CHAPTER 24 ERROR SYSTEM

This chapter replaces most of Chapter 24, Errors, of *Common Lisp, the Language*.

The Xerox Common Lisp error system is based on proposal number 8 for the Common Lisp error system. Deviations from this proposal are noted. In particular, proceeding and proceed functions are more like those in an earlier proposal. 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.

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.

proceed 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 *proceed* from a condition. Among other things, the use of these structured primitives for proceeding allow 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 Serious conditions which result from incorrect programs or data are called *errors*. Not all serious conditions are errors, however. Storage conditions are examples of serious 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 `xcl:condition`. That is,

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

is true if c is a condition.

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

`xcl: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 `xcl:handler-bind`.

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 happened, the returned values are ignored and the effect is the same as if the handler had been invisible to the mechanism seeking to find a handler. 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 `xcl:invoke-proceed-case`.
- It may signal another condition.
- It may invoke the interactive debugger.

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

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`.

proceed type A proceed case with a particular name, or a particular set of proceed cases that share an interface defined by a *proceed function*, are sometimes called a *proceed type*.

report In some cases, it may be useful to *report* a condition or a proceed case 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 proceed case.

Program Interface to the Condition System

Defining and Creating Conditions

`xcl:define-condition` *name parent-type* [*Macro*]
*{keyword value}**
*{slots}**

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 *keyword/value* pairs are:

`: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.

`: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` *form*

A short form of `:report-function` to cover two common cases.

If *form* is a constant string, this is the same as

```
:report-function
   (lambda (ignore stream)
      (write-string form stream))
```

Otherwise, this is the same as

```
:report-function
   (lambda (condition
            *standard-output*)
      form)
```

In the latter case, the form describes how to print objects of the type being defined. The form should do output to standard output. The condition being printed will be the value of the variable `condition` (the symbol `condition` in this usage is in the same package as the name of the new condition type). The condition's slots are accessible as simple variables within the report form.

`: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` *form*

An expression to be used as the body of a default handler for this condition type. While executing *form*, the variable `condition` will be bound to the condition being handled (as with `:report` above, the symbol `condition` in this usage is in the same package as the name of the new condition type). That is, this defines a function

```
(lambda (condition)
   form)
```

as the default handler for that type.

It is an error to specify both `:report-function` and `:report` in the same `xcl:define-condition` form. It is also an error to specify both `:handler-function` and `:handle`. If neither `:report-function` nor `:report` is specified, information about how to print this type of condition will be inherited from the `parent-type`. If neither `:handler-function` nor `:handle` was specified, there will be no default handler for the new condition type.

slots is a list of *slot-descriptions*, and specifies slots to be used by the given type. In addition to those specified, the slots of the *parent-type* are also available. A *slot-description* is exactly the same as for `defstruct` except that no slot-options are allowed, only an optional default-value expression. Condition objects are immutable, i.e., all of their slots are declared to be `:read-only`.

`xcl: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:

```
(xcl:define-condition bad-food-color food-lossage
      :report (format t "The food ~A was ~A"
                        food color)
   food
   color)
```

defines an error of type `bad-food-color` which inherits from the `food-lossage` condition type. The new type has slots `food` and `color` so that `xcl: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`:

```
(xcl:define-condition machine-error error
      :report (format t
                      "There is a problem with ~A."
                      machine-name)
   machine-name)
```

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

```
(xcl:define-condition machine-not-available-error
                      machine-error
    :report (format t
                "The machine ~A is not available."
                machine-name)
   machine-name)
```

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:

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

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 the report function is inherited. `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 the default handler for that type is inherited. `setf` may be used with this form to change the default handler for a condition type.

`xcl: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,

```
(xcl: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 (`xcl:signal`, `error`, `cerror`, and `warn`) to the condition being signaled. This is especially useful in proceed case filters. The top-level value of `xcl:*current-condition*` is `nil`.

`xcl:signal` *datum* `&rest` *arguments* [*Function*]

Invokes the signal facility on a condition. If the condition is not handled, `xcl: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 `xcl:arguments` to be non-`nil`.

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

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

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

```
(xcl:make-condition
   'xcl: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.

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

Like `xcl: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 `xcl:signal`. If *datum* is a string, a conditon of type `xcl: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 proceed type `xcl:proceed`, which will simply return the condition being signalled from `cerror`.

`cerror` is used to signal continuable errors. Like `error`, it signals an error and enters the debugger. However, `cerror` allows the program to be continued from the debugger after resolving the error.

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 output to `*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 signaled.

datum the same as for signal except that if *datum* is a string, a condition of type `xcl:simple-warning` is made.

The eventual condition type resulting from *datum* must be a subtype of `xcl:warning`.

`*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 `xcl:simple-error` conditions. The default clauses of `etypecase` and `ctypecase` forms signal `xcl:type-mismatch` conditions. `assert` signals the `xcl:assertion-failed` condition. `ccase` and `ctypecase` set up a `xcl:store-value` proceed case.

Handling Conditions

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

Executes the forms in a dynamic context where the given local handler *bindings* are in effect. The *bindings* must take the form (*type 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 `xcl:handler-bind` form first.

type may be the name of a condition type or a list of condition types.

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 `xcl:handler-bind` appears at the end of the `xcl:proceed-case` macro description.

`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 signaled, 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.

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)
   ((xcl:unbound-variable xcl:undefined-function) ()
      'unbound))
```

Note the difference between xcl:condition-case and xcl:handler-bind. In xcl: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.

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

Executes the forms in a context that handles errors of type `error` by returning control to this form. If no error is signaled, all values returned by the last form are returned by `xcl:ignore-errors`. Otherwise, the form returns `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`.

`break` `&optional` *datum* `&rest` *arguments* [*Function*]

Like `xcl:debug` except sets up a proceed case like `cerror`.

If *datum* is not specified, it defaults to `"Break"`.

If the `break` is proceeded, the value returned is the condition that was used.

`break` is approximately:

```
(defun break (&optional (datum "Break")
                &rest arguments)
   (xcl:proceed-case (apply #'xcl:debug datum
                                        arguments)
      (xcl:proceed (condition)
           :report "Return from BREAK."
       condition)))
```

Proceed Cases

`xcl:proceed-case` *form* `&rest` *clauses* [*Macro*]

The *form* is evaluated in a dynamic context where the clauses have special meanings as points to which control may be transferred. If *form* runs to completion, all values returned by the form are simply returned by the `xcl:proceed-case` form. On the other hand, the computation of forms may choose to transfer control to one of the proceed case 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 `xcl:proceed-case` form, and any values returned by the last such form will be returned by the `xcl:proceed-case` form.

A proceed case clause has the form:

(*proceed-function-name arglist* {*keyword value*}* {*body-form*}*)

The *proceed-function-name* may be `nil` or any symbol, usually the name of a defined proceed function. `xcl:define-proceed-function` will be described later.

The *arglist* is a list of optional argument specifications that will be bound and evaluated in the dynamic context of the `xcl:proceed-case` form. They will use whatever values were provided by `xcl:invoke-proceed-case`.

The valid *keyword/value* pairs are:

`:filter-function` *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 `xcl:find-proceed-function`.

`:filter` *form*

A shorthand form of `:filter-function` that is equivalent to

```
:filter-function (lambda () form)
```

`:condition` *type*

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

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

`:report-function` *expression*

The *expression* must be an appropriate argument to the `function` special form, and should designate a function of two arguments, a proceed case and a stream, that writes to the stream a summary of the action that this proceed case will take if invoked..

`:report` *form*

This is a shorthand for two important special cases of `:report-function`. If *form* is a constant string, then this is the same as:

```
:report-function
   (lambda (ignore stream)
      (write-string form stream))
```

Otherwise, this is the same as

```
:report-function
   (lambda (xcl:proceed-case
            *standard-output*)
      form)
```

In the latter case, form must do output to `*standard-output*`, summarizing the action that this proceed case will take if invoked. The proceed-case will be bound to the variable `xcl:proceed-case`.

Only one of `:condition`, `:filter` or `:filter-function` may be specified. Only one of `:report` or `:report-function` may be specified.

If a named proceed function has a default filter and the proceed case specifies a filter, then the information supplied in the proceed case takes precedence. Similarly, if `:report` or `:report-function` is specified in the proceed case, then only that information is considered, and any `:report` or `:report-function` specified as a default for the named proceed function is not used.

If a named proceed function is used but no report information is supplied, the name of the proceed function is used to generate the default help information. It is an error if no named proceed case is

used and no report information is provided; this means that you must always have a way of describing to the user how to proceed. If you don't specify report methods, make sure that the name of the proceed type is something sensible.

When `*print-escape*` is `nil`, the printer will use the report information for a proceed case.

Examples:

```
(xcl:proceed-case (a-random-computation)
   (new-function (new-function)
      (setq function new-function)))

(xcl:proceed-case (a-random-computation)
   (nil ((new-function (read-typed-object
                                  'function
                                  "Function: ")))
        :report "Use a different function."
        :condition undefined-function
      (setq function new-function)))

(xcl:proceed-case (a-command-loop)
   (return-from-command-level ()
         :report
            (format t
                 "Return from command level ~D."
                 level)
      nil))

(loop
   (xcl:proceed-case (another-computation)
      (xcl:proceed ())))
```

Assuming that `new-function` is defined as a proceed function with defaults:

```
:report "Use a different function."
:condition xcl:undefined-function
```

then 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" proceed function names, as in:

```
(when (xcl:find-proceed-case 'new-function
                             condition)
   (new-function condition 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 (bad-food-color-p food
                               color)))
     (xcl:proceed-case (error 'bad-food-color
                              :food my-food
                              :color my-color)
        (use-food (new-food)
           (setf my-food new-food))
        (use-color (new-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))
```

A handler can then proceed 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 `xcl:handler-bind` and `xcl:proceed-case`.

```
(xcl:handler-bind ((foo-error
                         #'(lambda (condition)
                             (xcl:use-value 7))))
   (xcl:proceed-case (error 'foo-error)
      (xcl: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))
```

`xcl:compute-proceed-cases` [*Function*]

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

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

`xcl:proceed-case-name`,
`xcl:find-proceed-case`,
`xcl:invoke-proceed-case`,
`princ`, and
`print`,

the identification of an object as a proceed case using `(typep` *x* `'proceed-case)`, and standard Lisp operations that work for all objects, such as `eq`, `eql`, `describe`, etc.

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

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

`xcl:proceed-case-name` *proceed-case* [*Function*]

Returns the name of the given *proceed-case*, 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.

`xcl:find-proceed-case` *name* [*Function*]

Searches for a proceed case by the given *name* which is in the current dynamic contour. This is determined by calling the proceed case's filter function.

If *name* is a proceed function name, then the innermost (ie, most recently established) proceed case with that function name that is active 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.

`xcl:invoke-proceed-case` *proceed-case* `&rest` *values* [*Function*]

Transfers control to the given *proceed-case*, passing it the given *values*. The *proceed-case* must be a proceed case object or the name of a proceed case which is valid in the current dynamic context. If the argument is not valid, the error `xcl:bad-proceed-case` will be signaled. 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.

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

Sets up a proceed case named `xcl:abort`.

If no call to the proceed function `xcl:abort` is made while executing *forms* and they return normally, all values returned by the last form in *forms* are returned. If an `xcl:abort` transfers control to this `xcl:catch-abort`, two values are returned: `nil` and the condition that was given to `xcl:abort` (or `nil` if none was given).

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

```
(defmacro xcl:catch-abort (print-form
                           &body forms)
   `(xcl:proceed-case (progn ,@forms)
       (xcl:abort (condition)
            :report ,print-form
         (values nil condition))))
```

Example:

```
(defun read-eval-print-loop (level)
   (xcl:catch-abort
         (format t "Exit command level ~D."
                   level)
      (loop
         (xcl:catch-abort
               (format t
                     "Return to command level ~D."
                     level)
            (print (eval (read)))))))
```

`xcl:abort &optional` *condition* [*Function*]

This is a predefined proceed function that transfers control to the innermost (dynamic) visible proceed case named `xcl:abort`.

`xcl:abort` could be defined by:

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

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

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

`xcl:use-value &optional` *new-value* [*Function*]

This is a predefined proceed function. It is intended to be used for supplying an alternate value to be used in a computation. If *new-value* is not provided, `xcl:use-value` will prompt the user for one.

`xcl:store-value &optional` *new-value* [*Function*]

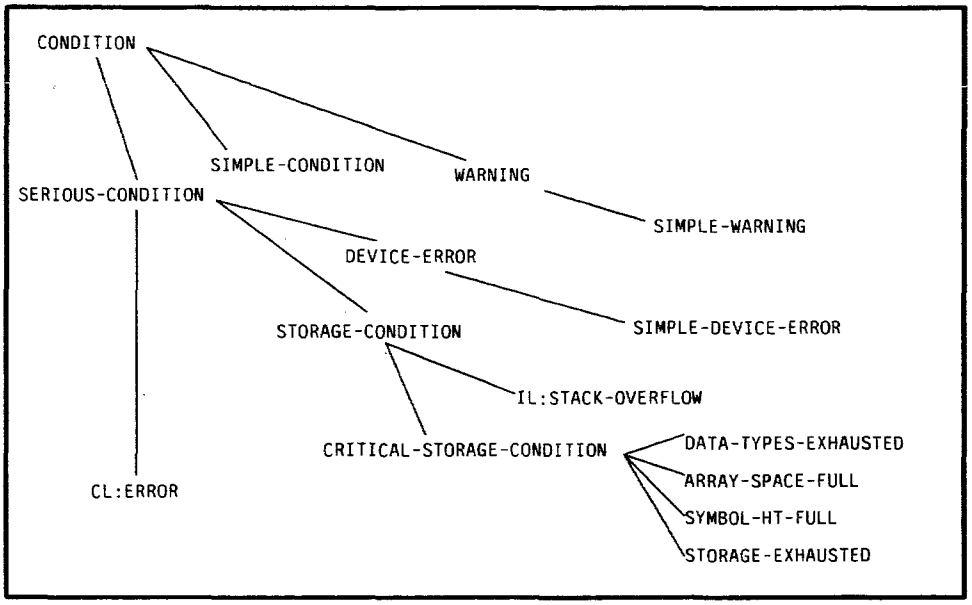
This is a predefined proceed function. It is intended to be used for supplying an alternate value to store in some location as a way of proceeding from an error. The proceed function `xcl:store-value` does not actually store the new value anywhere: it is up to proceed case to take care of that. If *new-value* is not provided, `xcl:store-value` will prompt the user for one. `xcl:store-value` is used by such forms as `check-type` and `cerror`.

Predefined Types

`xcl:proceed-case` *[Type]*

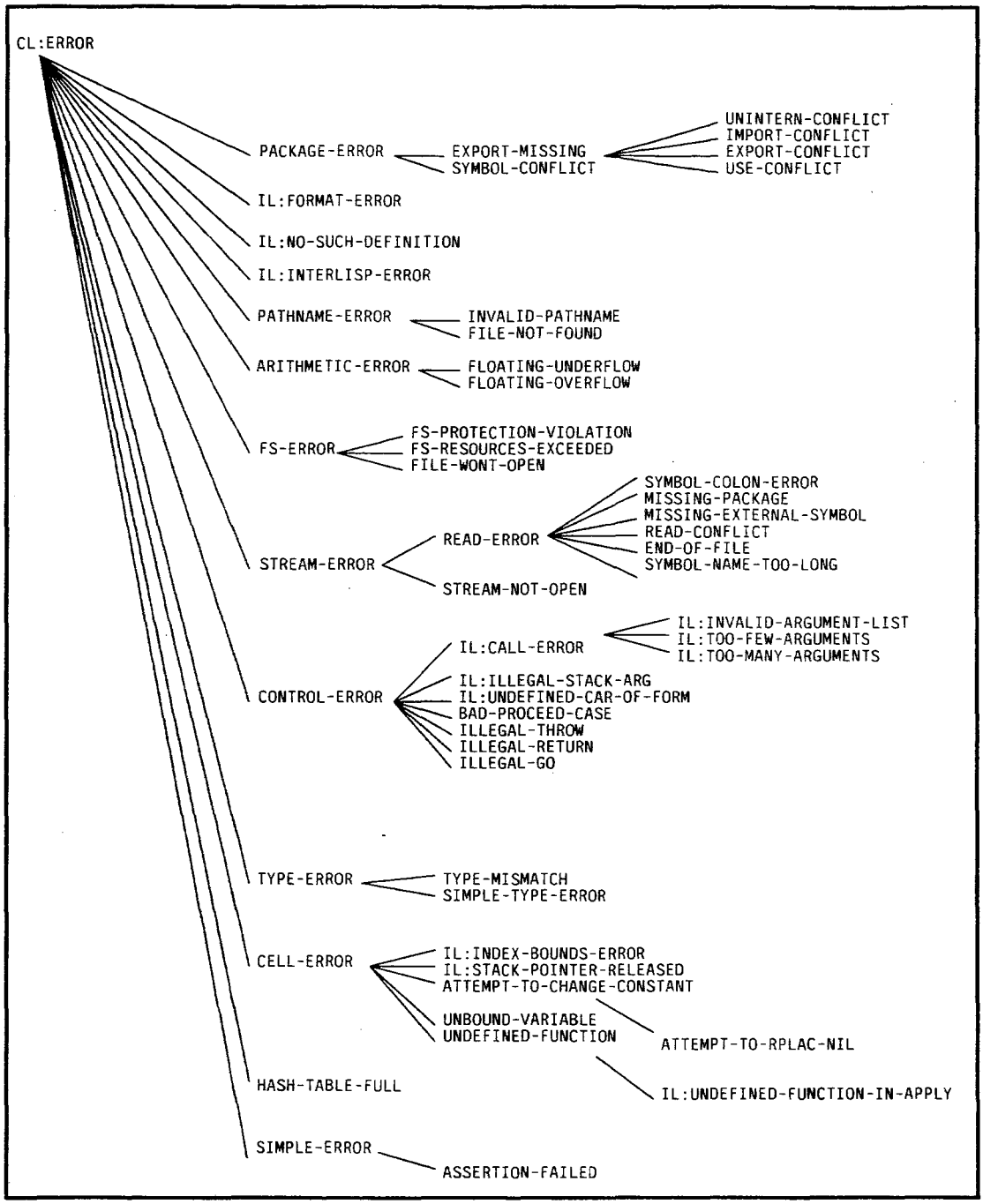
This is the data type used to represent a proceed case.

The condition type hierarchy looks like this:



All condition types shown in the graph above, and in the one that follows, are in the XCL package, unless otherwise qualified.

The hierarchy continues on the next page.


CL:ERROR
PACKAGE-ERROR
EXPORT-MISSING
SYMBOL-CONFLICT
UNINTERN-CONFLICT
IMPORT-CONFLICT
EXPORT-CONFLICT
USE-CONFLICT
IL:FORMAT-ERROR
IL:NO-SUCH-DEFINITION
IL:INTERLISP-ERROR
PATHNAME-ERROR
INVALID-PATHNAME
FILE-NOT-FOUND
ARITHMETIC-ERROR
FLOATING-UNDERFLOW
FLOATING-OVERFLOW
FS-ERROR
FS-PROTECTION-VIOLATION
FS-RESOURCES-EXCEEDED
FILE-WONT-OPEN
STREAM-ERROR
READ-ERROR
SYMBOL-COLON-ERROR
MISSING-PACKAGE
MISSING-EXTERNAL-SYMBOL
READ-CONFLICT
END-OF-FILE
SYMBOL-NAME-TOO-LONG
STREAM-NOT-OPEN
CONTROL-ERROR
IL:CALL-ERROR
IL:INVALID-ARGUMENT-LIST
IL:TOO-FEW-ARGUMENTS
IL:TOO-MANY-ARGUMENTS
IL:ILLEGAL-STACK-ARG
IL:UNDEFINED-CAR-OF-FORM
BAD-PROCEED-CASE
ILLEGAL-THROW
ILLEGAL-RETURN
ILLEGAL-GO
TYPE-ERROR
TYPE-MISMATCH
SIMPLE-TYPE-ERROR
CELL-ERROR
IL:INDEX-BOUNDS-ERROR
IL:STACK-POINTER-RELEASED
ATTEMPT-TO-CHANGE-CONSTANT
UNBOUND-VARIABLE
UNDEFINED-FUNCTION
ATTEMPT-TO-RPLAC-NIL
IL:UNDEFINED-FUNCTION-IN-APPLY
HASH-TABLE-FULL
SIMPLE-ERROR
ASSERTION-FAILED

The types that are non-terminals in the above tree:
`xcl:condition`,
`xcl:warning`,
`xcl:serious-condition`,
`xcl:storage-condition`,
`error`,
`xcl:control-error`, etc.
are provided primarily for type inclusion purposes. Normally, they would not be directly instantiated.

In the descriptions of condition types below, the names in italics on the first line of each description are the names of the slots defined for that condition type.

`xcl:condition` [*Condition*]

All types of conditions, whether `error` or non-`error`, must inherit from this type.

`xcl:warning` [*Condition*]

All types of warnings should inherit from this type. This is a subtype of `condition`.

`xcl:serious-condition` [*Condition*]

Any condition, whether error or non-error, which should enter the debugger when signalled but not handled should inherit from this type. This is a subtype of `xcl:condition`.

Note: `ignore-errors` will ignore conditions of type `error`, not of type `xcl:serious-condition`. Conditions which are serious conditions but not errors are typically those that may require more sophisticated handling than simply being ignored. For example, `xcl:ignore-errors` will not ignore an `xcl:storage-condition`, which is a serious condition but is not generally a program error.

Compatibility Note: `serious-condition` is similar to Zetalisp's `dbg:debugger-condition`.

`error` [*Condition*]

All types of error conditions inherit from this condition. This is a subtype of `xcl:serious-condition`.

`xcl:simple-condition` *format-string format-arguments* [*Condition*]

Conditions signalled by `xcl:signal` when given a format string as a first argument are of this type. This is a subtype of `xcl:condition`.

`xcl:simple-warning` *format-string format-arguments* [*Condition*]

Conditions signalled by `warn` when given a format string as a first argument are of this type. This is a subtype of `xcl:warning`.

`xcl:simple-error` *format-string format-arguments* [*Condition*]

Conditions signalled by `error` and `cerror` when given a format string as a first argument are of this type. This is a subtype of `error`.

`xcl:storage-condition` [*Condition*]

Conditions which relate to memory overflow conditions should inherit from this type. This is a subtype of `xcl:serious-condition`.

`xcl:stack-overflow` [*Condition*]

Conditions which relate to stack overflow should inherit from this type. This is a subtype of `xcl:storage-condition`.

`xcl:control-error` [*Condition*]

Errors in the transfer of control in a program should inherit from this type. This is a subtype of `error`.

`xcl:illegal-throw` *tag* [*Condition*]

The error which results when `throw` is given a tag which is not active should inherit from this. This is a subtype of `xcl:control-error`. *tag* is the offending tag.

`xcl:illegal-go` *tag* [*Condition*]

The error which results when `go` is given a tag which is no longer available should inherit from this. This is a subtype of `xcl:control-error`. *tag* is the offending tag.

`xcl:illegal-return` *tag* [*Condition*]

The error which results when `return-from` is given a block name which is no longer accessible should inherit from this. This is a subtype of `xcl:control-error`. *tag* is the offending block name.

`xcl:stream-error` *stream* [*Condition*]

Errors which occur during input from or output to a stream should inherit from this type. This is a subtype of `error`. The function `stream-error-stream` will access the offending stream.

`xcl:read-error` [*Condition*]

Errors which occur during an input operation on a stream should inherit from this type. This is a subtype of `xcl:stream-error`.

`xcl:end-of-file` [*Condition*]

The error which results when a read operation is done on a stream which has no more tokens should inherit from this type. This is a subtype of `read-error`.

`xcl:cell-error` *name* [*Condition*]

Errors which occur while accessing a location should inherit from this type. This is a subtype of `error`. *name* is the name of the offending cell.

`xcl:unbound-variable` [*Condition*]

The error which results from trying to access the value of an unbound variable should inherit from this type. This is a subtype of `xcl:cell-error`.

`xcl:undefined-function` [*Condition*]

The error which results from trying to access the value of an undefined function should inherit from this type. This is a subtype of `xcl:cell-error`.

[This page intentionally left blank]