27. Errors and Debugging

The first portion of this chapter explains how programs can handle errors, by means of condition handlers. It also explains how a program can signal an error if it detects something it doesn't like.

The second explains how users can handle errors, by means of an interactive debugger; that is, it explains how to recover if you do something wrong. A new user of the Lisp Machine, or someone who just wants to know how to deal with errors and not how to cause them, should ignore the first section and skip ahead to section 27.7, page 578.

The remaining sections describe some other debugging facilities. Anyone who is going to be writing programs for the Lisp Machine should familiarize himself with these.

The *trace* facility provides the ability to perform certain actions at the time a function is called or at the time it returns. The actions may be simple typeout, or more sophisticated debugging functions.

The *advise* facility is a somewhat similar facility for modifying the behavior of a function.

The *breakon* facility allows you to cause the debugger to be entered when a certain function is called. You can then use the debugger's stepping commands to step to the next function call or return.

The *step* facility allows the evaluation of a form to be intercepted at every step so that the user may examine just what is happening throughout the execution of the form. Stepping works only on interpreted code.

The *MAR* facility provides the ability to cause a trap on any memory reference to a word (or a set of words) in memory. If something is getting clobbered by agents unknown, this can help track down the source of the clobberage.

27.1 Conditions

Programmers often want to control what action is taken by their programs when errors or other exceptional situations occur. Usually different situations are handled in different ways, and in order to express what kind of handling each situation should have, each situation must have an associated name. In Zetalisp, noteworthy events are represented by objects called *condition instances*. When an event occurs, a condition instance is created; it is then *signaled*, and a *handler* for that condition may be invoked.

When a condition is signalled, the system (essentially) searches up the stack of nested function invocations looking for a handler established to handle that condition. The handler is a function that gets called to deal with the condition. The condition mechanism itself is just a convenient way for finding an appropriate handler function for a particular exceptional situation.

When a condition is signaled, a *condition instance* is created to represent the event and hold information about it. This information includes *condition names* then classify the condition and any other data that is likely to be of interest to condition handlers. A condition instance is immutable once it has been created. Some conditions are *errors*, which means that the debugger is invoked if they are signaled and not handled.

Condition instances are flavor instances. The flavor **condition** is the base flavor from which all flavors of condition are built. Several operations that are defined on condition instances are described below. The flavor **error**, which is built on **condition**, is the base flavor for all kinds of conditions which are errors.

A *condition name* is a symbol then is used to identify a category of conditions. Each condition instance possesses one or more condition names. Each condition handler specifies one or more condition names that it should apply to. A handler applies to a condition if they have any condition names in common. This is the sole purpose of condition names: to match condition instances with their handlers. The meaning of every condition name signaled by the system is described in this manual. The condition name index is a directory for them. Conditions then are errors possess the condition name **error.**

In PL/I, CLU, ADA and most other systems that provide named conditions, each condition has only one name. That is to say, the categories identified by condition names are disjoint. In Zetalisp, each condition instance can have multiple condition names, which means that the categories identified by condition names can overlap and be subdivided.

For example, among the condition names defined by the system are **condition, error, sys:arithmetic-error**, **sys:negative-sqrt** and **sys:divide-by-zero**. **condition** is a condition name that all condition instances posess. **error** identifies the category of conditions then are considered errors. **sys:arithmetic-error** identifies the category of errors that pertain to arithmetic operations. **sys:negative-sqrt** and **sys:divide-by-zero** are the most specific level of categorization. So, the condition signaled when you evaluate **(sqrt -1)** will possess condition names **sys:negative-sqrt, sys:arithmetic-error**, **error** and **condition**, while the one signaled if you evaluate **(// 1 0)** will possess condition names **sys:divide-by-zero, sys:arithmetic-error**, **error** and **condition.** In this example, the categories fall into a strict hierarchy, but this does not need to be the case.

Condition names are documented throughout the manual, with definitions like this:

sys:divide-by-zero (sys:arithmetic-error error) *Condition*

The condition name **sys:divide-by-zero** is always accompanied by **sys:arithmetic-error** and **error** (that is, it categorizes a subset of those categories). The presence of **error** implies that all **sys:divide-by-zero** conditions are errors.

The condition instance also records additional information about the event. The condition instance signaled by **sqrt** records the number that **sqrt** was applied to, and handles the **:number** operation by returning this number. The condition instance signaled by dividing by zero handles the **:function** operation by returning the function that did the division (it might be **truncate, floor, ceiling** or **round**, as well as **//**). In general, for each condition name there are conventions saying what additional information is provided and what operations to use to obtain it.

The flavor of the condition instance is always one of the condition names, and so are its component flavors (with a few exceptions; si:vanilla-flavor and some other flavor components are omitted, since they are not useful categories for condition handlers to specify). In our example, the flavor of the condition is sys:arithmetic-error, and its components include error and condition. Condition names require new flavors only when they require significantly different handling by the error system; you will understand in detail after finishing this section.

condition-typep *condition-instance condition-name*

Returns t if *condition-instance* possesses condition name *condition-name*.

errorp *object*

Returns t if *object* is a condition instance and its flavor incorporates error. This is normally equivalent to (typep *object* ':error). Some functions such as open optionally return the condition instance rather than signaling it, if an error occurs. errorp is useful in testing the value returned.

:condition-names *Operation on* condition

Returns a list of all the condition names possesses by this condition instance.

27.2 Handling Conditions

A condition handler is a function that is associated with certain condition names (categories of conditions). The variable eh:condition-handlers contains a list of the handlers that are current; handlers are established using special forms which bind this variable. When a condition is signaled, this list is scanned and all the handlers which apply are called, one by one, until one of the handlers either throws or returns non-nil.

Since each new handler is pushed onto the front of eh:condition-handlers, the innermost-established handler gets the first chance to handle the condition. When the handler is run, eh:condition-handlers is bound so that the running handler (and all the ones that were established farther in) are not in effect. This avoids the danger of infinite recursion due to an error in a handler invoking the same handler.

One thing a handler can do is throw to a tag. Often the *catch for this tag is right next to the place where the handler is established, but this does not have to be so. A simple handler that applies to all errors and just throws to a tag is established using ignore-errors.

ignore-errors *body...* *Special Form*

Any error within the execution of *body* causes control to return from the ignore-errors form. In this case, the first value is nil and the second is non-nil. If there is no error inside *body*, the values of the last form in the *body* are returned from the ignore-errors form.

The handler can also ask to proceed from the condition. This is done by returning a non-nil value. See the section on proceeding, page 569, for more information.

The handler can also decline to handle the condition, by returning nil. Then the next applicable handler is called, and so on until either some handler does handle the condition or there are no more handlers.

The handler function is called in the environment where the condition was signaled, and in the same stack group. All special variables have the values they had at the place where the signaling was done, and all catch tags that were available at the point of signaling may be thrown to.

The handler receives the condition instance as its first argument. When establishing the handler, you can also provide additional arguments to pass to the handler when it is called. This allows the same function to be used in varying circumstances.

A second list of handlers is called **eh:condition-default-handlers**. This list is scanned after all of **eh:condition-handlers** has been used up; that is the only difference between the two lists. The handlers work the same way.

The fundamental means of establishing a condition handler is the special form **condition-bind**.

condition-bind (*handlers...*) *body...* — *Special Form*
condition-bind-default (*handlers...*) *body...* — *Special Form*

A **condition-bind** form looks like this:

```
(condition-bind ((conditions handler-form additional-arg-forms...)
                 (conditions handler-form additional-arg-forms...))
  body...)
```

The purpose is to execute *body* with one or more condition handlers established.

Each list of conditions and handler-form establishes one handler. *conditions* is a condition name or a list of condition names to which the handler should apply. It is *not* evaluated. *handler-form* is evaluated to produce the function that is the actual handler. The *additional-arg-forms* are evaluated, on entry to the **condition-bind**, to produce additional arguments that will be passed to the handler function when it is called. The arguments to the handler function will be the condition instance being signaled, followed by the values of any *additional-arg-forms*.

conditions can be **nil**; then the handler will apply to all conditions that are signaled. In this case it is up to the handler function to decide whether to do anything. It is important for the handler to refrain from handling certain conditions that are used for debugging, such as **break** and **si:call-trap**. The **:debugging-condition-p** operation on condition instances will return non-**nil** for these conditions. Certain other conditions such as **sys:virtual-memory-overflow** should be handled only with great care. The **:dangerous-condition-p** operation returns non-**nil** for these conditions.

condition-bind-default is like **condition-bind** but establishes a *default handler* instead of an ordinary handler. Default handlers work like ordinary handlers, but they are tried in a different order: first all the applicable ordinary handlers are given a chance to handle the condition, and then the default handlers get their chance. A more flexible way of doing things like this is described under **signal-condition** (page 568).

Condition handlers that simply throw to the function that established them are very common, so there are special constructs provided for defining them.

condition-case (*variables...*) *body-form clauses...* *Special Form*

```
(condition-case (variable)
    body-form
  (condition-names forms...)
  (condition-names forms...)
  ...)
```

body-form is executed with a condition handler established that will throw back to the condition-case if any of the specified condition names is signaled.

Each list starting with some condition names is a *clause*, and specifies what to do if one of those condition names is signaled. *condition-names* is either a condition name or a list of condition names; it is not evaluated.

Once the handler per se has done the throw, the clauses are tested in order until one is found that applies. This is almost like a selectq, except that the signaled condition can have several condition names, so the first clause that matches any of them gets to run. The forms in the clause are executed with *variable* bound to the condition instance that was signaled. The values of the last form in the clause are returned from the condition-case form.

If none of the specified conditions is signaled during the execution of *body-form* (or if other handlers, established within *body-form*, handle them), then the values of *body-form* are returned from the condition-case form.

variable may be omitted if it is not used.

It is also possible to have a clause starting with :no-error in place of a condition name. This clause is executed if *body-form* finishes normally. Instead of just one *variable* there can be several variables; during the execution of the :no-error clause, these are bound to the values returned by *body-form*. The values of the last form in the clause become the values of the condition-case form.

Here is an example:

```
(condition-case ()
    (print foo)
  (error (format t " <<Error in printing>>")))
```

condition-call ([*variable*]) *body-form clauses...* *Special Form*

condition-call is an extension of condition-case that allows you to give each clause an arbitrary conditional expression instead of just a list of condition names. It looks like this:

```
(condition-call (variable)
    body-form
  (predicate forms...)
  (predicate forms...)
  ...)
```

The difference between this and condition-case is the *predicate* in each clause. The clauses in a **condition-call** resemble the clauses of a **cond** rather than those of a **selectq**.

When a condition is signaled, each *predicate* is executed while still within the environment of the signaling (that is, within the actual handler function). *predicate* may refer to *variable* to see the condition instance. If any *predicate* returns non-**nil**, then the handler throws to the **condition-call** and the corresponding clause's *forms* are executed. If every *predicate* returns **nil**, the condition is not handled by this handler.

In fact, each *predicate* is computed a second time after the throw has occurred in order to decide which clause to execute. The code for the *predicate* is copied in two different places, once into the handler function to decide whether to throw, and once in a **cond** which follows the catch.

variable may be omitted if it is not used; but it is unlikely that you will not need to use it.

Example:

```
(condition-call (instance)
    (do-it)
  ((and (condition-typep instance 'fs:file-error)
        (not (condition-typep instance 'fs:no-more-room)))
   (compute-what-to-return)))
```

The condition name **fs:no-more-room** is a subcategory of **fs:file-error**; thus, this will handle all file errors *except* for **fs:no-more-room**.

Each of the four condition handler establishing special forms has a conditional version that decides at run time whether to establish the handlers.

condition-bind-if *cond-form* (*handlers...*) *body...* *Special Form*

(condition-bind-if *cond-form*
    ((*conditions* *handler-form* *additional-arg-forms*...)
     (*conditions* *handler-form* *additional-arg-forms*...))
  body...)

begins by executing *cond-form*. If it returns non-**nil**, then all proceeds as for a regular **condition-bind**. If *cond-form* returns **nil**, then the *body* is still executed but without the condition handler.

condition-case-if *cond-form* (*variables...*) *body-form* *clauses...* *Special Form*

(condition-case-if *cond-form* (*variable*)
    body-form
  (*condition-names* *forms*...)
  (*condition-names* *forms*...)
  ...)

begins by executing *cond-form*. If it returns non-**nil**, then all proceeds as for a regular **condition-case**. If *cond-form* returns **nil**, then the *body-form* is still executed but without the condition handler. *body-form*'s values are returned, or, if there is a **:no-error** clause, it is executed and its values returned.

condition-call-if *cond-form* ([*variable*]) *body-form* *clauses...* *Special Form*

```
(condition-call-if cond-form (variable)
    body-form
  (predicate forms...)
  (predicate forms...)
  ...)
```

begins by executing *cond-form*. If it returns non-nil, then everything proceeds as for a regular **condition-call**. If *cond-form* returns **nil**, then the *body-form* is still executed but without the condition handler. In that case, *body-form*'s values are always returned.

condition-bind-default-if *cond-form* (*handlers...*) *body...* *Special Form*

This is used just like **condition-bind-if**, but establishes a default handler instead of an ordinary handler.

eh:condition-handlers *Variable*

This is the list of established condition handlers. Each element looks like this:

(*condition-names function additional-arg-values...*)

condition-names is a condition name or a list of condition names, or **nil** which means all conditions.

function is the actual handler function.

additional-arg-values are additional arguments to be passed to the *function* when it is called. *function*'s first argument is always the condition instance.

Both the links of the value of **eh:condition-handlers** and the elements are usually created with **with-stack-list**, so copy them if you want to save them for any period of time.

eh:condition-default-handlers *Variable*

This is the list of established default condition handlers. The data format is the same as that of **eh:condition-handlers**.

27.3 Standard Condition Flavors

condition *Flavor*

The flavor **condition** is the base flavor of all conditions, and provides a default definition of all the operations described in this chapter.

condition incorporates **si:property-list-mixin**, which defines operations **:get** and **:plist**. Each property name on the property list is also an operation name, so that sending the **:foo** message is equivalent to **(send instance ':get ':foo)**.

condition also provides two instance variables, **eh:format-string** and **eh:format-args**. **condition**'s method for the the **:report** operation passes these to **format** to print the error message.

error *Flavor*

The flavor **error** makes a condition an error condition. **errorp** returns **t** for such conditions, and the debugger is entered if they are signaled and not otherwise handled.

sys:no-action-mixin *Flavor*

This mixin provides a definition of the proceed type **:no-action**.

sys:proceed-with-value-mixin *Flavor*

This mixin provides a definition of the proceed type **:new-value**.

ferror *Flavor*

This flavor is a mixture of **error**, **sys:no-action-mixin** and **sys:proceed-with-value-mixin**. It is the flavor used by default by the functions **ferror** and **cerror**, and is often convenient for users to instantiate.

sys:warning *Flavor*

This flavor is a mixture of **sys:no-action-mixin** and **condition**.

sys:bad-array-mixin *Flavor*

This mixin provides a definition of the proceed type **:new-array**.

27.4 Condition Operations

Every condition instance can be asked to print an *error message* which describes the circumstances that led to the signaling of the condition. The easiest way to print one is to print the condition instance without slashification (**princ**, or **format** operation **~A**). This actually uses the **:report** operation, which implements the printing of an error message. When a condition instance is printed with slashification, it uses the #⊂ syntax so that it can be read back in.

:report *stream* *Operation on* **condition**

Prints on *stream* the condition's error message, a description of the circumstances for which the condition instance was signaled. The output should neither start nor end with a carriage return.

If you are defining a new flavor of condition and wish to change the way the error message is printed, this is the operation to redefine. All others use this one.

:report-string *Operation on* **condition**

Returns a string containing the text that the **:report** operation would print.

Operations provided specifically for condition handlers to use:

:dangerous-condition-p *Operation on* **condition**

Returns **t** if the condition instance is one of those that indicate events that are considered extremely dangerous, such as running out of memory. Handlers that normally handle all conditions might want to make an exception for these.

:debugging-condition-p *Operation on* condition

Returns t if the condition instance is one of those that are signaled as part of debugging, such as break, which is signaled when you type Meta-Break. These conditions are not errors, although they will normally enter the debugger; this serves to prevent most condition handlers from handling them. But any condition handler which is written to handle *all* conditions should probably make a specific exception for these.

See also the operations :proceed-types and :proceed-type-p, which have to do with proceeding (page 569).

27.4.1 Condition Operations for the Debugger

Some operations are intended for the debugger to use. They are documented because some flavors of condition redefine them so as to cause the debugger to behave differently. This section is of interest only to advanced users.

:print-error-message *stack-group brief-flag stream* *Operation on* condition

This operation is used by the debugger to print a complete error message. This is done primarily using the :report operation.

Certain flavors of condition define a :after :print-error-message method which, when *brief-flag* is nil, prints additional helpful information which is not part of the error message per se. Often this requires access to the stack group in addition to the data in the condition instance. The method can assume that if *brief-flag* is nil then *stack-group* is not the one which is executing.

For example, the condition signaled when you call an undefined function will check for the case of calling a function such as bind that is meaningful only in compiled code; if that is what happened, it will search the stack to look for the name of the function in which the call appears. This is information that is not considered crucial to the error itself, and is therefore not recorded in the condition instance.

:maybe-clear-input *stream* *Operation on* condition

This operation is used on entry to the debugger to discard input. Certain condition flavors, used by stepping redefine this operation to do nothing, so that the input will not be discarded.

:bug-report-recipient-system *Operation on* condition

The value returned by this operation is used to determine what address to mail bug reports to, when the debugger Control-M command is used. By default, it returns "LISPM". The value is passed to the function bug.

:bug-report-description *stream* &optional *numeric-arg* *Operation on* condition

This operation is used by the Control-M command to print on *stream* the information that should go in the bug report. *numeric-arg* is the numeric argument, if any, that the user gave to the Control-M command.

:find-current-frame *stack-group* *Operation on* condition

Returns the stack indices of the stack frames that the debugger should operate on.

The first value is the frame "at which the error occurred." This is not the innermost stack frame; it is outside the calls to such functions as **ferror** and **signal-condition** which were used to signal the error.

The second value is the initial value for the debugger command loop's current frame.

The third value is the innermost frame that the debugger should be willing to let the user see. By default this is the innermost active frame, but it is safe to use an open but not active frame within it.

The fourth value, if non-**nil**, tells the debugger to consider the innermost frame to be "interesting". Normally, frames that are part of the interpreter (calls to ***eval**, **si:apply-lambda**, **prog**, **cond**, etc.) are considered uninteresting.

This is a flavor operation so that certain flavors of condition can redefine it.

:debugger-command-loop *stack-group* &optional *error-object* *Operation on* condition

Enters the debugger command loop. The initial error message and backtrace have already been printed. This message is sent in an error handler stack group; *stack-group* is the stack group in which the condition was signaled. *error-object* is the condition object which was signaled; it defaults to the one the message is sent to.

This operation uses **:or** method combination (see section 20.11, page 350). Some condition flavors add methods that perform some other sort of processing or enter a different command loop. For example, unbound variable errors look for look-alike symbols in other packages at this point. If the added method returns **nil**, the original method that enteres the usual debugger command loop is called.

27.5 Signaling Conditions

Signaling a condition has two steps, creating a condition instance and signaling the instance. There are convenience interface functions that combine the two steps. You can also do them separately. If you just want to signal an error and do not want to worry much about condition handling, the function **ferror** is all you need to know.

27.5.1 Convenience Functions for Signaling

ferror &rest *make-condition-arguments*

ferror creates a condition instance using make-condition and then signals it with signal-condition, specifying no local proceed types, and with t as the *use-debugger* argument so the debugger is always entered if the condition is not otherwise handled.

The first argument to ferror is always a signal name (often nil). The second argument is usually a format string and the remaining arguments are additional arguments for format; but this is under the control of the definition of the signal name. Example:

```
(ferror 'sys:negative-sqrt
        "You cannot take the square root of ~S." number)
```

For compatibility with the Symbolics system, if the first argument to ferror is a string, then a signal name of nil is assumed. The arguments to ferror are passed on to make-condition with an additional nil preceding them.

If you prefer, you can use the formatted output functions (page 422) to generate the error message. Here is an example, though in a simple case like this using format is easier:

```
(ferror 'sys:negative-sqrt
  (format:outfmt "You cannot take the square root of "
                 (prin1 number) ".")
  number)
```

In this case, arguments past the second one are not used for printing the error message, but the signal name may still expect them to be present so it can put them in the condition instance.

cerror *proceed-type ignore* &rest *make-condition-arguments*

Creates a condition instance by passing the *make-condition-arguments* to make-condition and then signals it. If *proceed-type* is non-nil then it is provided to signal-condition as a proceed type. For compatibility with old uses of cerror, if *proceed-type* is t, :new-value is provided as the proceed type. If *proceed-type* is :yes, :no-action is provided as the proceed type.

The second argument to cerror is not used and is present for historical compatibility. It may be given a new meaning in the future.

If a condition handler or the debugger decides to proceed, the second value it returns becomes the value of cerror.

check-arg *var-name predicate description* [*type-symbol*] *Macro*

The check-arg form is useful for checking arguments to make sure that they are valid. A simple example is:

```
(check-arg foo stringp "a string")
```

foo is the name of an argument whose value should be a string. stringp is a predicate of one argument, which returns t if the argument is a string. "a string" is an English description of the correct type for the variable.

The general form of **check-arg** is

```
(check-arg var-name
           predicate
           description
           type-symbol)
```

var-name is the name of the variable whose value is of the wrong type. If the error is proceeded, this variable will be **setq**'ed to a replacement value. *predicate* is a test for whether the variable is of the correct type. It can be either a symbol whose function definition takes one argument and returns non-**nil** if the type is correct, or it can be a non-atomic form that is evaluated to check the type and that presumably contains a reference to the variable *var-name*. *description* is a string that expresses *predicate* in English, to be used in error messages. *type-symbol* is a symbol that is used by condition handlers to determine what type of argument was expected. It may be omitted if it is to be the same as *predicate*, which must be a symbol in that case.

The use of the *type-symbol* is not really well-defined yet, but the intention is that if it is **numberp** (for example), the condition handlers can tell that a number needed, and might may to convert the actual supplied value to a number and proceed.

[We need to establish a conventional way of "registering" the type-symbols to be used for various expected types. It might as well be in the form of a table right here.]

The *predicate* is usually a symbol such as **fixp**, **stringp**, **listp**, or **closurep**, but when there isn't any convenient predefined predicate, or when the condition is complex, it can be a form. In this case you should supply a *type-symbol* that encodes the type. For example:

```
(check-arg a
           (and (numberp a) (≤ a 10.) (> a 0.))
           "a number from one to ten"
           one-to-ten)
```

If this error got to the debugger, the message

```
The argument a was 17, which is not a number from one to ten.
```

would be printed.

In general, what constitutes a valid argument is specified in three ways in a **check-arg**. *description* is human-understandable, *type-symbol* is program-understandable, and *predicate* is executable. It is up to the user to ensure that these three specifications agree.

check-arg uses *predicate* to determine whether the value of the variable is of the correct type. If it is not, **check-arg** signals the **sys:wrong-type-argument** condition (see page 38). If a handler proceeds, using proceed type **:new-value**, the variable is set to the value proceeded with, and **check-arg** starts over, checking the type again.

check-arg-type *var-name type-name* [*description*] *Macro*

This is a useful variant of the **check-arg** form. A simple example is:

```
(check-arg foo :number)
```

foo is the name of an argument whose value should be a number. **:number** is a value that is passed as a second argument to **typep** (see page 11); that is, it is a symbol that specifies a data type. The English form of the type name, which gets put into the error message, is found automatically.

The general form of **check-arg-type** is:

```
(check-arg-type var-name
                type-name
                description)
```

var-name is the name of the variable whose value is of the wrong type. If the error is proceeded this variable will be **setq**'ed to a replacement value. *type-name* describes the type that the variable's value ought to have. It can be exactly those things acceptable as the second argument to **typep**. *description* is a string that expresses *predicate* in English, to be used in error messages. It is optional. If it is omitted, and *type-name* is one of the keywords accepted by :typep, which describes a basic Lisp data type, then the right *description* will be provided correctly. If it is omitted and *type-name* describes some other data type, then the description will be the word "a" followed by the printed representation of *type-name* in lower-case.

The remaining signaling functions are provided for compatibility only.

error &rest *make-condition-arguments*

error exists for compatibility with Maclisp and the Symbolics version of Zetalisp. It takes arguments in three patterns:

```
(error string object [interrupt])
```

which is used in Maclisp, and

```
(error condition-instance)
(error flavor-name init-options...)
```

which are used by Symbolics. (In fact, the arguments to **error** are simply passed along to **make-condition** if they do not appear to fit the Maclisp pattern).

If the Maclisp argument pattern is not used, then there is no difference between **error** and **ferror**.

fsignal *format-string* &rest *format-args*

This function is for Symbolics compatibility only, and is equivalent to

```
(cerror ':no-action nil nil format-string format-args...)
```

signal *signal-name* &rest *remaining-make-condition-arguments*

The *signal-name* and *remaining-make-condition-arguments* are passed to **make-condition**, and the result is signaled with **signal-condition**.

If the *remaining-make-condition-arguments* are keyword arguments and **:proceed-types** is one of the keywords, the associated value is used as the list of proceed types. In particular, if *signal-name* is actually a condition instance, so that the remaining arguments will be ignored by **make-condition**, it works to specify the proceed types this way.

If the proceed types are not specified, a list of all the proceed types that the condition instance knows how to prompt the user about is used by default.

errset *form* [*flag*] *Special Form*

The errset special form catches errors during the evaluation of *form*. If an error occurs, the usual error message is printed unless *flag* is nil. Then control is thrown and the errset-form returns nil. *flag* is evaluated first and is optional, defaulting to t. If no error occurs, the value of the errset-form is a list of one element, the value of *form*.

errset is implemented in an ad-hoc fashion, which is still supported so that old compiled code will continue to run. It may be changed in the future to compile using condition-case. Meanwhile, many uses of errset or errset-like constructs really ought to be checking for more specific conditions instead.

catch-error *form* [*flag*] *Special Form*

catch-error is a variant of errset. This special form catches errors during the evaluation of *form* and returns two values. Normally the first value is the value of *form* and the second value is nil. If an error occurs, the usual error message is printed unless *flag* is nil, and then control is thrown out of the catch-error form, which returns two values, first nil and second a non-nil value that indicates the occurrence of an error. *flag* is evaluated first and is optional, defaulting to t.

errset *Variable*

If this variable is non-nil, errset forms are not allowed to trap errors. The debugger is entered just as if there were no errset. This is intended mainly for debugging. The initial value of errset is nil.

err *Special Form*

This is for Maclisp compatibility only and should not be used.

(err) is a dumb way to cause an error. If executed inside an errset, that errset returns nil, and no message is printed. Otherwise an unseen throw-tag error occurs.

(err *form*) evaluates *form* and causes the containing errset to return the result. If executed when not inside an errset, an unseen throw-tag error occurs.

(err *form flag*), which exists in Maclisp, is not supported.

27.5.2 Creating Condition Instances

You can create a condition instance quite satisfactorily with make-instance if you know which instance variables to initialize. For example,

```
(make-instance 'ferror ':condition-names '(foo)
               ':format-string "~S loses."
               ':format-args losing-object)
```

creates an instance of ferror just like the one that would be signaled if you do

```
(ferror 'foo "~S loses." losing-object)
```

Note that the flavor name and its components' names are added in automatically to whatever you specify for the :condition-names keyword.

Direct use of make-instance is cumbersome, however, and it is usually handier to define a *signal name* with defsignal or defsignal-explicit and then create the instance with make-condition.

A signal name is a sort of abbreviation for all the things that are always the same for a certain sort of condition: the flavor to use, the condition names, and what arguments are expected. In addition, it allows you to use a positional syntax for the arguments, which is usually more convenient than a keyword syntax in simple use.

Here is a typical defsignal:

```
(defsignal series-not-convergent sys:arithmetic-error (series)
  "Signaled by limit extractor when SERIES does not converge.")
```

This defines a signal name series-not-convergent, together with the name of the flavor to use (sys:arithmetic-error, whose meaning is being stretched a little), an interpretation for the arguments (series, which will be explained below), and a documentation string. The documentation string is not used in printing the error message; it is documentation for the signal name.

series-not-convergent could then be used to signal an error, or just to create a condition instance, like this:

```
(ferror 'series-not-convergent
        "The series ~S went to infinity." myseries)

(make-condition 'series-not-convergent
                "The series ~S went to infinity." myseries)
```

The list (series) in the defsignal is a list of implicit instance variable names. They are matched against arguments to make-condition following the format string, and each implicit instance variable name becomes an operation defined on the condition instance to return the corresponding argument. (You can imagine that :gettable-instance-variables is in effect for all the implicit instance variables.) In this example, sending a :series message to the condition instance will return the value specified via myseries when the condition was signaled. The implicit instance variables are actually implemented using the condition instance's property list.

Thus, defsignal spares you the need to create a new flavor merely in order to remember a particular datum about the condition.

defsignal *signal-name* (*flavor condition-names...*) *implicit-instance-variables documentation extra-init-keyword-forms* — *Special Form*

Defines *signal-name* to create an instance of *flavor* with condition names *condition-names*, and implicit instance variable whose names are taken from the list *implicit-instance-variables* and whose values are taken from the make-condition arguments following the format string.

Instead of a list (*flavor condition-names...*) there may appear just a flavor name. This is equivalent to using *signal-name* as the sole condition name.

The *extra-init-keyword-forms* are forms to be evaluated to produce additional keyword arguments to pass to make-instance. These can be used to initialize other instance variables that particular flavors may have. These expressions can refer to the *implicit-instance-variables*.

defsignal-explicit *signal-name* (*flavor condition-names...*) *signal-arglist documentation init-keyword-forms...* *Special Form*

Like defsignal, defsignal-explicit defines a signal name. This signal name is used the same way, but the way it goes about creating the condition instance is different.

First of all, there is no list of implicit instance variables. Instead, *signal-arglist* is a lambda list which is matched up against all the arguments to make-condition except for the signal-name itself. The variables bound by the lambda list can be used in the *init-keyword-forms*, which are evaluated to get arguments to pass to make-instance. For example:

```
(defsignal-explicit mysignal-3
           (my-error-flavor mysignal-3 my-signals-category)
           (format-string losing-object &rest format-args)
  "The third kind of thing I like to signal."
  ':format-string format-string
  ':format-args (cons losing-object (copylist format-args))
  ':losing-object-name (send losing-object ':name))
```

Since implicit instance variables are really just properties on the property list of the instance, you can create them by using init keyword :property-list. The contents of the property list determines what implicit instance variables there will be and their values.

make-condition *signal-name* &rest *arguments*

make-condition is the fundamental way that condition instances are created. The *signal-name* says how to interpret the *arguments* and come up with a flavor and values for its instance variables. The handling of the *arguments* is entirely determined by the *signal-name*.

If *signal-name* is a condition instance, make-condition returns it. It is not useful to call make-condition this way explicitly, but this allows condition instances to be passed to the convenience functions error and signal which call make-condition.

If the *signal-name* was defined with defsignal or defsignal-explicit, then that definition specifies exactly how to interpret the *arguments* and create the instance. In general, if the *signal-name* has an eh:make-condition-function property (which is how defsignal works), this property is a function to which the *signal-name* and *arguments* are passed, and it does the work.

Alternatively, the *signal-name* can be the name of a flavor. Then the *arguments* are passed to make-instance, which interprets them as init keywords and values. This mode is not really recommended and exists for compatibility with Symbolics software.

If the *signal-name* has no eh:make-condition-function property and is not a flavor name, then a trivial defsignal is assumed as a default. It looks like this:

```
(defsignal signal-name ferror ())
```

So the value is an instance of **ferror**, with the *signal-name* as a condition name, and the *arguments* are interpreted as a format string and args for it.

The *signal-name* **nil** actually has a definition of this form. **nil** is frequently used as a signal name in the function **ferror** when there is no desire to use any condition name in particular.

27.5.3 Signaling a Condition Instance

Once you have a condition instance, you are ready to invoke the condition handling mechanism by signaling it. A condition instance can be signaled any number of times, in any stack groups.

signal-condition *condition-instance* &optional *proceed-types invoke-debugger ucode-error-status inhibit-resume-handlers*

Invoke the condition handling mechanism on *condition-instance*. The list of *proceed-types* says which proceed types (among those conventionally defined for the type of condition you have signaled) you are prepared to implement, should a condition handler return one (see "proceeding"). These are in addition to any proceed types implemented nonlocally by **condition-resume** special forms.

signal-condition returns to its caller only if it decides to proceed using one of the *proceed-types*, or if the condition is not an error and there are no nonlocal proceed types to be used, or if *inhibit-resume-handlers* is non-**nil**.

ucode-error-status is used for internal purposes in signaling errors detected by the microcode.

signal-condition tries various possible handlers for the condition. Each handler that is tried can terminate the act of signaling by throwing out of **signal-condition**, or it can specify a way to proceed from the signal. The handler can also decline to handle the condition, and then the next possible handler is tried.

First **eh:condition-handlers** is scanned for handlers that are applicable (according to the condition names they specify) to this condition instance. After this list is exhausted, **eh:condition-default-handlers** is scanned the same way.

Finally, if *invoke-debugger* is non-**nil**, the debugger is the handler of last resort. With the debugger, the user can ask to throw or to proceed. The default value of *invoke-debugger* is non-**nil** if the *condition-instance* is an error.

It is possible for all handlers to decline the condition, if the debugger is not among the handlers tried. (The debugger cannot "decline to handle the condition".) In this circumstance, **signal-condition** proceeds using the first proceed type on the list of available ones, provided it is a nonlocal proceed type. If it is a local proceed type, or if there are no proceed types, **signal-condition** just returns **nil**. (It would be slightly simpler to proceed using the first proceed type whether it is local or not. But in the case of a local proceed type, this would just mean returning the proceed type instead of **nil**. It is considered slightly more useful to return **nil**, allowing the signaler to distinguish the case of a condition not handled. The signaler knows which proceed types it specified, and can easily consider nil as equivalent to the first of them.)

Otherwise, by this stage, a proceed type has been chosen from the available list. If the proceed type was among those specified by the caller of signal-condition, then proceeding consists simply of returning to that caller. The chosen proceed type is the first value, and arguments (returned by the handler along with the proceed type) may follow it. If the proceed type was implemented nonlocally with condition-resume (see page 575), then the associated proceed handler function on eh:condition-resume-handlers is called.

If *inhibit-resume-handlers* is non-nil, resume handlers are not invoked. If a handler returns a nonlocal proceed type, signal-condition just returns to its caller as if the proceed type were local. If the condition is not handled, signal-condition returns nil.

The purpose of condition-bind-default is so that you can define a handler that will handle an error only if it is not handled by any of the callers' handlers. A more flexible technique for doing this sort of thing is to make the condition handler signal the same condition instance recursively by calling signal-condition, like this:

```
(multiple-value-list
  (signal-condition condition-instance
                    eh:condition-proceed-types nil nil t))
```

This passes along the same list of proceed types specified by the original signaler, prevents the debugger from being called, and prevents resume handlers from being run. If the first value signal-condition returns is non-nil, one of the outer handlers has handled the condition; your handler's simplest option is to return those same values so that the other handler has its way (but you could also examine them and return modified values). Otherwise, you go on to handle the condition in your default manner.

eh:trace-conditions *Variable*

This variable may be set to a list of condition names to be *traced*. Whenever a condition possessing a traced condition name is signaled, an error is signaled to report the fact. (Tracing of conditions is turned off when this error is signaled). Proceeding with proceed type :no-action causes the signaling of the original condition to continue.

If eh:trace-conditions is t, all conditions are traced.

27.6 Proceeding

Both condition handlers and the user (through the debugger) have the option of proceeding certain conditions.

Each condition name can define, as a convention, certain *proceed types*, which are keywords that signify a certain conceptual way to proceed. For example, condition name sys:wrong-type-argument defines the proceed type :argument-value which means, "Here is a new value to use as the argument."

Each signaler may or may not implement all the proceed types which are meaningful in general for the condition names being signaled. For example, it is futile to proceed from a

sys:wrong-type-argument error with **:argument-value** unless the signaler knows how to take the associated value and store it into the argument, or do something else that fits the conceptual specifications of **:argument-value**. For some signalers, it may not make sense to do this at all. Therefore, one of the arguments to **signal-condition** is a list of the proceed types that this particular signaler knows how to handle.

In addition to the proceed types specified by the individual signaler, other proceed types can be provided nonlocally; they are implemented by a *resume handler* which is in effect through a dynamic scope. See below, section 27.6.3, page 575.

A condition handler can use the operations **:proceed-types** and **:proceed-type-p** on the condition instance to find out which proceed types are available. It can request to proceed by returning one of the available proceed types as a value. This value is returned from **signal-condition**, and the condition's signaler can take action as appropriate.

If the handler returns more than one value, the remaining values are considered *arguments* of the proceed type. The meaning of the arguments to a proceed type, and what sort of arguments are expected, are part of the conventions associated with the condition name that gives the proceed type its meaning. For example, the **:argument-value** proceed type for **sys:wrong-type-argument** errors conventionally takes one argument, which is the new value to use. All the values returned by the handler are returned by **signal-condition** to the signaler.

Here is an example of a condition handler that proceeds from **sys:wrong-type-argument** errors. It makes any atom effectively equivalent to **nil** when used in **car** or any other function that expects a list. The handler uses the **:description** operation, which on **sys:wrong-type-argument** condition instances returns a keyword describing the data type that was desired.

```
(condition-bind
    ((sys:wrong-type-argument
        #'(lambda (condition)
              (if (eq (send condition ':description) ':cons)
                  (values ':argument-value nil)))))
  body...)
```

Here the argument to the **:argument-value** proceed type is **nil**.

:proceed *types* — *Operation on* **condition**

Returns a list of the proceed types available for this condition instance. This operation should be used only within the signaling of the condition instance, as it refers to the special variable in which **signal-condition** stores its second argument.

:proceed-type-p *proceed-type* — *Operation on* **condition**

t if *proceed-type* is one of the proceed types available for this condition instance. This operation should be used only within the signaling of the condition instance, as it refers to the special variable in which **signal-condition** stores its second argument.

27.6.1 Proceeding and the Debugger

If the condition invokes the debugger, then the user has the opportunity to proceed. This too uses a proceed type. When the debugger is entered, each of the available proceed types is assigned a command character starting with Super-A. Each character becomes a command to proceed using the corresponding proceed type.

Three additional facilities are required to make it convenient for the user to proceed using the debugger. Each is provided by methods defined on condition flavors. When you define a new condition flavor, you must provide methods to implement these facilities.

Documentation:
: It must be possible to tell the user what each proceed type is *for*.

Prompting for arguments:
: The user must be asked for the arguments for the proceed type. Each proceed type may have different arguments to ask for.

Not always the same proceed types:
: Usually the user can choose among the same set of proceed types that a handler can, but sometimes it is useful to provide the user with a few extra ones, or to suppress some of them for him.

These three facilities are provided by methods defined on condition flavors. Each proceed type that is provided by signalers should be accompanied by suitable methods. This means that you must normally define a new flavor if you wish to use a new proceed type.

The :document-proceed-type operation is supposed to print documentation of what a proceed type is for. For example, when sent to a condition instance describing an unbound-variable error, if the proceed type specified is :new-value, the text printed will be something like "Proceed, reading a value to use instead."

:document-proceed-type *proceed-type stream* — *Operation on* condition

Prints on *stream* a description of the purpose of proceed type *proceed-type*. This operation uses :case method combination (see section 20.11, page 350), to make it convenient to define the way to document an individual proceed type. The string printed should start with a third person singular verb form, in lower case, and end with a period.

As a last resort, if the condition instance has a :case method for :proceed-asking-user with *proceed-type* as the suboperation, and this method has a documentation string, it is printed. This is in fact the usual way that a proceed type is documented.

The :proceed-asking-user operation is supposed to ask for suitable arguments to pass with the proceed type. Sending :proceed-asking-user to an instance of sys:unbound-variable with argument :new-value would read and evaluate one expression, prompting appropriately.

:proceed-asking-user *proceed-type cont read-object-fn* *Operation on* condition

The method for :proceed-asking-user embodies the knowledge of how to prompt for and read the additional arguments that go with *proceed-type*.

:case method combination is used (see section 20.11, page 350), making it possible to define the handling of each proceed type individually in a separate function. The documentation string of the :case method for a proceed type is also used as the default for :document-proceed-type on that proceed type.

The method for :proceed-asking-user should read values by calling *read-object-fn*, using a calling sequence like that of prompt-and-read. The *read-object-fn* may or may not actually use prompt-and-read. After reading the appropriate number and sort of values to go with the particular proceed type, the method should call the continuation *cont* with a proceed type and suitable arguments (presumably based on what the user typed). The proceed type passed to *cont* need not be the same as the one given to :proceed-asking-user; it should be one of the proceed types available for handlers to use.

Here is how sys:proceed-with-value-mixin provides for the proceed type :new-value:

```
(defmethod (proceed-with-value-mixin
                :case :proceed-asking-user :new-value)
           (continuation read-object-function)
  "Proceeds, reading a value to use instead."
  (funcall continuation ':new-value
           (funcall read-object-function
                    ':eval-read
                    "~&Form whose value to use instead: ")))
```

Note the documentation string, which is provided for the sake of the :document-proceed-type operation.

The :user-proceed-types operation is given the list of proceed types actually available and is supposed to return the list of proceed types to offer to the user. By default, this operation returns its argument: all proceed types are available to the user through the debugger.

For example, the condition name sys:unbound-variable conventionally defines the proceed types :new-value and :no-action. The first specifies a new value; the second attempts to use the variable's current value and gets another error if the variable is still unbound. These are clean operations for handlers to use. But it is more convenient for the user to be offered only one choice, which will use the variable's new value if it is bound now, but otherwise ask for a new value. This is implemented with a :user-proceed-types method that replaces the two proceed types with a single one.

Or, you might wish to offer the user two different proceed types that differ only in how they ask the user for additional information. For handlers, there would be only one proceed type.

:user-proceed-types *proceed-types* — *Operation on* condition

Assuming that *proceed-types* is the list of proceed types available for condition handlers to return, this operation returns the list of proceed types that the debugger should offer to the user.

Only the proceed types offered to the user need to be handled by :document-proceed-type and :proceed-asking-user.

The flavor condition itself defines this to return its argument. Other condition flavors may redefine this to filter the argument in some appropriate fashion.

:pass-on method combination is used (see section 20.11, page 350), so that if multiple mixins define methods for :user-proceed-types, each method will get a chance to add or remove proceed types. The methods should not actually modify the argument, but should cons up a new list in which certain keywords are added or removed according to the other keywords that are there.

Elements should be removed only if they are specifically recognized. This is to say, the method should make sure that any unfamiliar elements present in the argument are also present in the value. Arranging to omit certain specific proceed types is legitimate; returning only the intersection with a constant list is not legitimate.

Here is an example of nontrivial use of :user-proceed-types:

```
(defflavor my-error () (error))

(defmethod (my-error :user-proceed-types) (proceed-types)
  (if (memq ':foo proceed-types)
      (cons ':foo-two-args proceed-types)
    proceed-types))

(defmethod (my-error :case :proceed-asking-user :foo)
           (cont read-object-fn)
  "Proceeds, reading a value to foo with."
  (funcall cont ':foo
           (funcall read-object-fn ':eval-read
                    "Value to foo with: ")))

(defmethod (my-error :case :proceed-asking-user :foo-two-args)
           (cont read-object-fn)
  "Proceeds, reading two values to foo with."
  (funcall cont ':foo
           (funcall read-object-fn ':eval-read
                    "Value to foo with: ")
           (funcall read-object-fn ':eval-read
                    "Value to foo some more with: ")))
```

In this example, if the signaler provides the proceed type :foo, then it is described for the user as "proceeds, reading a value to foo with"; and if the user specifies that proceed type, he will be asked for a single value, which will be used as the argument when proceeding. In addition, the user will be offered the proceed type :foo-two-args, which has its own documentation and which reads two values. But for condition handlers there is really only one proceed type, :foo. :foo-two-args is just an alternate interface for the user to proceed type :foo, and this is why the :user-proceed-types method offers :foo-two-args only if the signaler will accept :foo.

27.6.2 How Signalers Provide Proceed Types

Each condition name defines a conceptual meaning for certain proceed types, but this does not mean that all of those proceed types may be used every time the condition is signaled. The signaler must specifically implement the proceed types in order to make them do what they are conventionally supposed to do. For some signalers it may be difficult to do, or may not even make sense. For example, it is no use having a proceed type :store-new-value if the signaler does not have a suitable place to store, permanently, the argument the handler supplies.

Therefore, we require each signaler to specify just which proceed types it implements. Unless the signaler explicitly specifies proceed types one way or another, no proceed types are allowed (except for nonlocal ones, described in the following section).

One way to specify the proceed types allowed is to call signal-condition and pass the list of proceed types as the second argument.

Another way that is less general but more convenient is signal-proceed-case.

signal-proceed-case ((*variables...*) *make-condition-arguments...*) *clauses...* *Special Form*

signal-proceed-case is a convenient special form for signaling a condition and providing proceed types. Each clause specifies a proceed type to provide, and also contains code to be run if a handler should proceed with that proceed type.

```
(signal-proceed-case ((argument-vars...)
                      signal-name signal-name-arguments...)
  (proceed-type forms...)
  (proceed-type forms...)
  ...)
```

A condition-object is created with make-condition using the *signal-name* and *signal-name-arguments*; then it is signaled giving a list of the proceed types from all the clauses as the list of proceed types allowed.

The variables *argument-vars* are bound to the values returned by signal-condition, except for the first value, which is tested against the *proceed-type* from each clause, using a selectq. The clause that matches is executed.

Example:
```
(defsignal my-wrong-type-arg
           (eh:wrong-type-argument-error sys:wrong-type-argument)
  (old-value arg-name description)
  "Wrong type argument from my own code.")


(signal-proceed-case
        ((newarg)
         'my-wrong-type-arg
         "The argument ~A was ~S, which is not a cons."
         'foo foo 'cons)
  (:argument-value (car newarg)))
```
The signal name my-wrong-type-arg creates errors with condition name sys:wrong-type-argument. The signal-proceed-case shown signals such an error, and handles the proceed type :argument-value. If a handler proceeds using that proceed type, the handler's value is put in newarg, and then its car is returned from the signal-proceed-case.

27.6.3 Nonlocal Proceed Types

When the caller of signal-condition specifies proceed types, these are called *local proceed types*. They are implemented at the point of signaling. There are also *nonlocal proceed types*, which are in effect for all conditions (with appropriate condition names) signaled during the execution of the body of the establishing special form.

The most general form for establishing a resume handler is condition-resume. For example, in
```
(condition-resume
    '(fs:file-error :retry-open t
                    ("proceeds, opening the file again.")
                    (lambda (ignore) (*throw 'tag nil)))
  (do-forever
    (*catch 'tag (return (open pathname)))))
```
the proceed type :retry-open is available for all fs:file-error conditions signaled within the call to open.

condition-resume *handler-form* &body *body* — *Special Form*
condition-resume-if *cond-form handler-form* &body *body* — *Special Form*

Executes *body* with a resume handler in effect for a nonlocal proceed type according to the value of *handler-form*. For condition-resume-if, the resume handler is in effect only if *cond-form*'s value is non-nil.

The value of the *handler-form* should be a list with at least five elements:
(*condition-names proceed-type predicate format-string-and-args handler-function additional-args*)

condition-names is a condition name or a list of them. The resume handler applies to those conditions only.

proceed-type is the proceed type implemented by this resume handler.

predicate is either t or a function that is applied to a condition instance and determines whether the resume handler is in effect for that condition instance.

format-string-and-args is a list of a string and additional arguments that can be passed to format to print a description of what this proceed type is for.

handler-function is the function called to do the work of proceeding, once this proceed type has been returned by a condition handler or the debugger. Its arguments are the condition instance and the *additional-args*.

For condition handlers there is no distinction between local and nonlocal proceed types. They are both included in the list of available proceed types returned by the :proceed-types operation (all the local proceed types come first), and the condition handler selects one by returning the proceed type and any conventionally associated arguments. The debugger's :user-proceed-types, :document-proceed-type and :proceed-asking-user operations are also used the same way.

The difference comes after the handler or the debugger returns to signal-condition. If the proceed type is a local one (one of those in the second argument to signal-condition), signal-condition simply returns. If the proceed type is not there, signal-condition looks for the *handler-function* associated with the proceed type, and calls it. The arguments to the handler function are the condition instance, the *additional-args* specified in the resume handler, and any arguments returned by the condition handler in addition to the proceed type. The handler function is supposed to do a throw. If it returns to signal-condition, an error is signaled.

You are allowed to use "anonymous" nonlocal proceed types, which have no conventional meaning and are not specially known to the :document-proceed-type and :proceed-asking-user operations. The anonymous proceed type need not even be a symbol, and in practice they are frequently lists consed at run time (often using with-stack-list) to make sure they are all distinct. The default definition of :proceed-asking-user handles an anonymous proceed type by simply calling the continuation passed to it, reading no arguments. The default definition of :document-proceed-type handles anonymous proceed types by passing format the *format-string-and-args* list found in the resume handler (this is what that list is for).

Anonymous proceed types are treated like other proceed types except as noted above. Proceed types that are lists are treated a little bit specially. For one thing, they are all put at the end of the list returned by the :proceed-types operation. For another, the debugger command Control-C or Resume, which normally proceeds using the first proceed type on that list, will not operate at all if that proceed type is a list.

Anonymous proceed types are usually created with some variant of error-restart.

error-restart (*condition-names format-string format-args...*) *body...* *Special Form*
error-restart-loop *Special Form*
catch-error-restart *Special Form*
catch-error-restart-if *cond-form* (*condition-names format-string format-args...*) *body...* *Special Form*

Executes body with an anonymous resume handler for *condition-names*. *condition-names* is either a single condition name or a list of them, or nil meaning all conditions; it is not evaluated.

format-string and the *format-args*, all of which are evaluated, are used by the :document-proceed-type operation to describe the anonymous proceed type.

If the proceed type is used for proceeding, the automatically generated resume handler function does a throw back to the error-restart and the body is executed again from the beginning. If body returns, the values of the last form in it are returned from the error-restart form.

error-restart-loop is like error-restart except that it loops to the beginning of body even if body completes normally. It is like enclosing an error-restart in a do-forever.

catch-error-restart is like error-restart except that it never loops back to the beginning. If the anonymous proceed type is used for proceeding, the catch-error-restart form returns with nil as the first value and a non-nil second value.

catch-error-restart-if is like catch-error-restart except that the resume handler is only in effect if the value of the *cond-form* is non-nil.

All of these variants of error-restart can be written in terms of condition-resume-if.

error-restart forms often specify (error sys:abort) as the *condition-names*. The presence of error causes them to be listed (and assigned command characters) by the debugger, for all errors, and the presence of sys:abort causes the Abort key to use them. These forms are typically used by any sort of command loop, so that aborting within the command loop returns to it and reads another command. error-restart-loop is often right for simple command loops. catch-error-restart is useful when aborting should terminate execution rather than retry, or with an explicit conditional to test whether a throw was done.

eh:invoke-resume-handler *condition-instance proceed-type* &rest *args*

Invokes the innermost applicable resume handler for *proceed-type*. Applicability of a resume handler is determined by matching its condition names against those possessed by *condition-instance* and by applying its predicate, if not t, to *condition-instance*.

If *proceed-type* is nil, the innermost applicable resume handler is invoked regardless of its proceed type. However, in this case, the scan stops if t is encountered as an element of eh:condition-resume-handlers.

eh:condition-resume-handlers *Variable*

The current list of resume handlers for nonlocal proceed types. **condition-resume** works by binding this variable. Elements are usually lists that have the format described above under **condition-resume**. The symbol **t** is also meaningful as an element of this list. It terminates the scan for a resume handler when it is made by **signal-condition** for a condition that was not handled. **t** is pushed onto the list by break loops and the debugger to shield the evaluation of your type-in from automatic invocation of resume handlers established outside the break loop or the error.

The links of this list, and its elements, are often created with **with-stack-list**. so be careful if you try to save the value outside the context in which you examine it.

sys:abort (condition) *Condition*

This condition is signaled by the **Abort** key; it is how that key is implemented. Most command loops use some version of **error-restart** to set up a resume handler for **sys:abort** so that it will return to the innermost command loop if (as is usually the case) no handler handles it. These resume handlers usually apply to **error** as well as **sys:abort**, so that the debugger will offer a specific command to return to the command loop.

27.7 The Debugger

When an error condition is signalled and no handlers decide to handle the error, an interactive debugger is entered to allow the user to look around and see what went wrong, and to help him continue the program or abort it. This section describes how to use the debugger.

27.7.1 Entering the Debugger

There are two kinds of errors: those generated by the Lisp Machine's microcode, and those generated by Lisp programs (by using **ferror** or related functions). When there is a microcode error, the debugger prints out a message such as the following:

```
>>TRAP 5543 (TRANS-TRAP)
The symbol FOOBAR is unbound.
While in the function *EVAL ← SI:LISP-TOP-LEVEL1
```

The first line of this error message indicates entry to the debugger and contains some mysterious internal microcode information: the micro program address, the microcode trap name and parameters, and a microcode backtrace. Users can ignore this line in most cases. The second line contains a description of the error in English. The third line indicates where the error happened by printing a very abbreviated "backtrace" of the stack (see below); in the example, it is saying that the error was signalled inside the function ***eval**, which was called by **si:lisp-top-level1**.

Here is an example of an error from Lisp code:

```
>>ERROR: The argument X was 1, which is not a symbol,
While in the function FOO ← *EVAL ← SI:LISP-TOP-LEVEL1
```

Here the first line contains the English description of the error message, and the second line contains the abbreviated backtrace. **foo** signalled the error by calling **ferror**; however, **ferror** is censored out of the backtrace.

After the debugger's initial message, it prints the function that got the error and its arguments. Then it prints a list of commands you can use to proceed from the error, or to abort to various command loops. The possibilities depend on the kind of error and where it happened, so the list is different each time; that is why the debugger prints it. The commands in the list all start with Super-A, Super-B and continue as far as is necessary.

eh:*inhibit-debugger-proceed-prompt* *Variable*

If this is non-nil, the list of Super commands is not printed when the debugger is entered. Type Help X to see the list.

debug-io *Variable*

The debugger uses this stream for its I/O. Normally, the value is a synonym stream which indirects to the value of **terminal-io**.

The value of this variable in the stack group in which the error was signaled is the one that counts.

The debugger can be manually entered either by causing an error (e.g. by typing a ridiculous symbol name such as ahsdgf at the Lisp read-eval-print loop) or by typing the Break key with the Meta shift held down while the program is reading from the terminal. Typing the Break key with both Control and Meta held down will force the program into the debugger immediately, even if it is running. If the Break key is typed without Meta, it puts you into a read-eval-print loop using the **break** function (see page 644) rather into the debugger.

eh *process*

Stops *process* and calls the debugger on it so that you can look at its current state. Exit the debugger with the Control-Z command and **eh** will release the process and return. *process* can be a window, in which case the window's process will be used.

If *process* is not a process but a stack group, the current state of the stack group will be examined. The caller should ensure that no one tries to resume that stack group while the debugger is looking at it.

27.7.2 How to Use the Debugger

Once inside the debugger, the user may give a wide variety of commands. This section describes how to give the commands, then explains them in approximate order of usefulness. A summary is provided at the end of the listing.

When the debugger is waiting for a command, it prompts with an arrow:

If the error took place in the evaluation of an expression that you typed at the debugger, you are in a second (or deeper) level error. The number of arrows in the prompt indicates the depth.

The debugger will also warn you about certain unusual circumstances that may cause paradoxical results. If **default-cons-area** is anything except **working-storage-area**, a message to that effect is printed. If **base** and **ibase** are not the same, a message is printed.

At this point, you may type either a Lisp expression or a command; a **Control** or **Meta** character is interpreted as a command, whereas most normal characters are interpreted as the first character of an expression. If you type the **Help** key or the ? key, you will get some introductory help with the debugger.

If you type a Lisp expression, it will be interpreted as a Lisp form and will be evaluated in the context of the function which got the error. That is, all bindings which were in effect at the time of the error will be in effect when your form is evaluated, with certain exceptions explained below. The result of the evaluation will be printed, and the debugger will prompt again with an arrow. If, during the typing of the form, you change your mind and want to get back to the debugger's command level, type the **Abort** key or a **Control-G**; the debugger will respond with an arrow prompt. In fact, at any time that typein is expected from you, while you are in the debugger, you may type **Abort** or **Control-G** to flush what you are doing and get back to command level. This **read-eval-print** loop maintains the values of **+**, *****, and **-** just as the top-level one does.

If an error occurs in the evaluation of the Lisp expression you type, you will get into a second invocation of the debugger, looking at the new error. The prompt will be " ". You can abort the computation and get back to the first error by typing the **Abort** key (see below). However, if the error is trivial the abort will be done automatically and the original error message will be reprinted.

Various debugger commands ask for Lisp objects, such as an object to return or the name of a catch-tag. Whenever it tries to get a Lisp object from you, it expects you to type in a *form*; it will evaluate what you type in. This provides greater generality, since there are objects to which you might want to refer that cannot be typed in (such as arrays). If the form you type is non-trivial (not just a constant form), the debugger will show you the result of the evaluation, and ask you if it is what you intended. It expects a **Y** or **N** answer (see the function **y-or-n-p**, page 620), and if you answer negatively it will ask you for another form. To quit out of the command, just type **Abort** or **Control-G**.

Note that the variable bindings are those in effect at the point of error, *not* those of the current frame being looked at. The **Meta-S** and **Control-Meta-S** commands allow you to look at the bindings in effect at the current frame. A few variables are rebound by the debugger itself, so you you must use **Meta-S** to find the values they had at the point of error:

terminal-io **terminal-io** is rebound to the stream the debugger is using.

standard-input
standard-output
: **standard-input** and **standard-output** are rebound to be synonymous with **terminal-io**.

+
* + and * are rebound to the debugger's previous form and previous value. When the debugger is first entered, + is the last form typed, which is typically the one that caused the error, and * is the value of the *previous* form.

evalhook
si:applyhook These variables (see page 598) are rebound to nil, turning off the step facility if it was in use when the error occurred.

eh:condition-handlers
eh:condition-default-handlers
These are rebound to nil, so that errors occurring within forms you type while in the debugger do not magically resume execution of the erring program.

eh:condition-resume-handlers
To prevent resume handlers established outside the error from being invoked automatically by deeper levels of error, this variable is rebound to a new value, which has an element t added in the front.

27.7.3 Debugger Commands

All debugger commands are single characters, usually with the Control or Meta bits. The single most useful command is Abort (or Control-Z), which exits from the debugger and throws out of the computation that got the error. This is the Abort key, not a 5-letter command. Often you are not interested in using the debugger at all and just want to get back to Lisp top level; so you can do this in one character.

If the error happened while you were innocently using a system utility such as the editor, then it represents a bug in the system. Report the bug using the debugger command Control-M. This gives you an editor preinitialized with the error message and a backtrace. You should type in a *precise* description of what you did that led up to the problem, then send the message by typing End. Be as complete as possible, and always give the exact commands you typed, exact filenames, etc. rather then general descriptions, as much as possible. The person who investigates the bug report will have to try to make the problem happen again; if he does not know where to find *your* file, he will have a difficult time.

The Abort command signals the sys:abort condition, returning control to the most recent command loop. This can be Lisp top level, a break, or the debugger command loop associated with another error. Typing Abort multiple times will throw back to successively older read-eval-print or command loops until top level is reached. Typing Meta-Abort, on the other hand, will always throw to top level. Meta-Abort is not a debugger command, but a system command that is always available no matter what program you are in.

Note that typing Abort in the middle of typing a form to be evaluated by the debugger aborts that form and returns to the debugger's command level, while typing Abort as a debugger command returns out of the debugger and the erring program, to the *previous* command level. Typing Abort after entering a numeric argument just discards the argument.

Self-documentation is provided by the **Help** or **?** command, which types out some documentation on the debugger commands, including any special commands that apply to the particular error currently being handled.

Often you want to try to proceed from the error. When the debugger is entered, it prints a table of commands you can use to proceed, or abort to various levels. The commands are **Super-A**, **Super-B**, and so on. How many there are and what they do is different each time there is an error, but the table says what each one is for. If you want to see the table again, type **Help** followed by X.

The **Resume** (or **Control-C**) command is usually synonymous with **Super-A**. But **Resume** only proceeds, never aborts. If there is no way to proceed, just ways to abort, then **Resume** will not do anything.

The debugger knows about a current stack frame, and there are several commands that use it. The initially current stack frame is the one which signalled the error, either the one which got the microcode-detected error or the one which called **ferror**, **cerror**, or **error**. When the debugger starts it up it shows you this frame in the following format:

```
FOO:
   Arg 0 (X): 13
   Arg 1 (Y): 1
```

and so on. This means that **foo** was called with two arguments, whose names (in the Lisp source code) are x and y. The current values of x and y are 13 and 1 respectively. These may not be the original arguments if **foo** happens to **setq** its argument variables.

The **Clear-Screen** (or **Control-L**) command clears the screen, retypes the error message that was initially printed when the debugger was entered, and prints out a description of the current frame, in the above format.

Several commands are provided to allow you to examine the Lisp control stack and to make frames current other than the one that got the error. The control stack (or "regular pdl") keeps a record of all functions currently active. If you call **foo** at Lisp's top level, and it calls **bar**, which in turn calls **baz**, and **baz** gets an error, then a backtrace (a backwards trace of the stack) would show all of this information. The debugger has two backtrace commands. **Control-B** simply prints out the names of the functions on the stack; in the above example it would print

```
BAZ ← BAR ← FOO ← SI:*EVAL
 ← SI:LISP-TOP-LEVEL1 ← SI:LISP-TOP-LEVEL
```

The arrows indicate the direction of calling. The **Meta-B** command prints a more extensive backtrace, indicating the names of the arguments to the functions and their current values; for the example above it might look like:

```
BAZ:
   Arg 0 (X): 13
   Arg 1 (Y): 1

BAR:
   Arg 0 (ADDEND): 13

FOO:
   Arg 0 (FROB): (A B C . D)
```

and so on.

Moving around in the stack:

The Control-N command moves "down" to the "next" frame (that is, it changes the current frame to be the frame that called it), and prints out the frame in this same format. Control-P moves "up" to the "previous" frame (the one that this one called), and prints out the frame in the same format. Meta-< moves to the top of the stack, and Meta-> to the bottom; both print out the new current frame. Control-S asks you for a string and searches the stack for a frame whose executing function's name contains that string. That frame becomes current and is printed out. These commands are easy to remember since they are analogous to editor commands.

The Control-Meta-N, Control-Meta-P, and Control-Meta-B commands are like the corresponding Control commands but don't censor the stack. When running interpreted code, the debugger usually tries to skip over frames that belong to functions of the interpreter, such as *eval, prog, and cond, and only show "interesting" functions. Control-Meta-N, Control-Meta-P, and Control-Meta-B show everything. They also show frames that are not yet active; that is, frames whose arguments are still being computed for functions that are going to be called. The Control-Meta-U command goes up the stack to the next interesting function and makes that the current frame.

Meta-L prints out the current frame in "full screen" format, which shows the arguments and their values, the local variables and their values, and the machine code with an arrow pointing to the next instruction to be executed. Refer to chapter 28, page 602 for help in reading this machine code.

Commands such as Control-N and Control-P, which are meaningful to repeat, take a prefix numeric argument and repeat that many types. The numeric argument is typed by using Control or Meta and the number keys, as in the editor. Some other commands such as Control-M also use the numeric argument.

Resuming execution:

Meta-C is similar to Control-C, but in the case of an unbound variable or undefined function, actually setqs the variable or defines the function, so that the error will not happen again. Control-C (or Resume) provides a replacement value but does not actually change the variable. Meta-C proceeds using the proceed type :store-new-value, and is available only if that proceed type is provided.

Control-R is used to return a value from the current frame; the frame that called that frame continues running as if the function of the current frame had returned. This command prompts you for a form, which it will evaluate; it returns the resulting value, possibly after confirming it with you.

The Control-T command does a *throw to a given tag with a given value; you are prompted for the tag and the value.

Control-Meta-R is a variation of Control-R; it starts the current frame over with the same function and arguments. If the function has been redefined in the meantime (perhaps you edited it and fixed its bug) the new definition is used. Control-Meta-R asks for confirmation before doing it.

Stepping through function calls and returns:

You can request a trap to the debugger on exit from a particular frame, or the next time a function is called.

Each stack frame has a "trap on exit" bit. The Control-X command toggles this bit. The Meta-X command sets the bit to cause a trap for the current frame and all outer frames. If a program is in an infinite loop, this is a good way to find out how far back on the stack the loop is taking place. The Control-Meta-X command clears the trap-on-exit bit for the current frame and outer frames.

The Control-D command proceeds like Control-C but requests a trap the next time a function is called. The Meta-D command toggles the trap-on-next-call bit for the erring stack group. It is useful if you wish to set the bit and then resume execution with something other than Control-C. The function **breakon** is used to request a trap on calling a particular function. Trapping on entry to a frame automatically sets the trap-on-exit bit for that frame; use Control-X to clear it if you do not want another trap.

Transfering to other systems:

Control-E puts you into the editor, looking at the source code for the function in the current frame. This is useful when you have found the function that caused the error and needs to be fixed. The editor command Control-Z will return to the debugger, if it is still there.

Control-M puts you into the editor to mail a bug report. The error message and a backtrace are put into the editor buffer for you. A numeric argument says how many frames to include in the backtrace.

Control-Meta-W calls the window debugger, a display-oriented debugger. It is not documented in this manual, but should be usable without further documentation.

Examining and setting the arguments, local variables, and values:

Control-Meta-A takes a numeric argument, *n*, and prints out the value of the *n*th argument of the current frame. It leaves * set to the value of the argument, so that you can use the Lisp **read-eval-print** loop to examine it. It also leaves + set to a locative pointing to the argument on the stack, so that you can change that argument (by calling **rplacd** on the locative). **Control-Meta-L** is similar, but refers to the *n*th local variable of the frame. **Control-Meta-V** refers to the *n*th value this frame has returned (in a trap-on-exit). **Control-Meta-F** refers to the function executing in the frame; it ignores its numeric argument and doesn't allow you to change the function.

Another way to examine and set the arguments, locals and values of a frame is with the functions **eh-arg**, **eh-loc**, **eh-val** and **eh-fun**. Use these functions in expressions you evaluate inside the debugger, and they refer to the arguments, locals, values and function, respectively, of the debugger's current frame.

eh-arg *arg-number-or-name*

When used in an expression evaluated in the debugger, **eh-arg** returns the value of the specifed argument in the debugger's current frame. Argument names are compared ignoring packages; only the pname of the symbol you supply is relevant. **eh-arg** can appear in **setf** and **locf** to set an argument or get its location.

eh-loc *local-number-or-name*

eh-loc is just like **eh-arg** but accesses the current frame's locals instead of its arguments.

eh-val &optional *value-number-or-name*

eh-val is used in an expression evaluated in the debugger when the current frame is returning multiple values, to examine those values. This is only useful if the function has already begun to return some values (as in a trap-on-exit), since otherwise they are all **nil**. If a name is specified, it is looked for in the function's **:values** or **:return-list** declaration, if any.

eh-val can be used with **setf** and **locf**. You can make a frame return a specific sequence of values by setting all but the last value with **eh-val** and doing **Control-R** to return the last value.

eh-val with no argument returns a list of all the values this frame is returning.

eh-fun

eh-fun can be called in an expression being evalued inside the debugger to return the function-object being called in the current frame. It can be used with **setf** and **locf**.

27.7.4 Summary of Commands

Control-A	Print argument list of function in current frame.
Control-Meta-A	Examine or change the *n*th argument of the current frame.
Control-B	Print brief backtrace.
Meta-B	Print longer backtrace.
Control-Meta-B	Print longer backtrace with no censoring of interpreter functions.
Control-C or Resume	Attempt to continue, using the first proceed type on the list of available ones for this error.
Meta-C	Attempt to continue, setqing the unbound variable or otherwise "permanently" fixing the error. This uses the proceed type :store-new-value, and is available only if that proceed type is.
Control-D	Attempt to continue like Control-C, but trap on the next function call.
Meta-D	Toggle the flag that causes a trap on the next function call after you continue or otherwise exit the debugger.
Control-E	Edit the source code for the function in the current frame.
Control-Meta-F	Set * to the function in the current frame.
Control-G or Abort	Quit to command level. This is not a command, but something you can type to escape from typing in a form.
Control-L or Clear-Screen	Redisplay error message and current frame.
Meta-L	Full-screen typeout of current frame.
Control-Meta-L	Get local variable *n*.
Control-M	Send a bug report containing the error message and a backtrace of *n* frames (default is 3).
Control-N or Line	Move to next frame. With argument, move down *n* frames.
Meta-N	Move to next frame with full-screen typeout. With argument, move down *n* frames.
Control-Meta-N	Move to next frame even if it is "uninteresting" or still accumulating arguments. With argument, move down *n* frames.
Control-P or Return	Move to previous frame. With argument, move up *n* frames.
Meta-P	Move to previous frame with full-screen typeout. With argument, move up *n* frames.
Control-Meta-P	Move to previous frame even if it is "uninteresting" or still accumulating arguments. With argument, move up *n* frames.
Control-R	Return a value from the current frame.
Meta-R	Return multiple values from the current frame (doesn't work currently).

Control-Meta-R	Reinvoke the function in the current frame (throw back to it and start it over at its beginning.)
Control-S	Search for a frame containing a specified function.
Meta-S	Reads the name of a special variable and returns that variable's value in the current frame. Instance variables of **self** may also be specified even if not special.
Control-Meta-S	Prints a list of special variables bound by the current frame and the values they are bound to by the frame. If the frame binds **self**, all the instance variables of self are listed even if not special.
Control-T	Throw a value to a tag.
Control-Meta-U	Move up the stack to the previous "interesting" frame.
Control-X	Toggle the flag in the current frame that causes a trap on exit or throw through that frame.
Meta-X	Set the flag causing a trap on exit or throw through the frame for the current frame and all the frames outside of it.
Control-Meta-X	Clear the flag causing a trap on exit or throw through the frame for the current frame and all the frames outside of it.
Control-Meta-W	Call the window-oriented debugger.
Control-Z or **Abort**	Abort the computation and throw back to the most recent **break** or debugger, to the program's "command level", or to Lisp top level.
? or **Help**	Print a help message.
Meta-<	Go to top of stack.
Meta->	Go to bottom of stack.
Control-0 through Control-Meta-9	Numeric arguments to the following command are specified by typing a decimal number with Control and/or Meta held down.
Super-A, etc.	The commands Super-A, Super-B, etc. are assigned to all the available proceed types for this error. The assignments are different each time the debugger is entered, so it prints a list of them when it starts up.

27.7.5 Deexposed Windows and Background Processes

If the debugger is entered in a window that is not exposed, a notification is used to inform you that it has happened. The notification appears either as a brief message printed inside square brackets, or as a small window that pops up with a brief message. The notification reminds you that you can select the window in which the error happened by typing **Terminal 0 S**.

If the debugger is entered in a process that has no window or other suitable stream to type out on, the window system tries to assign it a "background window" and print a notification to tell you it is there. If the window system is in a clean state at the moment, this can be done, and you can then type **Terminal 0 S** to select the background window.

However, if the window system cannot notify you, because windows on the screen are locked, it can still inform you of the error. The who-line displays a string containing flashing asterisks that tells you there are errors in the background.

At this time you can either try using Terminal Control-Clear-Input to unlock the locks and allow the notification to be printed normally, or you can elect to handle the error using the cold-load stream by typing Terminal Call. This command normally enters a break-loop that uses the cold-load stream, but if there are any background errors, it offers to enter the debugger to handle them.

27.8 Tracing Function Execution

The trace facility allows the user to *trace* some functions. When a function is traced, certain special actions will be taken when it is called and when it returns. The default tracing action is to print a message when the function is called, showing its name and arguments, and another message when the function returns, showing its name and value(s).

The trace facility is closely compatible with Maclisp. One invokes it through the trace and untrace special forms, whose syntax is described below. Alternatively, you can use the trace system by clicking Trace in the system menu, or by using the Meta-X Trace command in the editor. This allows you to select the trace options from a menu instead of having to remember the following syntax.

trace *Special Form*

A trace form looks like:

(trace *spec-1 spec-2* ...)

Each *spec* can take any of the following forms:

a symbol
: This is a function name, with no options. The function will be traced in the default way, printing a message each time it is called and each time it returns.

a list (*function-name option-1 option-2* ...)
: *function-name* is a symbol and the *options* control how it is to be traced. The various options are listed below. Some options take arguments, which should be given immediately following the option name.

a list (:function *function-spec option-1 option-2* ...)
: This is like the previous form except that *function-spec* need not be a symbol (see section 10.2, page 154). It exists because if *function-name* was a list in the previous form, it would instead be interpreted as the following form:

a list ((*function-1 function-2*...) *option-1 option-2* ...)
: All of the functions are traced with the same options. Each *function* can be either a symbol or a general function-spec.

The following trace options exist:

:break *pred* — Causes a breakpoint to be entered after printing the entry trace information but before applying the traced function to its arguments, if and only if *pred* evaluates to non-nil. During the breakpoint, the symbol **arglist** is bound to a list of the arguments of the function.

:exitbreak *pred* — This is just like **break** except that the breakpoint is entered after the function has been executed and the exit trace information has been printed, but before control returns. During the breakpoint, the symbol **arglist** is bound to a list of the arguments of the function, and the symbol **values** is bound to a list of the values that the function is returning.

:error — Causes the error handler to be called when the function is entered. Use Resume (or Control-C) to continue execution of the function. If this option is specified, there is no printed trace output other than the error message printed by the error handler. This is semi-obsolete, as **breakon** is more convenient and does more exactly the right thing.

:step — Causes the function to be single-stepped whenever it is called. See the documentation on the step facility, section 27.11, page 596.

:entrycond *pred* — Causes trace information to be printed on function entry only if *pred* evaluates to non-**nil**.

:exitcond *pred* — Causes trace information to be printed on function exit only if *pred* evaluates to non-**nil**.

:cond *pred* — This specifies both :exitcond and :entrycond together.

:wherein *function* — Causes the function to be traced only when called, directly or indirectly, from the specified function *function*. One can give several trace specs to **trace**, all specifying the same function but with different **wherein** options, so that the function is traced in different ways when called from different functions.

This is different from **advise-within**, which only affects the function being advised when it is called directly from the other function. The **trace** :wherein option means that when the traced function is called, the special tracing actions occur if the other function is the caller of this function, or its caller's caller, or its caller's caller's caller, etc.

:argpdl *pdl* — This specifies a symbol *pdl*, whose value is initially set to **nil** by **trace**. When the function is traced, a list of the current recursion level for the function, the function's name, and a list of arguments is consed onto the *pdl* when the function is entered, and cdr'ed back off when the function is exited. The *pdl* can be inspected from within a breakpoint, for example, and used to determine the very recent history of the function. This option can be used with or without printed trace output. Each function can be given its own pdl, or one pdl may serve several functions.

:entryprint *form* — The *form* is evaluated and the value is included in the trace message for calls to the function. You can give this option more than once, and all the values will appear, preceded by \\.

:exitprint *form* — The *form* is evaluated and the value is included in the trace message for returns from the function. You can give this option more than once, and all the values will appear, preceded by \\.

:print *form* — The *form* is evaluated and the value is included in the trace messages for both calls to and returns from the function. You can give this option more than once, and all the values will appear, preceded by \\.

:entry *list* — This specifies a list of arbitrary forms whose values are to be printed along with the usual entry-trace. The list of resultant values, when printed, is preceded by \\ to separate it from the other information.

:exit *list* — This is similar to entry, but specifies expressions whose values are printed with the exit-trace. Again, the list of values printed is preceded by \\.

:arg :value :both nil — These specify which of the usual trace printouts should be enabled. If :arg is specified, then on function entry the name of the function and the values of its arguments will be printed. If :value is specified, then on function exit the returned value(s) of the function will be printed. If :both is specified, both of these will be printed. If nil is specified, neither will be printed. If none of these four options are specified the default is to :both. If any further *options* appear after one of these, they will not be treated as options! Rather, they will be considered to be arbitrary forms whose values are to be printed on entry and/or exit to the function, along with the normal trace information. The values printed will be preceded by a //. and follow any values specified by :entry or :exit. Note that since these options "swallow" all following options, if one is given it should be the last option specified.

If the variable arglist is used in any of the expressions given for the :cond, :break, :entry, or :exit options, or after the :arg, :value, :both, or nil option, when those expressions are evaluated the value of arglist will be bound to a list of the arguments given to the traced function. Thus

```
(trace (foo :break (null (car arglist))))
```

would cause a break in foo if and only if the first argument to foo is nil. If the :break or :error option is used, the variable arglist will be valid inside the break-loop. If you setq arglist, the arguments seen by the function will change. arglist should perhaps have a colon, but it can be omitted because this is the name of a system function and therefore global.

Similarly, the variable values will be a list of the resulting values of the traced function. For obvious reasons, this should be used only with the :exit option. If the :exitbreak option is used, the variables values and arglist are valid inside the break-loop. If you setq values, the values returned by the function will change. values should perhaps have a colon, but it can be omitted because this is the name of a system function and therefore global.

The trace specifications may be "factored", as explained above. For example,

```
(trace ((foo bar) :break (bad-p arglist) :value))
```

is equivalent to

```
(trace (foo :break (bad-p arglist) :value)
       (bar :break (bad-p arglist) :value))
```

Since a list as a function name is interpreted as a list of functions, non-atomic function names (see section 10.2, page 154) are specified as follows:

```
(trace (:function (:method flavor :message) :break t))
```

trace returns as its value a list of names of all functions it traced. If called with no arguments, as just (trace), it returns a list of all the functions currently being traced.

If you attempt to trace a function already being traced, **trace** calls **untrace** before setting up the new trace.

Tracing is implemented with encapsulation (see section 10.10, page 175), so if the function is redefined (e.g. with **defun** or by loading it from a QFASL file) the tracing will be transferred from the old definition to the new definition.

Tracing output is printed on the stream that is the value of **trace-output**. This is synonymous with **terminal-io** unless you change it.

untrace *Special Form*

untrace is used to undo the effects of trace and restore functions to their normal, untraced state. untrace will take multiple specifications, e.g. (untrace foo quux fuphoo). Calling untrace with no arguments will untrace all functions currently being traced.

Unlike Maclisp, if there is an error **trace** (or **untrace**) will invoke the error system and give an English message, instead of returning lists with question marks in them. Also, the **remtrace** function is not provided, since it is unnecessary.

trace-compile-flag *Variable*

If the value of trace-compile-flag is non-nil, the functions created by trace will get compiled, allowing you to trace special forms such as cond without interfering with the execution of the tracing functions. The default value of this flag is nil.

You can also cause the tracing of a particular function to be compiled by calling **compile-encapsulations**. Set **compile-encapsulations-flag** non-nil does what **trace-compile-flag** does, and more; it makes all kinds of encapsulations be compiled. See page 229.

27.9 Breakon

The function **breakon** allows you to request that the debugger be entered whenever a certain function is called.

breakon *function-spec* &optional *condition-form*

Encapsulate the definition of *function-spec* so that a trap-on-call occurs when it is called. This enters the debugger. A trap-on-exit will occur when the stack frame is exited.

If *condition-form* is non-nil, its value should be a form to be evaluated each time *function-spec* is called. The trap occurs only if *condition-form* evaluates to non-nil. Omitting the *condition-form* is equivalent to supplying t. If breakon is called more than once for the same *function-spec* and different *condition-forms*, the trap occurs if any of the conditions is true.

Condition breakons are useful for causing the trap to occur only in a certain stack group. This sometimes allows debugging of calls functions that are being used frequently in background processes.

```
(breakon 'foo '(eq current-stack-group ',current-stack-group))
```

If you wish to trap on calls to foo when called from the execution of bar, you can use (si:function-active-p 'bar) as the condition. If you want to trap only calls made directly from bar, the thing to do is

```
(breakon '(:within bar foo))
```

rather than a conditional breakon.

Another useful form of conditional breakon allows you to control trapping from the keyboard:

```
(breakon 'foo '(tv:key-state ':mode-lock))
```

The trap will occur only when the Mode-Lock key is down. This key is not normally used for much else. With this technique, you can successfully trap on functions used by the debugger!

unbreakon *function-spec* &optional *conditional-form*

Remove the breakon set on *function-spec*. If *conditional-form* is specified, remove only that condition. Breakons with other conditions are not removed.

With no arguments, unbreakon removes all breakons from all functions.

eh:breakon-functions *Variable*

A list of all function specs on which breakons currently exist.

To cause the encapsulation which implements the breakon to be compiled, call compile-encapsulations or set compile-encapsulations-flag non-nil. See page 229. This may eliminate some of the problems that occur if you breakon a function such as prog that is used by the evaluator. (A conditional to trap only in one stack group will help here also.)

27.10 Advising a Function

To advise a function is to tell it to do something extra in addition to its actual definition. It is done by means of the function advise. The something extra is called a piece of advice, and it can be done before, after, or around the definition itself. The advice and the definition are independent, in that changing either one does not interfere with the other. Each function can be given any number of pieces of advice.

Advising is fairly similar to tracing, but its purpose is different. Tracing is intended for temporary changes to a function to give the user information about when and how the function is called and when and with what value it returns. Advising is intended for semi-permanent changes to what a function actually does. The differences between tracing and advising are motivated by this difference in goals.

Advice can be used for testing out a change to a function in a way that is easy to retract. In this case, you would call advise from the terminal. It can also be used for customizing a function that is part of a program written by someone else. In this case you would be likely to put a call to advise in one of your source files or your login init file (see page 648), rather than modifying the other person's source code.

Advising is implemented with encapsulation (see section 10.10, page 175), so if the function is redefined (e.g. with **defun** or by loading it from a QFASL file) the advice will be transferred from the old definition to the new definition.

advise *Special Form*

A function is advised by the special form

(advise *function class name position*
form1 form2...)

None of this is evaluated. *function* is the function to put the advice on. It is usually a symbol, but any function spec is allowed (see section 10.2, page 154). The *forms* are the advice; they get evaluated when the function is called. *class* should be either **:before**, **:after**, or **:around**, and says when to execute the advice (before, after, or around the execution of the definition of the function). The meaning of **:around** advice is explained a couple of sections below.

name is used to keep track of multiple pieces of advice on the same function. *name* is an arbitrary symbol that is remembered as the name of this particular piece of advice. If you have no name in mind, use **nil**; then we say the piece of advice is anonymous. A given function and class can have any number of pieces of anonymous advice, but it can have only one piece of named advice for any one name. If you try to define a second one, it replaces the first. Advice for testing purposes is usually anonymous. Advice used for customizing someone else's program should usually be named so that multiple customizations to one function have separate names. Then, if you reload a customization that is already loaded, it does not get put on twice.

position says where to put this piece of advice in relation to others of the same class already present on the same function. If *position* is *nil*, the new advice goes in the default position: it usually goes at the beginning (where it is executed before the other advice), but if it is replacing another piece of advice with the same name, it goes in the same place that the old piece of advice was in.

If you wish to specify the position, *position* can be the numerical index of which existing piece of advice to insert this one before. Zero means at the beginning; a very large number means at the end. Or, *position* can be the name of an existing piece of advice of the same class on the same function; the new advice is inserted before that one.

For example,

```
(advise factorial :before negative-arg-check nil
  (if (minusp (first arglist))
      (ferror nil "factorial of negative argument")))
```

This modifies the factorial function so that if it is called with a negative argument it signals an error instead of running forever.

unadvise *Special Form*

(unadvise *function class position*)

removes pieces of advice. None of its "arguments" are evaluated. *function* and *class* have the same meaning as they do in the function **advise**. *position* specifies which piece of advice to remove. It can be the numeric index (zero means the first one) or it can be the name of the piece of advice.

unadvise can remove more than one piece of advice if some of its arguments are missing. If *position* is missing or nil, then all advice of the specified class on the specified function is removed. If *class* is missing or nil as well, then all advice on the specified function is removed. (unadvise) removes all advice on all functions, since *function* is not specified.

The following are the primitive functions for adding and removing advice. Unlike the above special forms, these are functions and can be conveniently used by programs. advise and unadvise are actually macros that expand into calls to these two.

si:advise-1 *function class name position forms*

Adds advice. The arguments have the same meaning as in advise. Note that the *forms* argument is *not* a &rest argument.

si:unadvise-1 *function* &optional *class position*

Removes advice. If *class* or *position* is nil or unspecified, all classes of advice or advice at all positions are removed.

You can find out manually what advice a function has with grindef, which grinds the advice on the function as forms that are calls to advise. These are in addition to the definition of the function.

To poke around in the advice structure with a program, you must work with the encapsulation mechanism's primitives. See section 10.10, page 175.

To cause the advice to be compiled, call compile-encapsulations or set compile-encapsulations-flag non-nil. See page 229.

si:advised-functions *Variable*

A list of all functions which have been advised.

27.10.1 Designing the Advice

For advice to interact usefully with the definition and intended purpose of the function, it must be able to interface to the data flow and control flow through the function. We provide conventions for doing this.

The list of the arguments to the function can be found in the variable arglist. :before advice can replace this list, or an element of it, to change the arguments passed to the definition itself. If you replace an element, it is wise to copy the whole list first with

```
(setq arglist (copylist arglist))
```

After the function's definition has been executed, the list of the values it returned can be found in the variable values. :after advice can set this variable or replace its elements to cause different values to be returned.

All the advice is executed within a prog, so any piece of advice can exit the entire function with return. The arguments of the return will be returned as the values of the function. No further advice will be executed. If a piece of :before advice does this, then the function's definition will not even be called.

27.10.2 :around Advice

A piece of :before or :after advice is executed entirely before or entirely after the definition of the function. :around advice is wrapped around the definition; that is, the call to the original definition of the function is done at a specified place inside the piece of :around advice. You specify where by putting the symbol :do-it in that place.

For example, (+ 5 :do-it) as a piece of :around advice would add 5 to the value returned by the function. This could also be done by (setq values (list (+ 5 (car values)))) as :after advice.

When there is more than one piece of :around advice, the pieces are stored in a sequence just like :before and :after advice. Then, the first piece of advice in the sequence is the one started first. The second piece is substituted for :do-it in the first one. The third one is substituted for :do-it in the second one. The original definition is substituted for :do-it in the last piece of advice.

:around advice can access arglist, but values is not set up until the outermost :around advice returns. At that time, it is set to the value returned by the :around advice. It is reasonable for the advice to receive the values of the :do-it (e.g. with multiple-value-list) and fool with them before returning them (e.g. with values-list).

:around advice can return from the prog at any time, whether the original definition has been executed yet or not. It can also override the original definition by failing to contain :do-it. Containing two instances of :do-it may be useful under peculiar circumstances. If you are careless, the original definition may be called twice, but something like

```
(if (foo) (+ 5 :do-it) (* 2 :do-it))
```

will certainly work reasonably.

27.10.3 Advising One Function Within Another

It is possible to advise the function foo only for when it is called directly from a specific other function bar. You do this by advising the function specifier (:within bar foo). That works by finding all occurrences of foo in the definition of bar and replacing them with altered-foo-within-bar. This can be done even if bar's definition is compiled code. The symbol altered-foo-within-bar starts off with the symbol foo as its definition; then the symbol altered-foo-within-bar, rather than foo itself, is advised. The system remembers that foo has been replaced inside bar, so that if you change the definition of bar, or advise it, then the replacement is propagated to the new definition or to the advice. If you remove all the advice on (:within bar foo), so that its definition becomes the symbol foo again, then the replacement is unmade and everything returns to its original state.

(grindef bar) will print foo where it originally appeared, rather than altered-foo-within-bar, so the replacement will not be seen. Instead, grindef will print out calls to advise to describe all the advice that has been put on foo or anything else within bar.

An alternate way of putting on this sort of advice is to use advise-within.

advise-within *Special Form*

(advise-within *within-function function-to-advise class name position forms...*)

advises *function-to-advise* only when called directly from the function *within-function*. The other arguments mean the same thing as with **advise**. None of them are evaluated.

To remove advice from (:within bar foo), you can use **unadvise** on that function specifier. Alternatively, you can use **unadvise-within**.

unadvise-within *Special Form*

(unadvise-within *within-function function-to-advise class position*)

removes advice that has been placed on (:within *within-function function-to-advise*). The arguments *class* and *position* are interpereted as for **unadvise**. For example, if those two are omitted, then all advice placed on *function-to-advise* within *within-function* is removed. Additionally, if *function-to-advise* is omitted, all advice on any function within *within-function* is removed. If there are no arguments, than all advice on one function within another is removed. Other pieces of advice, which have been placed on one function and not limited to within another, are not removed.

(unadvise) removes absolutely all advice, including advice for one function within another.

The function versions of advise-within and unadvise-within are called si:advise-within-1 and si:unadvise-within-1. advise-within and unadvise-within are macros that expand into calls to the other two.

27.11 Stepping Through an Evaluation

The Step facility gives you the ability to follow every step of the evaluation of a form, and examine what is going on. It is analogous to a single-step proceed facility often found in machine-language debuggers. If your program is doing something strange, and it isn't obvious how it's getting into its strange state, then the stepper is for you.

There are two ways to enter the stepper. One is by use of the **step** function.

step *form*

This evaluates *form* with single stepping. It returns the value of *form*.

For example, if you have a function named foo, and typical arguments to it might be t and 3, you could say

```
(step '(foo t 3))
```

and the form (foo t 3) will be evaluated with single stepping.

The other way to get into the stepper is to use the :step option of **trace** (see page 588). If a function is traced with the :step option, then whenever that function is called it will be single stepped.

Note that any function to be stepped must be interpreted; that is, it must be a lambda-expression. Compiled code cannot be stepped by the stepper.

When evaluation is proceeding with single stepping, before any form is evaluated, it is (partially) printed out, preceded by a forward arrow (→) character When a macro is expanded, the expansion is printed out preceded by a double arrow (⇔) character. When a form returns a value, the form and the values are printed out preceded by a backwards arrow (←) character; if there is more than one value being returned, an and-sign (∧) character is printed between the values. When the stepper has evaluated the args to a form and is about to apply the function, it prints a lambda (λ) because entering the lambda is the next thing to be done.

Since the forms may be very long, the stepper does not print all of a form; it truncates the printed representation after a certain number of characters. Also, to show the recursion pattern of who calls whom in a graphic fashion, it indents each form proportionally to its level of recursion.

After the stepper prints any of these things, it waits for a command from the user. There are several commands to tell the stepper how to proceed, or to look at what is happening. The commands are:

Control-N (Next)
: Step to the Next thing. The stepper continues until the next thing to print out, and it accepts another command.

Space
: Go to the next thing at this level. In other words, continue to evaluate at this level, but don't step anything at lower levels. This is a good way to skip over parts of the evaluation that don't interest you.

Control-A (Args)
: Skip over the evaluation of the arguments of this form, but pause in the stepper before calling the function that is the car of the form.

Control-U (Up)
: Continue evaluating until we go up one level. This is like the space command, only more so; it skips over anything on the current level as well as lower levels.

Control-X (eXit)
: Exit; finish evaluating without any more stepping.

Control-T (Type)
: Retype the current form in full (without truncation).

Control-G (Grind)
: Grind (i.e. prettyprint) the current form.

Control-E (Editor)
: Switch windows, to the editor.

Control-B (Breakpoint)
: Breakpoint. This command puts you into a breakpoint (i.e. a read-eval-print loop) from which you can examine the values of variables and other aspects of the current environment. From within this loop, the following variables are available:

step-form which is the current form.

step-values which is the list of returned values.

step-value which is the first returned value.

If you change the values of these variables, you will affect execution.

Control-L
Clear the screen and redisplay the last 10. pending forms (forms that are being evaluated).

Meta-L
Like Control-L, but doesn't clear the screen.

Control-Meta-L
Like Control-L, but redisplays all pending forms.

? or Help
Prints documentation on these commands.

It is strongly suggested that you write some little function and try the stepper on it. If you get a feel for what the stepper does and how it works, you will be able to tell when it is the right thing to use to find bugs.

27.12 Evalhook

The evalhook facility provides a "hook" into the evaluator; it is a way you can get a Lisp form of your choice to be executed whenever the evaluator is called. The stepper uses evalhook, and usually it is the only thing that ever needs to. However, if you want to write your own stepper or something similar, this is the primitive facility that you can use to do so. The way this works is a bit hairy, but unless you need to write your own stepper you don't have to worry about it.

evalhook *Variable*

If the value of evalhook is non-nil, then special things happen in the evaluator. When a form (any form, even a number or a symbol) is to be evaluated, evalhook is bound to nil and the function that was evalhook's value is applied to one argument—the form that was trying to be evaluated. The value it returns is then returned from the evaluator.

si:applyhook *Variable*

If the value of si:applyhook is non-nil, it is used rather than apply the next time the interpreter is about to apply a function to its evaluated arguments.

When either the evalhook or the applyhook is called, both variables are bound to nil. They are also rebound to nil by break and by the debugger, and setq'ed to nil when errors are dismissed by throwing to the Lisp top level loop. This provides the ability to escape from this mode if something bad happens.

In order not to impair the efficiency of the Lisp interpreter, several restrictions are imposed on the evalhook and applyhook. They apply only to evaluation—whether in a read-eval-print loop, internally in evaluating arguments in forms, or by explicit use of the function eval. They *do not* have any effect on compiled function references, on use of the function apply, or on the "mapping" functions. (In Zetalisp, as opposed to Maclisp, it is not necessary to do (*rset t) or (sstatus evalhook t). Also, Maclisp's special-case check for store is not implemented.)

evalhook *form evalhook* &optional *applyhook*

evalhook is a function that helps exploit the evalhook feature. The *form* is evaluated with evalhook lambda-bound to the function *evalhook*, and with si:applyhook lambda-bound to the function *applyhook*. The checking of the hooks is bypassed in the evaluation of *form* itself, but not in any subsidiary evaluations, for instance of arguments in the *form*. This is like a "one-instruction proceed" in a machine-language debugger.

Example:

```
;; This function evaluates a form while printing debugging information.
(defun hook (x)
   (terpri)
   (evalhook x 'hook-function))

;; Notice how this function calls evalhook to evaluate the form f,
;; so as to hook the sub-forms.
(defun hook-function (f)
   (let ((v (evalhook f 'hook-function)))
     (format t "form: ~s~%value: ~s~%" f v)
     v))

;; This isn't a very good program, since if f uses multiple
;; values, it will not work.
```

The following output might be seen from (hook '(cons (car '(a . b)) 'c)):

```
form: (quote (a . b))
value: (a . b)
form: (car (quote (a . b)))
value: a
form: (quote c)
value: c
(a . c)
```

27.13 The MAR

The MAR facility allows any word or contiguous set of words to be monitored constantly, and can cause an error if the words are referenced in a specified manner. The name MAR is from the similar device on the ITS PDP-10's; it is an acronym for "Memory Address Register". The MAR checking is done by the Lisp Machine's memory management hardware, so the speed of general execution is not significantly slowed down when the MAR is enabled. However, the speed of accessing pages of memory containing the locations being checked is slowed down somewhat, since every reference involves a microcode trap.

These are the functions that control the MAR:

set-mar *location cycle-type* &optional *n-words*

The set-mar function clears any previous setting of the MAR, and sets the MAR on *n-words* words, starting at *location*. *location* may be any object. Often it will be a locative pointer to a cell, probably created with the locf special form. *n-words* currently defaults to 1, but eventually it may default to the size of the object. *cycle-type* says under what conditions to trap. :read means that only reading the location should cause an error, :write means that only writing the location should, t means that both should. To set the MAR to detect setq (and binding) of the variable foo, use

```
(set-mar (value-cell-location 'foo) ':write)
```

clear-mar

This turns off the MAR. Warm-booting the machine disables the MAR but does not turn it off, i.e. references to the MARed pages are still slowed down. clear-mar does not currently speed things back up until the next time the pages are swapped out; this may be fixed some day.

mar-mode

(mar-mode) returns a symbol indicating the current state of the MAR. It returns one of:

nil	The MAR is not set.
:read	The MAR will cause an error if there is a read.
:write	The MAR will cause an error if there is a write.
t	The MAR will cause an error if there is any reference.

Note that using the MAR makes the pages on which it is set somewhat slower to access, until the next time they are swapped out and back in again after the MAR is shut off. Also, use of the MAR currently breaks the read-only feature if those pages were read-only.

Proceeding from a MAR break allows the memory reference that got an error to take place, and continues the program with the MAR still effective. When proceeding from a write, you have the choice of whether to allow the write to take place or to inhibit it, leaving the location with its old contents.

sys:mar-break (condition) *Condition*

This is the condition, not an error, signaled by a MAR break.

The condition instance supports these operations:

:object	The object one of whose words was being referenced.
:offset	The offset within the object of the word being referenced.
:value	The value read, or to be written.
:direction	Either :read or :write.

The proceed type :no-action simply proceeds, continuing with the interrupted program as if the MAR had not been set. If the trap was due to writing, the proceed type :proceed-no-write is also provided, and causes the program to proceed but does not store the value in the memory location.

Most—but not all—write operations first do a read. **setq** and **rplaca** are examples. This means that if the MAR is in **:read** mode it will catch writes as well as reads; however, they will trap during the reading phase, and consequently the data to be written will not be displayed. This also means that setting the MAR to **t** mode causes most writes to trap twice, first for a read and then again for a write. So when the MAR says that it trapped because of a read, this means a read at the hardware level, which may not look like a read in your program.

27.14 Variable Monitoring

monitor-variable *var* &optional *current-value-cell-only-p monitor-function*

Calls a given function just after a given special variable is **setq**'ed (by compiled code or otherwise). Does not trigger on binding of the variable. The function is given both the old and new values as arguments. It does not get the name of the variable as an argument, so it is usually necessary to use a closure as *monitor-function* in order to remember this. The old value will be **nil** if the variable had been unbound.

The default monitoring function just prints the symbol and the old and new values. This behavior can be changed by specifying the *monitor-function* argument.

Normally this feature applies to all **setq**'s, but if *current-value-cell-only-p* is specified non-**nil**, it applies only to those **setq**'s which would alter the variable's currently active value cell. This is only relevant when *var* is subject to a closure.

Don't try to use this with variables that are forwarded to A memory (e.g. **inhibit-scheduling-flag**).

unmonitor-variable &optional *var*

If *var* is being monitored, it is restored to normal. If no *var* is specified, all variables that have been monitored are unmonitored.