9. Conditions

9.1 Condition System Concepts

Common Lisp constructs are described not only in terms of their behavior in situations during which they are intended to be used (see the "Description" part of each *operator* specification), but in all other situations (see the "Exceptional Situations" part of each *operator* specification).

A situation is the evaluation of an expression in a specific context. A *condition* is an *object* that represents a specific situation that has been detected. *Conditions* are *generalized instances* of the *class* **condition**. A hierarchy of *condition* classes is defined in Common Lisp. A *condition* has *slots* that contain data relevant to the situation that the *condition* represents.

An error is a situation in which normal program execution cannot continue correctly without some form of intervention (either interactively by the user or under program control). Not all errors are detected. When an error goes undetected, the effects can be *implementation-dependent*, *implementation-defined*, unspecified, or undefined. See Section 1.4 (Definitions). All detected errors can be represented by *conditions*, but not all *conditions* represent errors.

Signaling is the process by which a *condition* can alter the flow of control in a program by raising the *condition* which can then be *handled*. The functions **error**, **cerror**, **signal**, and **warn** are used to signal *conditions*.

The process of signaling involves the selection and invocation of a *handler* from a set of *active handlers*. A *handler* is a *function* of one argument (the *condition*) that is invoked to handle a *condition*. Each *handler* is associated with a *condition type*, and a *handler* will be invoked only on a *condition* of the *handler*'s associated *type*.

Active handlers are *established* dynamically (see **handler-bind** or **handler-case**). *Handlers* are invoked in a *dynamic environment* equivalent to that of the signaler, except that the set of *active handlers* is bound in such a way as to include only those that were *active* at the time the *handler* being invoked was *established*. Signaling a *condition* has no side-effect on the *condition*, and there is no dynamic state contained in a *condition*.

If a *handler* is invoked, it can address the *situation* in one of three ways:

Decline
: It can decline to *handle* the *condition*. It does this by simply returning rather than transferring control. When this happens, any values returned by the handler are ignored and the next most recently established handler is invoked. If there is no such handler and the signaling function is **error** or **cerror**, the debugger is entered in the *dynamic environment* of the signaler. If there is no such handler and the signaling function is either **signal** or **warn**, the signaling function simply returns **nil**.

Handle
: It can *handle* the *condition* by performing a non-local transfer of control. This can be done either primitively by using **go**, **return**, **throw** or more abstractly by using a function such as **abort** or **invoke-restart**.

Defer
: It can put off a decision about whether to *handle* or *decline*, by any of a number of actions, but most commonly by signaling another condition, resignaling the same condition, or forcing entry into the debugger.

9.1.1 Condition Types

The next figure lists the *standardized condition types*. Additional *condition types* can be defined by using **define-condition**.

```
arithmetic-error                  floating-point-overflow   simple-type-error
cell-error                        floating-point-underflow  simple-warning
condition                         package-error             storage-condition
control-error                     parse-error               stream-error
division-by-zero                  print-not-readable        style-warning
end-of-file                       program-error             type-error
error                             reader-error              unbound-slot
file-error                        serious-condition         unbound-variable
floating-point-inexact            simple-condition          undefined-function
floating-point-invalid-operation  simple-error              warning
```

Figure 9-1. Standardized Condition Types

All *condition* types are *subtypes* of *type* **condition**. That is,

```
 (typep c 'condition) =>  true
```

if and only if *c* is a *condition*.

Implementations must define all specified *subtype* relationships. Except where noted, all *subtype* relationships indicated in this document are not mutually exclusive. A *condition* inherits the structure of its *supertypes*.

The metaclass of the *class* **condition** is not specified. *Names* of *condition types* may be used to specify *supertype* relationships in **define-condition**, but the consequences are not specified if an attempt is made to use a *condition type* as a *superclass* in a **defclass** *form*.

The next figure shows *operators* that define *condition types* and creating *conditions*.

```
define-condition  make-condition
```

Figure 9-2. Operators that define and create conditions.

The next figure shows *operators* that *read* the *value* of *condition slots*.

```
arithmetic-error-operands   simple-condition-format-arguments
arithmetic-error-operation  simple-condition-format-control
cell-error-name             stream-error-stream
file-error-pathname         type-error-datum
package-error-package       type-error-expected-type
print-not-readable-object   unbound-slot-instance
```

Figure 9-3. Operators that read condition slots.

9.1.1.1 Serious Conditions

A *serious condition* is a *condition* serious enough to require interactive intervention if not handled. *Serious conditions* are typically signaled with **error** or **cerror**; non-serious *conditions* are typically signaled with **signal** or **warn**.

9.1.2 Creating Conditions

The function **make-condition** can be used to construct a *condition object* explicitly. Functions such as **error**, **cerror**, **signal**, and **warn** operate on *conditions* and might create *condition objects* implicitly. Macros such as **ccase**, **ctypecase**, **ecase**, **etypecase**, **check-type**, and **assert** might also implicitly create (and *signal*) *conditions*.

9.1.2.1 Condition Designators

A number of the functions in the condition system take arguments which are identified as *condition designators*. By convention, those arguments are notated as

datum `&rest` *arguments*

Taken together, the *datum* and the *arguments* are "*designators* for a *condition* of default type *default-type*." How the denoted *condition* is computed depends on the type of the *datum*:

* If the *datum* is a *symbol* naming a *condition type ...*
  The denoted *condition* is the result of

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

* If the *datum* is a *format control ...*
  The denoted *condition* is the result of

```
(make-condition defaulted-type
                :format-control datum
                :format-arguments arguments)
```

  where the *defaulted-type* is a *subtype* of *default-type*.

* If the *datum* is a *condition ...*
  The denoted *condition* is the *datum* itself. In this case, unless otherwise specified by the description of the *operator* in question, the *arguments* must be *null*; that is, the consequences are undefined if any *arguments* were supplied.

Note that the *default-type* gets used only in the case where the *datum string* is supplied. In the other situations, the resulting condition is not necessarily of *type default-type*.

Here are some illustrations of how different *condition designators* can denote equivalent *condition objects*:

```
(let ((c (make-condition 'arithmetic-error :operator '/ :operands '(7 0))))
  (error c))
==  (error 'arithmetic-error :operator '/ :operands '(7 0))

(error "Bad luck.")
==  (error 'simple-error :format-control "Bad luck." :format-arguments '())
```

9.1.3 Printing Conditions

If the `:report` argument to **define-condition** is used, a print function is defined that is called whenever the defined *condition* is printed while the *value* of ***print-escape*** is *false*. This function is called the *condition reporter*; the text which it outputs is called a *report message*.

When a *condition* is printed and ***print-escape*** is *false*, the *condition reporter* for the *condition* is invoked. *Conditions* are printed automatically by functions such as **invoke-debugger**, **break**, and **warn**.

When ***print-escape*** is *true*, the *object* should print in an abbreviated fashion according to the style of the implementation (e.g., by **print-unreadable-object**). It is not required that a *condition* can be recreated by reading its printed representation.

No *function* is provided for directly *accessing* or invoking *condition reporters*.

9.1.3.1 Recommended Style in Condition Reporting

In order to ensure a properly aesthetic result when presenting *report messages* to the user, certain stylistic conventions are recommended.

There are stylistic recommendations for the content of the messages output by *condition reporters*, but there are no formal requirements on those *programs*. If a *program* violates the recommendations for some message, the display of that message might be less aesthetic than if the guideline had been observed, but the *program* is still considered a *conforming program*.

The requirements on a *program* or *implementation* which invokes a *condition reporter* are somewhat stronger. A *conforming program* must be permitted to assume that if these style guidelines are followed, proper aesthetics will be maintained. Where appropriate, any specific requirements on such routines are explicitly mentioned below.

9.1.3.1.1 Capitalization and Punctuation in Condition Reports

It is recommended that a *report message* be a complete sentences, in the proper case and correctly punctuated. In English, for example, this means the first letter should be uppercase, and there should be a trailing period.

```
(error "This is a message")  ; Not recommended
(error "this is a message.") ; Not recommended

(error "This is a message.") ; Recommended instead
```

9.1.3.1.2 Leading and Trailing Newlines in Condition Reports

It is recommended that a *report message* not begin with any introductory text, such as "`Error: `" or "`Warning: `" or even just *freshline* or *newline*. Such text is added, if appropriate to the context, by the routine invoking the *condition reporter*.

It is recommended that a *report message* not be followed by a trailing *freshline* or *newline*. Such text is added, if appropriate to the context, by the routine invoking the *condition reporter*.

```
(error "This is a message.~%")   ; Not recommended
(error "~&This is a message.")   ; Not recommended
(error "~&This is a message.~%") ; Not recommended

(error "This is a message.")     ; Recommended instead
```

9.1.3.1.3 Embedded Newlines in Condition Reports

Especially if it is long, it is permissible and appropriate for a *report message* to contain one or more embedded *newlines*.

If the calling routine conventionally inserts some additional prefix (such as "`Error: `" or "`;; Error: `") on the first line of the message, it must also assure that an appropriate prefix will be added to each subsequent line of the output, so that the left edge of the message output by the *condition reporter* will still be properly aligned.

```
(defun test ()
  (error "This is an error message.~%It has two lines."))

;; Implementation A
(test)
This is an error message.
It has two lines.
```

```
;; Implementation B
(test)
;; Error: This is an error message.
;;        It has two lines.

;; Implementation C
(test)
>> Error: This is an error message.
          It has two lines.
```

9.1.3.1.4 Note about Tabs in Condition Reports

Because the indentation of a *report message* might be shifted to the right or left by an arbitrary amount, special care should be taken with the semi-standard *character* <Tab> (in those *implementations* that support such a *character*). Unless the *implementation* specifically defines its behavior in this context, its use should be avoided.

9.1.3.1.5 Mentioning Containing Function in Condition Reports

The name of the containing function should generally not be mentioned in *report messages*. It is assumed that the *debugger* will make this information accessible in situations where it is necessary and appropriate.

9.1.4 Signaling and Handling Conditions

The operation of the condition system depends on the ordering of active *applicable handlers* from most recent to least recent.

Each *handler* is associated with a *type specifier* that must designate a *subtype* of *type* **condition**. A *handler* is said to be *applicable* to a *condition* if that *condition* is of the *type* designated by the associated *type specifier*.

Active handlers are *established* by using **handler-bind** (or an abstraction based on **handler-bind**, such as **handler-case** or **ignore-errors**).

Active handlers can be *established* within the dynamic scope of other *active handlers*. At any point during program execution, there is a set of *active handlers*. When a *condition* is signaled, the *most recent* active *applicable handler* for that *condition* is selected from this set. Given a *condition*, the order of recentness of active *applicable handlers* is defined by the following two rules:

1. Each handler in a set of active handlers H1 is more recent than every handler in a set H2 if the handlers in H2 were active when the handlers in H1 were established.
2. Let h1 and h2 be two applicable active handlers established by the same *form*. Then h1 is more recent than h2 if h1 was defined to the left of h2 in the *form* that established them.

Once a handler in a handler binding *form* (such as **handler-bind** or **handler-case**) has been selected, all handlers in that *form* become inactive for the remainder of the signaling process. While the selected *handler* runs, no other *handler* established by that *form* is active. That is, if the *handler* declines, no other handler established by that *form* will be considered for possible invocation.

The next figure shows *operators* relating to the *handling* of *conditions*.

```
handler-bind  handler-case  ignore-errors
```

Figure 9-4. Operators relating to handling conditions.

9.1.4.1 Signaling

When a *condition* is signaled, the most recent applicable *active handler* is invoked. Sometimes a handler will decline by simply returning without a transfer of control. In such cases, the next most recent applicable active handler is invoked.

If there are no applicable handlers for a *condition* that has been signaled, or if all applicable handlers decline, the *condition* is unhandled.

The functions **cerror** and **error** invoke the interactive *condition* handler (the debugger) rather than return if the *condition* being signaled, regardless of its *type*, is unhandled. In contrast, **signal** returns **nil** if the *condition* being signaled, regardless of its *type*, is unhandled.

The *variable* ***break-on-signals*** can be used to cause the debugger to be entered before the signaling process begins.

The next figure shows *defined names* relating to the *signaling* of *conditions*.

```
*break-on-signals*  error   warn
cerror              signal
```

Figure 9-5. Defined names relating to signaling conditions.

9.1.4.1.1 Resignaling a Condition

During the *dynamic extent* of the *signaling* process for a particular *condition object*, **signaling** the same *condition object* again is permitted if and only if the *situation* represented in both cases are the same.

For example, a *handler* might legitimately *signal* the *condition object* that is its *argument* in order to allow outer *handlers* first opportunity to *handle* the condition. (Such a *handlers* is sometimes called a "default handler.") This action is permitted because the *situation* which the second *signaling* process is addressing is really the same *situation*.

On the other hand, in an *implementation* that implemented asynchronous keyboard events by interrupting the user process with a call to **signal**, it would not be permissible for two distinct asynchronous keyboard events to *signal identical condition objects* at the same time for different situations.

9.1.4.2 Restarts

The interactive condition handler returns only through non-local transfer of control to specially defined *restarts* that can be set up either by the system or by user code. Transferring control to a restart is called "invoking" the restart. Like handlers, active *restarts* are *established* dynamically, and only active *restarts* can be invoked. An active *restart* can be invoked by the user from the debugger or by a program by using **invoke-restart**.

A *restart* contains a *function* to be *called* when the *restart* is invoked, an optional name that can be used to find or invoke the *restart*, and an optional set of interaction information for the debugger to use to enable the user to manually invoke a *restart*.

The name of a *restart* is used by **invoke-restart**. *Restarts* that can be invoked only within the debugger do not need names.

Restarts can be established by using **restart-bind**, **restart-case**, and **with-simple-restart**. A *restart* function can itself invoke any other *restart* that was active at the time of establishment of the *restart* of which the *function* is part.

The *restarts established* by a **restart-bind** *form*, a **restart-case** *form*, or a **with-simple-restart** *form* have *dynamic extent* which extends for the duration of that *form*'s execution.

Restarts of the same name can be ordered from least recent to most recent according to the following two rules:

1. Each *restart* in a set of active restarts R1 is more recent than every *restart* in a set R2 if the *restarts* in R2 were active when the *restarts* in R1 were established.
2. Let r1 and r2 be two active *restarts* with the same name established by the same *form*. Then r1 is more recent than r2 if r1 was defined to the left of r2 in the *form* that established them.

If a *restart* is invoked but does not transfer control, the values resulting from the *restart* function are returned by the function that invoked the restart, either **invoke-restart** or **invoke-restart-interactively**.

9.1.4.2.1 Interactive Use of Restarts

For interactive handling, two pieces of information are needed from a *restart*: a report function and an interactive function.

The report function is used by a program such as the debugger to present a description of the action the *restart* will take. The report function is specified and established by the `:report-function` keyword to **restart-bind** or the `:report` keyword to **restart-case**.

The interactive function, which can be specified using the `:interactive-function` keyword to **restart-bind** or `:interactive` keyword to **restart-case**, is used when the *restart* is invoked interactively, such as from the debugger, to produce a suitable list of arguments.

invoke-restart invokes the most recently *established restart* whose name is the same as the first argument to **invoke-restart**. If a *restart* is invoked interactively by the debugger and does not transfer control but rather returns values, the precise action of the debugger on those values is *implementation-defined*.

9.1.4.2.2 Interfaces to Restarts

Some *restarts* have functional interfaces, such as **abort**, **continue**, **muffle-warning**, **store-value**, and **use-value**. They are ordinary functions that use **find-restart** and **invoke-restart** internally, that have the same name as the *restarts* they manipulate, and that are provided simply for notational convenience.

The next figure shows *defined names* relating to *restarts*.

```
abort             invoke-restart-interactively  store-value
compute-restarts  muffle-warning                use-value
continue          restart-bind                  with-simple-restart
find-restart      restart-case
invoke-restart    restart-name
```

Figure 9-6. Defined names relating to restarts.

9.1.4.2.3 Restart Tests

Each *restart* has an associated test, which is a function of one argument (a *condition* or **nil**) which returns *true* if the *restart* should be visible in the current *situation*. This test is created by the `:test-function` option to **restart-bind** or the `:test` option to **restart-case**.

9.1.4.2.4 Associating a Restart with a Condition

A *restart* can be "associated with" a *condition* explicitly by **with-condition-restarts**, or implicitly by **restart-case**. Such an assocation has *dynamic extent*.

A single *restart* may be associated with several *conditions* at the same time. A single *condition* may have several associated *restarts* at the same time.

Active restarts associated with a particular *condition* can be detected by *calling* a *function* such as **find-restart**, supplying that *condition* as the *condition argument*. Active restarts can also be detected without regard to any associated *condition* by calling such a function without a *condition argument*, or by supplying a value of **nil** for such an *argument*.

9.1.5 Assertions

Conditional signaling of *conditions* based on such things as key match, form evaluation, and *type* are handled by assertion *operators*. The next figure shows *operators* relating to assertions.

```
assert  check-type  ecase
ccase   ctypecase   etypecase
```

Figure 9-7. Operators relating to assertions.

9.1.6 Notes about the Condition System’s Background

For a background reference to the abstract concepts detailed in this section, see *Exceptional Situations in Lisp*. The details of that paper are not binding on this document, but may be helpful in establishing a conceptual basis for understanding this material.