error (condition) condition)
   (division-by-zero () 0)
   ((or unbound-variable undefined-function) ()
      'unbound))

(xcl:condition-case (open my-file)
   (file-error ()
     (format *error-output* "Couldn't open ~S."
       my-file))
   (:no-error (stream)
     (open-more-files my-file stream) stream)))
```

Note the difference between `xcl:condition-case` and `conditions:handler-bind`. In `conditions:handler-bind`, you are specifying functions that will be called in the dynamic context of the condition signalling form. In `xcl:condition-case`, you are specifying continuations to be used instead of the original form if a condition of a particular type is signaled. These continuations will be executed in the same dynamic context as the original form.

conditions:ignore-errors `&`**body** *forms* [Macro]

Executes the forms in a context that handles conditions of type `error` by returning control to this form. If no error is signaled, all

values returned by the last form are returned by `conditions:ignore-errors`. Otherwise, the form returns the two values `nil` and the condition that was signaled. Synonym for

```
(xcl:condition-case (progn . forms)
   (error (condition)
      (values nil condition)).
```

~~`xcl:debug &optional` *datum* `&rest` *arguments* [Function]~~

~~Enters the debugger with a given condition without signalling that condition. When the debugger is entered, it will announce the condition by invoking the condition's report function.~~

~~*datum* is treated the same as for `xcl:signal` except if *datum* is not specified, it defaults to "`Call to DEBUG`".~~

~~This function will never directly return to its caller. Return can occur only by a special transfer of control, such as to a `catch`, `block`, `tagbody`, `xcl:proceed-case` or `xcl:catch-abort`.~~

conditions:invoke-debugger *condition* [Function]

Invokes the debugger with the given condition. This is intended to be used as a portable entry point to the debugger. For finer control over the debugging state, see the function `xcl:debugger`.

break &optional *format-string* `&`**rest** *format-arguments* [Function]

Enters the debugger with a simple condition with the given arguments. If no *format-string* is provided, it defaults to "`Break.`" Computation may be continued by invoking the `conditions:continue` restart. If continued, `break` returns `nil`.

`break` is approximately:

```
(defun break (&optional (format-string "Break")
                &rest format-arguments)
   (conditions:restart-case (conditions:invoke-debugger
(conditions:make-conditions 'conditions:simple-condition
:format-string format-string :format-arguments format-
arguments)
      (conditions:continue ()
            :report "Return from BREAK."
         nil)))
```

Restarts

conditions:restart-case *expression* {`(`*case-name arglist* {*keyword value*}* {*form*}*`)`}* [Macro]

The *expression* is evaluated in a dynamic context where the case clauses have special meanings as points to which control may be transferred. If *expression* runs to completion, all values returned by the form are simply returned by the `conditions:restart-case` form. On the other hand, the computation of *expression* may choose to transfer control to one of the restart clauses. If a transfer to a clause occurs, the forms in the body of that clause will be

evaluated in the same dynamic context as the `conditions:restart-case` form, and any values returned by the last such form will be returned by the `conditions:restart-case` form.

A restart clause has the form given above:

(*case-name arglist* {*keyword value*}* {*form*}*)

The *case-name* may be `nil` or any symbol.

The *arglist* is a normal lambda list that will be bound and evaluated in the dynamic context of the `conditions:restart-case` form. They will use whatever values were provided by `conditions:invoke-restart` or `conditions:invoke-restart-interactively`. Definitions of these two functions appear later in this section.

The valid *keyword*/*value* pairs are:

`:filter` *expression*

expression should be suitable as an argument to the `function` special form. It defines a predicate of no arguments that determines if this clause is visible to `conditions:find-restart`. Default = true.

`:condition` *type*

Shorthand for a common special case of `:filter`. The following two *key/value* pairs are equivalent:

```
:condition foo

:filter
   (lambda ()
      (typep xcl:*current-condition*
             'foo))
```

`:interactive` *expression*

The *expression* must be a form suitable as an argument to `function`. (`function` *expression*) will be evaluated in the current lexical and dynamic environments. The result should be a function of no arguments which returns a list of values to be used by `conditions:invoke-restart-interactively`. This function will be called in the dynamic environment available prior to any restart attempt. Any interaction with the user should be done here and not in the body of the restart.

If there is no `:interactive` option specified and the restart is invoked interactively, no arguments will be supplied.

`:report` *expression*

The *expression* can either be a constant string or a form suitable as an argument to `function`.

If expression is not a string, (`function` *expression*) will be evaluated in the current lexical and dynamic environment. The

result should be a function of one argument, a stream, which will be called to report that restart. This function should print a short summary of the action that restart will take if invoked.

If *expression* is a string, it is a shorthand for `(lambda (s) (format s` *expression*`))`.

Only one of `:condition` or `:filter` may be specified. If no `:report` is specified, the *case-name* will be used. It is an error to have a null case name and no report function.

Examples:

```
(loop
  (conditions:restart-case
      (return (apply function some-args))
    (new-function (new-fn)
      :report "Use a different function."
      :interactive (lambda ()
                     (list (prompt-for 'function "Function:
")))
      (setq function new-fn))))

(loop
  (conditions:restart-case
      (return (apply function some-args))
    (nil (new-fn)
      :report "Use a different function."
      :interactive (lambda ()
                     (list (prompt-for 'function "Function:
")))
      (setq function new-fn))))

(conditions:restart-case (a-command-loop)
   (return-from-command-level ()
         :report
            (lambda (stream)
              (format stream "Return from command level ~D."
level))
      nil))

(loop
   (conditions:restart-case (another-computation)
      (conditions:continue () nil)))
```

The first and second examples are equivalent from the point of view of someone using the interactive debugger, but differ in one important aspect for non-interactive handling. If a handler "knows about" restart names, as in:

```
(when (conditions:find-restart 'new-function)
   (conditions:invoke-restart 'new-function the-
replacement))
```

then only the first example, and not the second, will have control transferred to its correction clause.

Here's a more complete example:

```
(let ((my-food 'milk)
      (my-color 'greenish-blue))
   (do ()
```

```
        ((not (food-colorable-p my-food
                                my-color)))
      (conditions:restart-case (error 'bad-food-color
                                :food my-food
                                :color my-color)
        (use-food (new-food)
           :report "Use another food."
           (setf my-food new-food))
        (use-color (new-color)
           :report "Use another color."
           (setf my-color new-color))))
   ;; We won't get to here until my-food
   ;; and my-color are compatible.
   (list my-food my-color))
```

Assuming that `use-food` and `use-color` have been defined as

```
(defun use-food (new-food)
  (invoke-restart 'use-food new-food))

(defun use-color (new-color)
  (invoke-restart 'use-color new-color))
```

then a handler can proceed from the error in either of two ways. It may correct the color or correct the food. For example:

```
#'(lambda (condition) ...
     ;; Corrects color
     (use-color 'white)  ...)
```

or

```
#'(lambda (condition) ...
     ;; Corrects food
     (use-food 'cheese) ...)
```

Here is an example using `conditions:handler-bind` and `conditions:restart-case`.

```
(conditions:handler-bind ((foo-error
                           #'(lambda (condition)
                               (conditions:use-value 7))))
   (conditions:restart-case (error 'foo-error)
      (conditions:use-value (x) (* x x))))
```

The above form returns `49`.

~~`xcl:define-proceed-function` *name* [Macro]~~
~~{*keyword value*}*~~
~~{*variable*}*~~

~~Valid *keyword*/*value* pairs are the same as those which are defined for the `xcl:proceed-case` special form. That is, `:filter`, `:filter-function`, `:condition`, `:report`, and `:report-function`. The filter and report functions specified in a `xcl:define-proceed-function` form will be used for `xcl:proceed-case` clauses with the same name that do not specify their own filter or report functions, respectively.~~

~~This form defines a function called `name` which will invoke a proceed case with the same name. The proceed function takes optional arguments which are given by the *variables* specification. The parameter list for the proceed function will look like~~

(&optional . *variables*)

The only thing that a proceed function really does is collect values to be passed on to a proceed case clause.

Each element of *variables* has the form *variable-name* or (*variable-name initial-value*). If *initial-value* is not supplied, it defaults to `nil`.

For example, here are some possible proceed functions which might be useful in conjunction with the `bad-food-color` error we used as an example earlier:

```
(xcl:define-proceed-function use-food
       :report "Use another food."
    (food (read-typed-object 'food
                             "Food to use instead: ")))

(xcl:define-proceed-function use-color
       :report "Change the food's color."
    (color
       (read-typed-object 'food
            "Color to make the food: ")))

(defun maybe-use-water (condition)
    ;; A sample handler
    (when (eq (bad-food-color-food condition)
              'milk)
       (use-food 'water)))

(xcl:handler-bind ((bad-food-color
                                  #'maybe-use-water))
    ...)
```

If a named proceed function is invoked in a context in which there is no active proceed case by that name, the proceed function simply returns `nil`. So, for example, in each of the following pairs of handlers, the first is equivalent to the second but less efficient:

```
#'(lambda (condition)              ; OK, but slow
      (when (xcl:find-proceed-case 'use-food)
         (use-food 'milk)))
#'(lambda (condition)              ; Preferred
      (use-food 'milk))

#'(lambda (condition)
      (cond ((xcl:find-proceed-case 'use-food)
                (use-food 'chocolate))
            ((xcl:find-proceed-case 'use-color)
                (use-color 'orange))))
#'(lambda (condition)
      (use-food 'chocolate)
      (use-color 'orange))
```

conditions:restart-bind ({(*name function* {*keyword value*}*)}* {*form*}* [Macro]

Executes the *form*s in a dynamic context where the given restart bindings are in effect.

name may be `nil` to indicate an anonymous restart, or some other symbol to indicate a named restart.

function will be evaluated in the current lexical and dynamic contexts and should produce a function of no arguments to be used to perform the restart. This function will be called when that restart is activated by `conditions:invoke-restart` or `conditions:invoke-restart-interactively`. Note that unlike `conditions:restart-case`, invoking the restart does not automatically transfer control back to the contour in which it was established. If that is appropriate for that restart it is up to the individual restart function to do this.

The valid *keyword*/*value* pairs are:

`:interactive-function` *form*

form will be evaluated in the current lexical and dynamic environments and should produce a function of no arguments that will construct the list of values to be used by `conditions:invoke-restart-interactively`.

`:report-function` *form*

form will be evaluated in the current lexical and dynamic environments and should produce a function of one argument, a stream, that will be used to report that restart.

`:filter-function` *form*

form will be evaluated in the current lexical and dynamic environments and should produce a function of no arguments that will be used to determine if the given restart is currently active.

This form is a more primitive way of establishing restarts than *conditions:restart-case*. It is expected that *conditions:restart-case* will be sufficient for most uses of the restart facility. An example of where the more general facility provided by *conditions:restart-bind* may be useful is:

```
(conditions:restart-bind ((nil #'(lambda () (expunge-directory the-dir)) :report-function #'(lambda (stream) (format stream "Expunge ~A." (directory-namestring the-dir))))) (cerror "Try this file operation again." 'directory-full :directory the-dir))
```

In this case, a restart is provided that allows the user to expunge the full directory and return to the debugger after doing so. He can then try some other restart, such as `conditions:continue` to retry the failed operation.

conditions:compute-restarts [Function]

Uses the dynamic state of the program to compute a list of *restarts*.

Each restart object represents a point in the current dynamic state of the program to which control may be transferred. The only operations that Lisp defines for such objects are:

`conditions:restart-name`,
`conditions:find-restart`,
`conditions:invoke-restart`, `conditions:invoke-restart-interactively`,
`princ`, and
`prin1`,

to identify an object as a restart using `(typep` *x* `'conditions:restart)`, and standard Lisp operations that work for all objects, such as `eq`, `eql`, `describe`, etc.

The list which results from a call to `conditions:compute-restarts` is ordered so that the innermost (ie, more-recently established) restarts are nearer the head of the list.

Note also that `conditions:compute-restarts` returns *all* valid restarts, even if some of them have the same name as others and therefore would not be found by `conditions:find-restart`.

It is an error to modify the list returned by `conditions:compute-restarts`.

conditions:restart-name *restart* [Function]

Returns the name of the given *restart*, or `nil` if it is not named.

~~`xcl:default-proceed-test` *proceed-case-name* [Macro]~~

~~Returns the default filter function for proceed-cases with the given *proceed-case-name*. May be used with `setf` to change it.~~

~~`xcl:default-proceed-report` *proceed-case-name* [Macro]~~

~~Returns the default report function for proceed-cases with the given *proceed-case-name*. This may be a string or a function just as for condition types. May be used with `setf` to change it.~~

conditions:find-restart *identifier* [Function]

Searches for a restart by the given *identifier* which is in the current dynamic environment.

If *identifier* is a symbol, then the innermost (ie, most recently established) restart with that name that is active is returned. `nil` is returned if no such restart is found.

If *identifier* is a restart object, then it is simply returned unless it is not currently valid for use. In that case, `nil` is returned.

When searching for a matching restart, the filter function, if any, of potential matches will be called to see if they are active. If it returns

`nil`, then the restart is considered to not have been seen and the search for a match continues.

Although anonymous restarts have a name of `nil`, it is an error for the symbol `nil` to be given as an *identifier* to this function. If it is approriate to search for anonymous restarts, you should use `conditions:compute-restarts` instead.

conditions:invoke-restart *restart* `&`**rest** *values* [*Function*]

Calls the function associated with the given *restart*, passing the *values* as arguments. The *restart* must be a restart object or the non-null name of a restart which is valid in the current dynamic context. If an argument is not valid, an error of type `conditions:control-error` will be signalled.

~~If the argument is a named proceed case that has a corresponding proceed function, `xcl:invoke-proceed-case` will do the optional argument resolution specified by that function before transferring control to the proceed case.~~

conditions:invoke-restart-interactively *restart* [Function]

Calls the function associated with the given *restart*, providing for any necessary arguments. The *restart* must be a restart object or the non-null name or a restart which is valid in the current dynamic context. If the *restart* is not valid, an error of type `conditions:control-error` will be signalled.

`conditions:invoke-restart-interactively` will first call the *restart*'s interactive function as specified by the `:interactive` keyword of `conditions:restart-case` or the `:interactive-function` keyword of `conditions:restart-bind`. The interactive function should return a list of values to be passed as arguments to the *restart*. This list must be at least as long as the number of required arguments that the *restart* has.

If the *restart* has no interactive function, no arguments will be passed to the restart function. It is an error for a restart to require arguments but not have an interactive function.

Once the arguments have been determined, `conditions:invoke-restart-interactively` will simply do `(apply #'conditions:invoke-restart` *restart arguments*`)`.

conditions:with-simple-restart `(`*name format-string* `{`*format-arguments*`}*)` `{`*form*`}*`

[Macro]

This is a shorthand for one of the most common uses of `conditions:restart-case`.

If the *restart* designated by *name* is not invoked while executing the *form*s, all values produced by the last *form* are returned. If the restart established by `conditions:with-simple-restart` is

invoked, control is transferred to the `conditions:with-simple-restart` form, which immediately returns the two values `nil` and `t`.

It is permissible for *name* to be `nil`. In that case, an anonymous restart is established.

`conditions:with-simple-restart` is essentially:

```
(defmacro conditions:with-simple-restart
    ((restart-name format-string
      &rest format-arguments)
     &body forms)
  `(conditions:restart-case (progn ,@forms)
     (,restart-name ()
         :report (lambda (stream)
                   (format stream
                     ,format-string
                     ,@format-arguments))
       (values nil t))))
```

Example:

```
(defun read-eval-print-loop (level)
  (conditions:with-simple-restart
      (conditions:abort "Exit command level ~D." level)
    (loop
      (conditions:with-simple-restart
          (conditions:abort "Return to command level ~D."
level)
        (print (eval (read)))))))
```

xcl:catch-abort *print-form* &**body** *forms* [Macro]

Like `conditions:with-simple-restart`, but always uses the name `conditions:abort` .

`xcl:catch-abort` could be defined by:

```
(defmacro xcl:catch-abort (print-form
                           &body forms)
   `(conditions:with-simple-restart
        (conditions:abort ,print-form)
      ,@forms))
```

conditions:abort [Function]

This function transfers control to the nearest active restart named `conditions:abort`. If there is none, this function signals an error of type `conditions:control-error`.

~~`xcl:abort` could be defined by:~~

```
~~(define-proceed-function xcl:abort
    :report "Abort")~~
```

conditions:continue [Function]

This function transfers control to the nearest active restart named `conditions:continue`. If none exists it simply returns `nil`.

The `conditions:continue` restart is generally part of simple protocols where there is a single "obvious" way to continue, such as in `break` and `cerror`.

NB: `conditions:continue` replaces `xcl:proceed`.

~~`xcl:proceed &optional` *condition* [Function]~~

~~This is a predefined proceed function. It is used by such functions as `break`, `cerror`, etc.~~

conditions:muffle-warning [Function]

This function transfers control to the nearest active restart named `conditions:muffle-warning`. If none exists, an error of type `conditions:control-error` is signalled.

`warn` sets up this restart so that handlers of `conditions:warning` conditions have a way to tell `warn` that the warning has been dealt with and that no further action is warranted.

conditions:use-value *new-value* [Function]

This function transfers control (and one value) to the nearest active restart named `conditions:use-value`. If no such restart exists, this function simply returns `nil`.

The `conditions:use-value` restart is generally used by handlers trying to recover from errors of types such as `conditions:cell-error`, where the handler may wish to supply a replacement datum for one-time use.

conditions:store-value *new-value* [Function]

This function transfers control (and one value) to the nearest active restart named `conditions:store-value`. If no such restart exists, this function simply returns `nil`.

The `conditions:use-value` restart is generally used by handlers trying to recover from errors of types such as `conditions:cell-error`, where the handler may wish to supply a replacement datum to be stored in the offending cell.

[This page intentionally left blank]