3. Evaluation

The following is a complete description of the actions taken by the evaluator, given a *form* to evaluate.

If *form* is a number, the result is *form*.

If *form* is a string, the result is *form*.

If *form* is a symbol, the result is the value of *form*, considered as a variable. If *form* is unbound, an error is signalled. The way symbols are bound to values is explained in section 3.1, page 15 below.

If *form* is not any of the above types, and is not a list, an error is signalled.

In all remaining cases, *form* is a list. The evaluator examines the car of the list to figure out what to do next. There are three possibilities: this form may be a *special form*, a *macro form*, or a plain old *function form*. Conceptually, the evaluator knows specially about all the symbols whose appearance in the car of a form make that form a special form, but the way the evaluator actually works is as follows. If the car of the form is a symbol, the evaluator finds the object in the function cell of the symbol (see chapter 6, page 96) and starts all over as if that object had been the car of the list. If the car isn't a symbol, then if it's a cons whose car is the symbol **macro**, then this is a macro form; if it is a "special function" (see page 164) then this is a special form; otherwise, it should be a regular function, and this is a function form.

If *form* is a special form, then it is handled accordingly; each special form works differently. All of them are documented in this manual. The internal workings of special forms are explained in more detail on page 164, but this hardly ever affects you.

If *form* is a macro form, then the macro is expanded as explained in chapter 17.

If *form* is a function form, it calls for the *application* of a function to *arguments*. The car of the form is a function or the name of a function. The cdr of the form is a list of subforms. Each subform is evaluated, sequentially. The values produced by evaluating the subforms are called the "arguments" to the function. The function is then applied to those arguments. Whatever results the function *returns* are the values of the original *form*.

There is a lot more to be said about evaluation. The way variables work and the ways in which they are manipulated, including the binding of arguments, is explained in section 3.1, page 15. A basic explanation of functions is in section 3.2, page 21. The way functions can return more than one value is explained in section 3.5, page 33. The description of all of the kinds of functions, and the means by which they are manipulated, is in chapter 10. Macros are explained in chapter 17. The **evalhook** facility, which lets you do something arbitrary whenever the evaluator is invoked, is explained in section 27.12, page 598. Special forms are described all over the manual; each special form is in the section on the facility it is part of.

3.1 Variables

In Zetalisp, variables are implemented using symbols. Symbols are used for many things in the language, such as naming functions, naming special forms, and being keywords; they are also useful to programs written in Lisp, as parts of data structures. But when the evaluator is given a symbol, it treats it as a variable, using the value cell to hold the value of the variable. If you evaluate a symbol, you get back the contents of the symbol's value cell.

There are two different ways of changing the value of a variable. One is to *set* the variable. Setting a variable changes its value to a new Lisp object, and the previous value of the variable is forgotten. Setting of variables is usually done with the **setq** special form.

The other way to change the value of a variable is with *binding* (also called "lambda-binding"). When a variable is bound, its old value is first saved away, and then the value of the variable is made to be the new Lisp object. When the binding is undone, the saved value is restored to be the value of the variable. Bindings are always followed by unbindings. The way this is enforced is that binding is only done by special forms that are defined to bind some variables, then evaluate some subforms, and then unbind those variables. So the variables are all unbound when the form is finished. This means that the evaluation of the form doesn't disturb the values of the variables that are bound; whatever their old value was, before the evaluation of the form, gets restored when the evaluation of the form is completed. If such a form is exited by a non-local exit of any kind, such as ***throw** (see page 55) or **return** (see page 52), the bindings are undone whenever the form is exited.

The simplest construct for binding variables is the **let** special form. The **do** and **prog** special forms can also bind variables, in the same way **let** does, but they also control the flow of the program and so are explained elsewhere (see page 45). **let*** is just a sequential version of **let**; the other special forms below are only used for esoteric purposes.

Binding is an important part of the process of applying interpreted functions to arguments. This is explained in the next section.

When a Lisp function is compiled, the compiler understands the use of symbols as variables. However, the compiled code generated by the compiler does not actually use symbols to represent variables. Rather, the compiler converts the references to variables within the program into more efficient references, that do not involve symbols at all. A variable that has been changed by the compiler so that it is not implemented as a symbol is called a "local" variable. When a local variable is bound, a memory cell is allocated in a hidden, internal place (the Lisp control stack) and the value of the variable is stored in this cell. You cannot use a local variable without first binding it; you can only use a local variable inside of a special form that binds that variable. Local variables do not have any "top level" value; they do not even exist outside of the form that binds them.

The variables which are associated with symbols (the kind which are used by non-compiled programs) are called "special" variables.

Local variables and special variables do not behave quite the same way, because "binding" means different things for the two of them. Binding a special variable saves the old value away and then uses the value cell of the symbol to hold the new value, as explained above. Binding a

local variable, however, does not do anything to the symbol. In fact, it creates a new memory cell to hold the value, i.e. a new local variable.

Thus, if you compile a function, it may do different things after it has been compiled. Here is an example:

```
(setq a 2)              ;Set the variable a to the value 2.

(defun foo ()           ;Define a function named foo.
  (let ((a 5))          ;Bind the symbol a to the value 5.
    (bar)))             ;Call the function bar.

(defun bar ()           ;Define a function named bar.
  a)                    ;It just returns the value of the variable a.

(foo) => 5              ;Calling foo returns 5.

(compile 'foo)          ;Now compile foo.

(foo) => 2              ;This time, calling foo returns 2.
```

This is a very bad thing, because the compiler is only supposed to speed things up, without changing what the function does. Why did the function **foo** do something different when it was compiled? Because **a** was converted from a special variable into a local variable. After **foo** was compiled, it no longer had any effect on the value cell of the symbol **a**, and so the symbol retained its old contents, namely **2**.

In most uses of variables in Lisp programs, this problem doesn't come up. The reason it happened here is because the function **bar** refers to the symbol **a** without first binding **a** to anything. A reference to a variable that you didn't bind yourself is called a *free reference*; in this example, **bar** makes a free reference to **a**.

We mentioned above that you can't use a local variable without first binding it. Another way to say this is that you can't ever have a free reference to a local variable. If you try to do so, the compiler will complain. In order for our functions to work, the compiler must be told *not* to convert **a** into a local variable; **a** must remain a special variable. Normally, when a function is compiled, all variables in it are made to be "local". You can stop the compiler from making a variable local by "declaring" to the compiler that the variable is "special". When the compiler sees references to a variable that has been declared special, it uses the symbol itself as the variable instead of making a local variable.

Variables can be declared by the special forms **defvar** and **defconst** (see below), or by explicit compiler declarations (see page 235). The most common use of special variables is as "global" variables: variables used by many different functions throughout a program, that have top-level values.

Had **bar** been compiled, the compiler would have seen the free reference and printed a warning message: **Warning: a declared special**. It would have automatically declared **a** to be special and proceeded with the compilation. It knows that free references mean that special

declarations are needed. But when a function is compiled that binds a variable that you want to be treated as a special variable but that you have not explicitly declared, there is, in general, no way for the compiler to automatically detect what has happened, and it will produce incorrect output. So you must always provide declarations for all variables that you want to be treated as special variables.

When you make a variable special with **defvar** or **defconst**, or with a global special declaration, the variable becomes special in all functions that use it. You can also declare a variable special in one function or expression only, using **local-declare** (see page 233) or a **declare** at the front of a function (see page 234). These techniques are useful when the special variable is needed only for communication between a handful of functions defined near each other, or between one function and its internal functions.

Here are the special forms used for setting variables.

setq {*variable value*}... *Special Form*

The **setq** special form is used to set the value of a variable or of many variables. The first *value* is evaluated, and the first *variable* is set to the result. Then the second *value* is evaluated, the second *variable* is set to the result, and so on for all the variable/value pairs. **setq** returns the last value, i.e. the result of the evaluation of its last subform.
Example:

```
(setq x (+ 3 2 1) y (cons x nil))
```

x is set to 6, **y** is set to (6), and the **setq** form returns (6). Note that the first variable was set before the second value form was evaluated, allowing that form to use the new value of **x**.

psetq {*variable value*}... *Special Form*

A **psetq** form is just like a **setq** form, except that the variables are set "in parallel"; first all of the *value* forms are evaluated, and then the *variables* are set to the resulting values.
Example:

```
(setq a 1)
(setq b 2)
(psetq a b b a)
a => 2
b => 1
```

Here are the special forms used for binding variables.

let ((*var value*)...) *body*... *Special Form*

let is used to bind some variables to some objects, and evaluate some forms (the "body") in the context of those bindings. A **let** form looks like

```
(let ((var1 vform1)
      (var2 vform2)
      ...)
  bform1
  bform2
  ...)
```

When this form is evaluated, first the *vforms* (the values) are evaluated. Then the *vars* are bound to the values returned by the corresponding *vforms*. Thus the bindings happen in

parallel; all the *vforms* are evaluated before any of the *vars* are bound. Finally, the *bforms* (the body) are evaluated sequentially, the old values of the variables are restored, and the result of the last *bform* is returned.

You may omit the *vform* from a **let** clause, in which case it is as if the *vform* were **nil**: the variable is bound to **nil**. Furthermore, you may replace the entire clause (the list of the variable and form) with just the variable, which also means that the variable gets bound to **nil**. Example:

```
(let ((a (+ 3 3))
      (b 'foo)
      (c)
      d)
  ...)
```

Within the body, **a** is bound to **6**, **b** is bound to **foo**, **c** is bound to **nil**, and **d** is bound to **nil**.

let* ((*var value*)...) *body...* *Special Form*

let* is the same as **let** except that the binding is sequential. Each *var* is bound to the value of its *vform* before the next *vform* is evaluated. This is useful when the computation of a *vform* depends on the value of a variable bound in an earlier *vform*. Example:

```
(let* ((a (+ 1 2))
       (b (+ a a)))
  ...)
```

Within the body, **a** is bound to **3** and **b** is bound to **6**.

let-if *condition* ((*var value*)...) *body...* *Special Form*

let-if is a variant of **let** in which the binding of variables is conditional. The variables must all be special variables. The **let-if** special form, typically written as

```
(let-if cond
        ((var-1 val-1) (var-2 val-2)...)
  body-form1 body-form2...)
```

first evaluates the predicate form *cond*. If the result is non-**nil**, the value forms *val-1*, *val-2*, etc. are evaluated and then the variables *var-1*, *var-2*, etc. are bound to them. If the result is **nil**, the *vars* and *vals* are ignored. Finally the body forms are evaluated.

let-globally ((*var value*)...) *body...* *Special Form*

let-globally is similar in form to **let** (see page 17). The difference is that **let-globally** does not *bind* the variables; instead, it saves the old values and *sets* the variables, and sets up an **unwind-protect** (see page 56) to set them back. The important difference between **let-globally** and **let** is that when the current stack group (see chapter 12, page 186) co-calls some other stack group, the old values of the variables are *not* restored. Thus **let-globally** makes the new values visible in all stack groups and processes that don't bind the variables themselves, not just the current stack group.

progv *symbol-list value-list body...* *Special Form*

progv is a special form to provide the user with extra control over binding. It binds a list of special variables to a list of values, and then evaluates some forms. The lists of special variables and values are computed quantities; this is what makes progv different from let, prog, and do.

progv first evaluates *symbol-list* and *value-list*, and then binds each symbol to the corresponding value. If too few values are supplied, the remaining symbols are bound to nil. If too many values are supplied, the excess values are ignored.

After the symbols have been bound to the values, the *body* forms are evaluated, and finally the symbols' bindings are undone. The result returned is the value of the last form in the body.
Example:

```
(setq a 'foo b 'bar)

(progv (list a b 'b) (list b)
  (list a b foo bar))
    => (foo nil bar nil)
```

During the evaluation of the body of this progv, foo is bound to bar, bar is bound to nil, b is bound to nil, and a retains its top-level value foo.

progw *vars-and-vals-form body...* *Special Form*

progw is a somewhat modified kind of progv. Like progv, it only works for special variables. First, *vars-and-val-forms-form* is evaluated. Its value should be a list that looks like the first subform of a let:

```
((var1 val-form-1)
 (var2 val-form-2)
 ...)
```

Each element of this list is processed in turn, by evaluating the *val-form*, and binding the *var* to the resulting value. Finally, the *body* forms are evaluated sequentially, the bindings are undone, and the result of the last form is returned. Note that the bindings are sequential, not parallel.

This is a very unusual special form because of the way the evaluator is called on the result of an evaluation. Thus progw is mainly useful for implementing special forms and for functions part of whose contract is that they call the interpreter. For an example of the latter, see sys:*break-bindings* (page 645); break implements this by using progw.

Here are the special forms for defining special variables.

defvar *variable* [*initial-value*] [*documentation*] *Special Form*

defvar is the recommended way to declare the use of a global variable in a program. Placed at top level in a file,

```
(defvar variable initial-value
  "documentation")
```

declares *variable* special for the sake of compilation, and records its location in the file for the sake of the editor so that you can ask to see where the variable is defined. The documentation string is remembered and returned if you do (documentation '*variable*).

variable is initialized to the result of evaluating the form *initial-value* unless it already has a value, in which case it keeps that value. *initial-value* is not evaluated unless it is used; this is useful if it does something expensive like creating a large data structure.

If you do not wish to give *variable* any initial value, use the symbol :unbound as the *initial-value* form. This is treated specially; no attempt is made to evaluate :unbound.

Using a documentation string is better than using a comment to describe the use of the variable, because the documentation string is accessible to system programs that can show the documentation to you while you are using the machine. While it is still permissible to omit *initial-value* and the documentation string, it is recommended that you put a documentation string in every **defvar**.

defvar should be used only at top level, never in function definitions, and only for global variables (those used by more than one function). **(defvar foo 'bar)** is roughly equivalent to

```
(declare (special foo))
(if (not (boundp 'foo))
    (setq foo 'bar))
```

If **defvar** is used in a patch file (see section 25.7, page 531) or is a single form (not a region) evaluated with the editor's compile/evaluate from buffer commands, if there is an initial-value the variable is always set to it regardless of whether it is already bound.

defconst *variable initial-value* [*documentation*] *Special Form*

defconst is the same as **defvar** except that if an initial value is given the variable is always set to it regardless of whether it is already bound. The rationale for this is that **defvar** declares a global variable, whose value is initialized to something but will then be changed by the functions that use it to maintain some state. On the other hand, **defconst** declares a constant, whose value will be changed only by changes *to* the program, never by the operation of the program as written. **defconst** always sets the variable to the specified value so that if, while developing or debugging the program, you change your mind about what the constant value should be, and then you evaluate the **defconst** form again, the variable will get the new value. It is *not* the intent of **defconst** to declare that the value of *variable* will never change; for example, **defconst** is *not* license to the compiler to build assumptions about the value of *variable* into programs being compiled.

As with **defvar**, you should include a documentation string in every **defconst**.

3.2 Functions

In the description of evaluation on page 14, we said that evaluation of a function form works by applying the function to the results of evaluating the argument subforms. What is a function, and what does it mean to apply it? In Zetalisp there are many kinds of functions, and applying them may do many different kinds of things. For full details, see chapter 10, page 154. Here we will explain the most basic kinds of functions and how they work. In particular, this section explains *lambda lists* and all their important features.

The simplest kind of user-defined function is the *lambda-expression*, which is a list that looks like:

(lambda *lambda-list body1 body2...*)

The first element of the lambda-expression is the symbol **lambda**; the second element is a list called the *lambda list*, and the rest of the elements are called the *body*. The lambda list, in its simplest form, is just a list of variables. Assuming that this simple form is being used, here is what happens when a lambda expression is applied to some arguments. First, the number of arguments and the number of variables in the lambda list must be the same, or else an error is signalled. Each variable is bound to the corresponding argument value. Then the forms of the body are evaluated sequentially. After this, the bindings are all undone, and the value of the last form in the body is returned.

This may sound something like the description of **let**, above. The most important difference is that the lambda-expression is not a form at all; if you try to evaluate a lambda-expression, you will get told that **lambda** is not a defined function. The lambda-expression is a *function*, not a form. A **let** form gets evaluated, and the values to which the variables are bound come from the evaluation of some subforms inside the **let** form; a lambda-expression gets applied, and the values are the arguments to which it is applied.

The variables in the lambda list are sometimes called *parameters*, by analogy with other languages. Some other terminologies would refer to these as *formal parameters*, and to arguments as *actual parameters*.

Lambda lists can have more complex structure than simply being a list of variables. There are additional features accessible by using certain keywords (which start with &) and/or lists as elements of the lambda list.

The principal weakness of the simple lambda lists is that any function written with one must only take a certain, fixed number of arguments. As we know, many very useful functions, such as **list**, **append**, **+**, and so on, accept a varying number of arguments. Maclisp solved this problem by the use of *lexprs* and *lsubrs*, which were somewhat inelegant since the parameters had to be referred to by numbers instead of names (e.g. **(arg 3)**). (For compatibility reasons, Zetalisp supports *lexprs*, but they should not be used in new programs.) Simple lambda lists also require that arguments be matched with parameters by their position in the sequence. This makes calls hard to read when there are a great many arguments. Keyword parameters enable the use of other styles of call which are more readable.

In general, a function in Zetalisp has zero or more *positional* parameters, followed if desired by a single *rest* parameter, followed by zero or more *keyword* parameters. The positional and keyword parameters may be *required* or *optional*, but all the optional parameters must follow all

the required ones. The required/optional distinction does not apply to the rest parameter.

The caller must provide enough arguments so that each of the required parameters gets bound, but he may provide extra arguments for some of the optional parameters. Also, if there is a rest parameter, he can provide as many extra arguments as he wants, and the rest parameter will be bound to a list of all these extras. Optional parameters may have a *default-form*, which is a form to be evaluated to produce the default value for the parameter if no argument is supplied.

Positional parameters are matched with arguments by the position of the arguments in the argument list. Keyword parameters are matched with their arguments by matching the keyword name; the arguments need not appear in the same order as the parameters. If an optional positional argument is omitted, then no further arguments can be present. Keyword parameters allow the caller to decide independently for each one whether to specify it.

Here is the exact explanation of how this all works. When apply (the primitive function that applies functions to arguments) matches up the arguments with the parameters, it follows the following algorithm:

Required positional parameters:

> The first required positional parameter is bound to the first argument. apply continues to bind successive required positional parameters to the successive arguments. If, during this process, there are no arguments left but there are still some required parameters (positional or keyword) which have not been bound yet, it is an error ("too few arguments").

Optional positional parameters:

> After all required parameters are handled, apply continues with the optional positional parameters, if any. It binds successive parameter to the next argument. If, during this process, there are no arguments left, each remaining optional parameter's default-form is evaluated, and the parameter is bound to it. This is done one parameter at a time; that is, first one default-form is evaluated, and then the parameter is bound to it, then the next default-form is evaluated, and so on. This allows the default for an argument to depend on the previous argument.

After the positional parameters:

> Now, if there are no remaining parameters (rest or keyword), and there are no remaining arguments, we are finished. If there are no more parameters but there are still some arguments remaining, an error is signaled ("too many arguments"). If parameters remain, all the remaining arguments are used for *both* the rest parameter, if any, and the keyword parameters.

Rest parameter:

> If there is a rest parameter, it is bound to a list of all the remaining arguments. If there are no remaining arguments, it gets bound to nil.

Keyword parameters:

If there are keyword parameters, the same remaining arguments are used to bind them, as follows.

The arguments for the keyword parameters are treated as a list of alternating keyword symbols and associated values. Each symbol is matched with the keyword parameter names, and the matching keyword paramater is bound to the value which follows the symbol. All the remaining arguments are treated in this way. Since the arguments are usually obtained by evaluation, those arguments which are keyword symbols are typically quoted in the call; but they do not have to be. The keyword symbols are compared by means of **eq**, which means they must be specified in the correct package. The keyword symbol for a parameter has the same print name as the parameter, but resides in the keyword package regardless of what package the parameter name itself resides in. (You can specify the keyword symbol explicitly in the lambda list if you must; see below.)

If any keyword parameter has not received a value when all the arguments have been processed, this is an error if the parameter is required. If it is optional, the default-form for the parameter is evaluated and the parameter is bound to its value.

There may be a keyword symbol among the arguments which does not match any keyword parameter name. The function itself specifies whether this is an error. If it is not an error, then the non-matching symbols and their associated values are ignored. The function can access these symbols and values through the rest parameter, if there is one. It is common for a function to check only for certain keywords, and pass its rest parameter to another function using **lexpr-funcall**; that function will check for the keywords that concern it.

The way you express which parameters are required, optional, and rest is by means of specially recognized symbols, which are called *&-keywords*, in the lambda list. All such symbols' print names begin with the character "&". A list of all such symbols is the value of the symbol **lambda-list-keywords**.

The keywords used here are **&key**, **&optional** and **&rest**. The way they are used is best explained by means of examples; the following are typical lambda lists, followed by descriptions of which parameters are positional, rest or keyword; and required or optional.

`(a b c)` **a**, **b**, and **c** are all required and positional. The function must be passed three arguments.

`(a b &optional c)`
a and **b** are required, **c** is optional. All three are positional. The function may be passed either two or three arguments.

`(&optional a b c)`
a, **b**, and **c** are all optional and positional. The function may be passed any number of arguments between zero and three, inclusive.

`(&rest a)` **a** is a rest parameter. The function may be passed any number of arguments.

`(a b &optional c d &rest e)`
a and **b** are required positional, **c** and **d** are optional positional, and **e** is rest. The function may be passed two or more arguments.

`(&key a b)` a and b are both required keyword parameters. A typical call would look like

```
(foo ':b 69 ':a '(some elements))
```

This illustrates that the parameters can be matched in either order. If a keyword is specified twice, the first value is used.

`(&key a &optional b)`

a is required keyword, and b is optional keyword. The sample call above would be legal for this function also; so would

```
(foo ':a '(some elements))
```

which doesn't specify b.

`(x &optional y &rest z &key a b)`

x is required positional, y is optional positional, z is rest, and a and b are optional keyword. One or more arguments are allowed. One or two arguments specify only the positional parameters. Arguments beyond the second specify both the rest parameter and the keyword parameters, so that

```
(foo 1 2 ':b '(a list))
```

specifies 1 for x, 2 for y, (:b (a list)) for z, and (a list) for b. It does not specify a.

`(&rest z &key &optional a b c &allow-other-keys)`

z is rest, and a, b and c are optional keyword parameters. &allow-other-keys says that absolutely any keyword symbols may appear among the arguments; these symbols and the values that follow them have no effect on the keyword parameters, but do become part of the value of z.

`(&rest z &key &allow-other-keys)`

This is equivalent to (&rest z). So, for that matter, is the previous example, if the function does not use the values of a, b and c.

In all of the cases above, the *default-form* for each optional parameter is nil. To specify your own default forms, instead of putting a symbol as the element of a lambda list, put in a list whose first element is the symbol (the parameter itself) and whose second element is the default-form. Only optional parameters may have default forms; required parameters are never defaulted, and rest parameters always default to nil. For example:

`(a &optional (b 3))`

The default-form for b is 3. a is a required parameter, and so it doesn't have a default form.

`(&optional (a 'foo) &rest d &key b (c (symeval a)))`

a's default-form is 'foo, b's is nil, and c's is (symeval a). Note that if the function whose lambda list this is were called on no arguments, a would be bound to the symbol foo, and c would be bound to the value of the symbol foo; this illustrates the fact that each variable is bound immediately after its default-form is evaluated, and so later default-forms may take advantage of earlier parameters in the lambda list. b and d would be bound to nil.

Occasionally it is important to know whether a certain optional parameter was defaulted or not. You can't tell from just examining its value, since if the value is the default value, there's no way to tell whether the caller passed that value explicitly, or whether the caller didn't pass any value and the parameter was defaulted. The way to tell for sure is to put a third element into

the list: the third element should be a variable (a symbol), and that variable is bound to **nil** if the parameter was not passed by the caller (and so was defaulted), or **t** if the parameter was passed. The new variable is called a "supplied-p" variable; it is bound to **t** if the parameter is supplied. For example:

`(a &optional (b 3 c))`
: The default-form for **b** is **3**, and the "supplied-p" variable for **b** is **c**. If the function is called with one argument, **b** will be bound to **3** and **c** will be bound to **nil**. If the function is called with two arguments, **b** will be bound to the value that was passed by the caller (which might be **3**), and **c** will be bound to **t**.

It is possible to specify a keyword parameter's symbol independently of its parameter name. To do this, use *two* nested lists to specify the parameter. The outer list is the one which can contain the default-form and supplied-p variable, if the parameter is optional. The first element of this list, instead of a symbol, is again a list, whose elements are the keyword symbol and the parameter variable name. For example:

`(&key ((:a a)) &optional ((:b b) t))`
: This is equivalent to **(&key a &optional (b t))**.

`(&key ((:base base-value)))`
: This allows a keyword which the user will know under the name **:base**, without making the parameter shadow the value of **base**, which is used for printing numbers.

It is also possible to include, in the lambda list, some other symbols, which are bound to the values of their default-forms upon entry to the function. These are *not* parameters, and they are never bound to arguments; they just get bound, as if they appeared in a **let** form. (Whether you use these aux-variables or bind the variables with **let** is a stylistic decision.)

To include such symbols, put them after any parameters, preceeded by the &-keyword **&aux**. Examples:

`(a &optional b &rest c &aux d (e 5) (f (cons a e)))`
: **d**, **e**, and **f** are bound, when the function is called, to **nil**, **5**, and a cons of the first argument and **5**.

Note that aux-variables are bound sequentially rather than in parallel.

It is important to realize that the list of arguments to which a rest-parameter is bound is set up in whatever way is most efficiently implemented, rather than in the way that is most convenient for the function receiving the arguments. It is not guaranteed to be a "real" list. Sometimes the rest-args list is a stack list (see section 5.8, page 80) stored in the function-calling stack, and loses its validity when the function returns. If a rest-argument is to be returned or made part of permanent list-structure, it must first be copied (see **copylist**, page 66), as you must always assume that it is one of these special lists. The system will not detect the error of omitting to copy a rest-argument; you will simply find that you have a value which seems to change behind your back.

At other times the rest-args list will be an argument that was given to **apply**; therefore it is not safe to **rplaca** this list as you may modify permanent data structure. An attempt to **rplacd** a rest-args list will be unsafe in this case, while in the first case it would cause an error, since lists in the stack are impossible to **rplacd**.

There are some other keywords in addition to those mentioned here. See section 10.7, page 168 for a complete list. You need to know only about **&optional**, **&key**, and **&rest** in order to understand this manual.

3.3 Some Functions and Special Forms

This section describes some functions and special forms. Some are parts of the evaluator, or closely related to it. Some have to do specifically with issues discussed above such as keyword arguments. Some are just fundamental Lisp forms that are very important.

eval *x*

(eval *x*) evaluates *x*, and returns the result.
Example:

```
(setq x 43 foo 'bar)
(eval (list 'cons x 'foo))
    => (43 . bar)
```

It is unusual to explicitly call **eval**, since usually evaluation is done implicitly. If you are writing a simple Lisp program and explicitly calling **eval**, you are probably doing something wrong. **eval** is primarily useful in programs which deal with Lisp itself, rather than programs about knowledge or mathematics or games.

Also, if you are only interested in getting at the value of a symbol (that is, the contents of the symbol's value cell), then you should use the primitive function **symeval** (see page 96).

Note: the actual name of the compiled code for **eval** is **"si:*eval"**; this is because use of the *evalhook* feature binds the function cell of **eval**. If you don't understand this, you can safely ignore it.

Note: unlike Maclisp, **eval** never takes a second argument; there are no "binding context pointers" in Zetalisp. They are replaced by Closures (see chapter 11, page 180).

apply *f arglist*

(**apply** *f arglist*) applies the function *f* to the list of arguments *arglist*. *arglist* should be a list; *f* can be any function.
Examples:

```
(setq fred '+) (apply fred '(1 2)) => 3
(setq fred '-) (apply fred '(1 2)) => -1
(apply 'cons '((+ 2 3) 4)) =>
        ((+ 2 3) . 4)   not (5 . 4)
```

Of course, *arglist* may be **nil**.

Note: unlike Maclisp, **apply** never takes a third argument; there are no "binding context pointers" in Zetalisp.

Compare **apply** with **funcall** and **eval**.

funcall *f* &rest *args*

(**funcall** *f a1 a2 ... an*) applies the function *f* to the arguments *a1*, *a2*, ..., *an*. *f* may not be a special form nor a macro; this would not be meaningful.

Example:

```
(cons 1 2) => (1 . 2)
(setq cons 'plus)
(funcall cons 1 2) => 3
```

This shows that the use of the symbol **cons** as the name of a function and the use of that symbol as the name of a variable do not interact. The **cons** form invokes the function named **cons**. The **funcall** form evaluates the variable and gets the symbol **plus**, which is the name of a different function.

lexpr-funcall *f* &rest *args*

lexpr-funcall is like a cross between **apply** and **funcall**. (**lexpr-funcall** *f a1 a2 ... an l*) applies the function *f* to the arguments *a1* through *an* followed by the elements of the list *l*.

Examples:

```
(lexpr-funcall 'plus 1 1 1 '(1 1 1)) => 6

(lexpr-funcall 'plus '(1 2)) => 3

(lexpr-funcall '(car (a))) => a
      ;Not the same as (eval '(car (a)))

(defun report-error (&rest args)
   (lexpr-funcall (function format) error-output args))
```

lexpr-funcall with two arguments does the same thing as **apply**. **lexpr-funcall** with only one argument treats it as a list of a function and some arguments.

Note: the Maclisp functions **subrcall**, **lsubrcall**, and **arraycall** are not needed on the Lisp Machine; **funcall** is just as efficient. **arraycall** is provided for compatibility; it ignores its first subform (the Maclisp array type) and is otherwise identical to **aref**. **subrcall** and **lsubrcall** are not provided.

call *function* &rest *argument-specifications*

call offers a very general way of controlling what arguments you pass to a function. You can provide either individual arguments a la **funcall** or lists of arguments a la **apply**, in any order. In addition, you can make some of the arguments *optional*. If the function is not prepared to accept all the arguments you specify, no error occurs if the excess arguments are optional ones. Instead, the excess arguments are simply not passed to the function.

The *argument-specs* are alternating keywords (or lists of keywords) and values. Each keyword or list of keywords says what to do with the value that follows. If a value happens to require no keywords, provide () as a list of keywords for it.

Two keywords are presently defined: :optional and :spread. :spread says that the following value is a list of arguments. Otherwise it is a single argument. :optional says that all the following arguments are optional. It is not necessary to specify :optional with all the following *argument-specs*, because it is sticky.

Example:

```
(call #'foo () x ':spread y '(:optional :spread) z () w)
```

The arguments passed to foo are the value of x, the elements of the value of y, the elements of the value of z, and the value of w. The function foo must be prepared to accept all the arguments which come from x and y, but if it does not want the rest, they are ignored.

quote *object* *Special Form*

(quote *x*) simply returns *x*. It is useful specifically because *x* is not evaluated; the quote is how you make a form that returns an arbitrary Lisp object. quote is used to include constants in a form.

Examples:

```
(quote x) => x
(setq x (quote (some list)))    x => (some list)
```

Since quote is so useful but somewhat cumbersome to type, the reader normally converts any form preceded by a single quote (') character into a quote form.

For example,

```
(setq x '(some list))
```

is converted by read into

```
(setq x (quote (some list)))
```

function *f* *Special Form*

This means different things depending on whether *f* is a function or the name of a function. (Note that in neither case is *f* evaluated.) The name of a function is a symbol or a function-spec list (see section 10.2, page 154). A function is typically a list whose car is the symbol lambda; however, there are several other kinds of functions available (see section 10.5, page 160).

If you want to pass an anonymous function as an argument to a function, you could just use quote; for example:

```
(mapc (quote (lambda (x) (car x))) some-list)
```

This works fine as far as the evaluator is concerned. However, the compiler cannot tell that the first argument is going to be used as a function; for all it knows, mapc will treat its first argument as a piece of list structure, asking for its car and cdr and so forth. So the compiler cannot compile the function; it must pass the lambda-expression unmodified. This means that the function will not get compiled, which will make it execute more slowly than it might otherwise.

The function special form is one way to tell the compiler that it can go ahead and compile the lambda-expression. You just use the symbol function instead of quote:

```
(mapc (function (lambda (x) (car x))) some-list)
```

This will cause the compiler to generate code such that mapc will be passed a compiled-code object as its first argument.

That's what the compiler does with a function special form whose subform *f* is a function. The evaluator, when given such a form, just returns *f*; that is, it treats function just like quote.

To ease typing, the reader converts *#'thing* into (function *thing*). So *#'* is similar to ' except that it produces a function form instead of a quote form. So the above form could be written as

```
(mapc #'(lambda (x) (car x)) some-list)
```

If *f* is not a function but the name of a function (typically a symbol, but in general any kind of function spec), then function returns the function definition of *f*; it is like fdefinition except that it is a special form instead of a function, and so

```
(function fred)  is like  (fdefinition 'fred)
                 which is like (fsymeval 'fred)
```

since fred is a symbol. function is the same for the compiler and the interpreter when *f* is the name of a function.

Another way of explaining function is that it causes *f* to be treated the same way as it would as the car of a form. Evaluating the form (*f arg1 arg2...*) uses the function definition of *f* if it is a symbol, and otherwise expects *f* to be a list which is a lambda-expression. Note that the car of a form may not be a non-symbol function spec, to avoid difficult-to-read code. This can be written as

```
(funcall (function spec) args...)
```

You should be careful about whether you use *#'* or '. Suppose you have a program with a variable x whose value is assumed to contain a function that gets called on some arguments. If you want that variable to be the car function, there are two things you could say:

```
(setq x 'car)
```

or

```
(setq x #'car)
```

The former causes the value of x to be the symbol car, whereas the latter causes the value of x to be the function object found in the function cell of car. When the time comes to call the function (the program does (funcall x ...)), either of these two will work (because if you use a symbol as a function, the contents of the symbol's function cell is used as the function, as explained in the beginning of this chapter). Using 'car is insignificantly slower, because the function call has to indirect through the symbol, but it allows the function to be redefined, traced (see page 588), or advised (see page 593). The latter case, while faster, picks up the function definition out of the symbol car and does not see any later changes to it.

The other way to tell the compiler that an argument that is a lambda expression should be compiled is for the function that takes the function as an argument to use the &functional keyword in its lambda list; see section 10.7, page 168. The basic system functions that take functions as arguments, such as map and sort, have this &functional keyword and hence quoted lambda-expressions given to them will be recognized as functions by the compiler.

In fact, mapc uses &functional and so the example given above is bogus; in the particular case of the first argument to the function mapc, quote and function are synonymous. It is good style to use function (or #') anyway, to make the intent of the program completely clear.

false

Takes no arguments and returns nil.

true

Takes no arguments and returns t.

ignore &rest *ignore*

Takes any number of arguments and returns nil. This is often useful as a "dummy" function; if you are calling a function that takes a function as an argument, and you want to pass one that doesn't do anything and won't mind being called with any argument pattern, use this.

comment *Special Form*

comment ignores its form and returns the symbol comment.
Example:

```
(defun foo (x)
    (cond ((null x) 0)
          (t (comment x has something in it)
             (1+ (foo (cdr x))))))
```

Usually it is preferable to comment code using the semicolon-macro feature of the standard input syntax. This allows the user to add comments to his code which are ignored by the Lisp reader.
Example:

```
(defun foo (x)
    (cond ((null x) 0)
          (t (1+ (foo (cdr x))))      ;x has something in it
      ))
```

A problem with such comments is that they are discarded when the form is read into Lisp. If the function is read into Lisp, modified, and printed out again, the comment will be lost. However, this style of operation is hardly ever used; usually the source of a function is kept in an editor buffer and any changes are made to the buffer, rather than the actual list structure of the function. Thus, this is not a real problem.

progn *body...* *Special Form*

The *body* forms are evaluated in order from left to right and the value of the last one is returned. **progn** is the primitive control structure construct for "compound statements". Although lambda-expressions, **cond** forms, **do** forms, and many other control structure forms use **progn** implicitly, that is, they allow multiple forms in their bodies, there are occasions when one needs to evaluate a number of forms for their side-effects and make them appear to be a single form.

Example:

```
(foo (cdr a)
     (progn (setq b (extract frob))
            (car b))
     (cadr b))
```

(When *form1* is 'compile, the **progn** form has a special meaning to the compiler. This is discussed in section 17.4.3, page 260.)

prog1 *first-form body...* *Special Form*

prog1 is similar to **progn**, but it returns the value of its *first* form rather than its last. It is most commonly used to evaluate an expression with side effects, and return a value which must be computed *before* the side effects happen.

Example:

```
(setq x (prog1 y (setq y x)))
```

interchanges the values of the variables x and y. **prog1** never returns multiple values.

prog2 *first-form second-form body...* *Special Form*

prog2 is similar to **progn** and **prog1**, but it returns its *second* form. It is included largely for compatibility with old programs.

See also **bind** (page 212), which is a subprimitive that gives you maximal control over binding.

The following three functions (**arg**, **setarg**, and **listify**) exist only for compatibility with Maclisp *lexprs*. To write functions that can accept variable numbers of arguments, use the **&optional** and **&rest** keywords (see section 3.2, page 21).

arg *x*

(arg nil), when evaluated during the application of a lexpr, gives the number of arguments supplied to that lexpr. This is primarily a debugging aid, since lexprs also receive their number of arguments as the value of their **lambda**-variable.

(arg *i*), when evaluated during the application of a lexpr, gives the value of the *i*'th argument to the lexpr. *i* must be a fixnum in this case. It is an error if *i* is less than 1 or greater than the number of arguments supplied to the lexpr.

Example:

```
(defun foo nargs                ;define a lexpr foo.
    (print (arg 2))             ;print the second argument.
    (+ (arg 1)                  ;return the sum of the first
       (arg (- nargs 1))))      ;and next to last arguments.
```

setarg *i x*

setarg is used only during the application of a lexpr. (setarg *i x*) sets the lexpr's *i*'th argument to *x*. *i* must be greater than zero and not greater than the number of arguments passed to the lexpr. After (setarg *i x*) has been done, (arg *i*) will return *x*.

listify *n*

(listify *n*) manufactures a list of *n* of the arguments of a lexpr. With a positive argument *n*, it returns a list of the first *n* arguments of the lexpr. With a negative argument *n*, it returns a list of the last (abs *n*) arguments of the lexpr. Basically, it works as if defined as follows:

```
(defun listify (n)
     (cond ((minusp n)
            (listify1 (arg nil) (+ (arg nil) n 1)))
           (t
            (listify1 n 1)) ))

(defun listify1 (n m)        ; auxiliary function.
     (do ((i n (1- i))
          (result nil (cons (arg i) result)))
         ((< i m) result) ))
```

3.4 Tail Recursion

When one function ends by calling another function (possibly itself), as in

```
(defun last (x)
  (cond ((atom x) x)
        ((atom (cdr x)) x)
        (t (last (cdr x)))))
```

it is called *tail recursion*. In general, if *X* is a form, and *Y* is a sub-form of *X*, then if the value of *Y* is unconditionally returned as the value of *X*, with no intervening computation, then we say that *X* tail-recursively evaluates *Y*.

In a tail-recursive situation, it is not strictly necessary to remember anything about the first call to last when the second one is activated. The stack frame for the first call can be discarded completely, allowing last to use a bounded amount of stack space independent of the length of its argument. A system which does this is called *tail-recursive*.

The Lisp machine system works tail recursively if the variable tail-recursion-flag is non-nil. This is often faster, because it reduces the amount of time spent in refilling the hardware's pdl buffer. Also, it may cause a program to run in an ordinary-size stack instead of periodically exhausting its stack. However, you forfeit a certain amount of useful debugging information: once the outer call to last has been removed from the stack, you can no longer see its frame in the debugger.

tail-recursion-flag *Variable*

If this variable is non-nil, the calling stack frame is discarded when a tail-recursive call is made in compiled code.

There are many things which can make it dangerous to discard the outer stack frame. For example, it may have done a ***catch**; it may have bound special variables; it may have a **&rest** argument on the stack; it may have asked for the location of an argument or local variable. The system detects all of these conditions automatically and retains the outer stack frame to bring about proper execution. Some of these conditions occur in **eval**; as a result, interpreted code is never completely tail recursive.

3.5 Multiple Values

The Lisp Machine includes a facility by which the evaluation of a form can produce more than one value. When a function needs to return more than one result to its caller, multiple values are a cleaner way of doing this than returning a list of the values or **setq**'ing special variables to the extra values. In most Lisp function calls, multiple values are not used. Special syntax is required both to *produce* multiple values and to *receive* them.

The primitive for producing multiple values is **values**, which takes any number of arguments and returns that many values. If the last form in the body of a function is a **values** with three arguments, then a call to that function will return three values. The other primitive for producing multiple values is **return**, which when given more than one argument returns all its arguments as the values of the **prog** or **do** from which it is returning. The variant **return-from** also can produce multiple values. Many system functions produce multiple values, but they all do it via the **values** and **return** primitives.

The special forms for receiving multiple values are **multiple-value**, **multiple-value-bind**, and **multiple-value-list**. These consist of a form and an indication of where to put the values returned by that form. With the first two of these, the caller requests a certain number of returned values. If fewer values are returned than the number requested, then it is exactly as if the rest of the values were present and had the value **nil**. If too many values are returned, the rest of the values are ignored. This has the advantage that you don't have to pay attention to extra values if you don't care about them, but it does no error checking on the number of values actually returned.

values &rest *args*

Returns multiple values, its arguments. This is the primitive function for producing multiple values. It is legal to call **values** with no arguments; it returns no values in that case.

values-list *list*

Returns multiple values, the elements of the *list*. **(values-list '(a b c))** is the same as **(values 'a 'b 'c)**. *list* may be **nil**, the empty list, which causes no values to be returned.

return and its variants can only be used within the **do** and **prog** special forms and their variants, and so they are explained on page 52.

multiple-value (*variable...*) *form* *Special Form*

multiple-value is a special form used for calling a function which is expected to return more than one value. *form* is evaluated, and the *variables* are *set* (not lambda-bound) to the values returned by *form*. If more values are returned than there are variables, the extra values are ignored. If there are more variables than values returned, extra values of nil are supplied. If nil appears in the *var-list*, then the corresponding value is ignored (you can't use nil as a variable).

Example:

```
(multiple-value (symbol already-there-p)
        (intern "goo"))
```

In addition to its first value (the symbol), intern returns a second value, which is t if the symbol returned as the first value was already interned, or else nil if intern had to create it. So if the symbol goo was already known, the variable already-there-p will be set to t, otherwise it will be set to nil. The third value returned by intern will be ignored.

multiple-value is usually used for effect rather than for value; however, its value is defined to be the first of the values returned by *form*.

multiple-value-bind (*variable...*) *form body...* *Special Form*

This is similar to multiple-value, but locally binds the variables which receive the values, rather than setting them, and has a body—a set of forms which are evaluated with these local bindings in effect. First *form* is evaluated. Then the *variables* are bound to the values returned by *form*. Then the *body* forms are evaluated sequentially, the bindings are undone, and the result of the last *body* form is returned.

Example:

```
(multiple-value-bind (sym already-there)
    (intern string)
  ;; If an existing symbol was found, deallocate the string.
  (if already-there
      (return-storage (prog1 string (setq string nil))))
  sym)
```

multiple-value-list *form* *Special Form*

multiple-value-list evaluates *form*, and returns a list of the values it returned. This is useful for when you don't know how many values to expect.

Example:

```
(setq a (multiple-value-list (intern "goo")))
a => (goo nil #<Package USER 10112147>)
```

This is similar to the example of multiple-value above; a will be set to a list of three elements, the three values returned by intern.

When one form finished by tail recursively evaluating a subform (see section 3.4, page 32), all of the subform's multiple values are passed back by the outer form. For example, the value of a cond is the value of the last form in the selected clause. If the last form in that clause produces multiple values, so does the cond. This *passing-back* of multiple values of course has no effect unless eventually one of the special forms for receiving multiple values is reached.

If the outer form returns a value computed by a subform, but not in a tail recursive fashion (for example, if the value of the subform is examined first), multiple values or only single values may be returned at the discretion of the implementation; users should not depend on whatever way it happens to work, as it may change in the future or in other implementations. The reason we don't guarantee non-transmission of multiple values is because such a guarantee would not be very useful and the efficiency cost of enforcing it would be high. Even **setq**'ing a variable to the result of a form, then returning the value of that variable, might pass multiple values if an optimizing compiler realized that the **setq**ing of the variable was unnecessary. Since extra returned values are generally ignored, it is not vital to eliminate them.

Note that use of a form as an argument to a function never receives multiple values from that form. That is, if the form **(foo (bar))** is evaluated and the call to **bar** returns many values, **foo** will still only be called on one argument (namely, the first value returned), rather than being called on all the values returned. We choose not to generate several separate arguments from the several values, because this would make the source code obscure; it would not be syntactically obvious that a single form does not correspond to a single argument. Instead, the first value of a form is used as the argument and the remaining values are discarded. Receiving of multiple values is done only with the above-mentioned special forms.

For clarity, descriptions of the interaction of several common special forms with multiple values follow. This can all be deduced from the rule given above. Note well that when it says that multiple values are not returned, it really means that they may or may not be returned, and you should not write any programs that depend on which way it works.

The body of a **defun** or a **lambda**, and variations such as the body of a function, the body of a **let**, etc., pass back multiple values from the last form in the body.

eval, **apply**, **funcall**, and **lexpr-funcall** pass back multiple values from the function called.

progn passes back multiple values from its last form. **progv** and **progw** do so also. **prog1** and **prog2**, however, do not pass back multiple values.

Multiple values are passed back from the last subform of an **and** or **or** form, but not from previous forms since the return is conditional. Remember that multiple values are only passed back when the value of a sub-form is unconditionally returned from the containing form. For example, consider the form **(or (foo) (bar))**. If **foo** returns a non-**nil** first value, then only that value will be returned as the value of the form. But if it returns **nil** (as its first value), then **or** returns whatever values the call to **bar** returns.

cond passes back multiple values from the last form in the selected clause, provided that that last form's value will be returned unconditionally. This is true if the clause has two or more forms in it, and is always true for the last clause.

The variants of **cond** such as **if**, **select**, **selectq**, and **dispatch** pass back multiple values from the last form in the selected clause.

The number of values returned by **prog** depends on the **return** form used to return from the **prog**. (If a **prog** drops off the end it just returns a single **nil**.) If **return** is given two or more subforms, then **prog** will return as many values as the **return** has subforms. However, if the

return has only one subform, then the prog will return all of the values returned by that one subform.

do behaves like prog with respect to return. All the values of the last *exit-form* are returned.

unwind-protect passes back multiple values from its protected form. In a sense, this is an exception to the rule; but it is useful, and it makes sense to consider the execution of the unwind forms as a byproduct of unwinding the stack and not as part of sequential execution.

*catch does not pass back multiple values from the last form in its body, because it is defined to return its own second value (see page 54) to tell you whether the *catch form was exited normally or abnormally. To do a catch and propagate multiple values, use catch-continuation.

3.6 Evaluator Errors

Here is a description of the error conditions that the evaluator can signal. This is for use by those who are writing condition handlers (section 27.2, page 554). The novice should skip this section.

sys:invalid-form (error) *Condition*

This is signaled when eval's argument is not a recognizable kind of form; the wrong data type, perhaps. The condition instance supports the operation :form, which returns the supposed form to be evaluated.

sys:invalid-function (error) *Condition*

This is signaled when eval or apply finds an object that is supposed to be applied to arguments, but it is not a valid Lisp function. The condition instance supports the operation :function, which returns the supposed function to be called.

The :new-function proceed type is provided; it expects one argument, a function to call instead.

sys:invalid-lambda-list (sys:invalid-function error) *Condition*

This condition name is present in addition to sys:invalid-function when the function to be called looks like an interpreted function, and the only problem is the syntax of its lambda list.

sys:too-few-arguments (error) *Condition*

This condition is signaled when a function is applied to too few arguments. The condition instance supports the operations :function and :arguments which return the function and the list of the arguments provided.

The proceed types :additional-arguments and :new-argument-list are provided. Both take one argument. In the first case, the argument is a list of arguments to pass in addition to the ones supplied. In the second, it is a list of arguments to replace the ones actually supplied.

sys:too-many-arguments (error) *Condition*

This is similar to sys:too-few-arguments. Instead of the :additional-arguments proceed type, :fewer-arguments is provided. Its argument is a number, which is how many of the originally supplied arguments to use in calling the function again.

sys:missing-keyword-argument (error) *Condition*

This is signaled when a required keyword argument is missing. The :keyword operation on the condition instance returns the missing keyword.

The proceed type :argument-value is provided. It expects one argument, which is a value to use for the keyword argument.

sys:undefined-keyword-argument (error) *Condition*

This is signaled when a function that takes keyword arguments is given a keyword that it does not accept and &allow-other-keys was not used. The :keyword operation on the condition instance returns the extraneous keyword, and the :value operation returns the value supplied with it.

The proceed type :new-keyword is provided. It expects one argument, which is a keyword to use instead of the one supplied.

sys:cell-contents-error (error) *Condition Flavor*

This condition name categorizes all the errors signaled because of "undefined" objects found in memory. It includes "unbound" variables, "undefined" functions, and other things.

:address A locative pointer to the referenced cell.

:current-address
A locative pointer to the cell which currently contains the contents that are found in the referenced cell when the erring stack group is running. This can be different from the original address in the case of special variable bindings, which move between special PDLs and symbol value cells.

:cell-type A keyword saying what type of cell was referred to: :function, :value, :closure, or nil for a cell that is not one of those.

:containing-structure
The object (list, array, symbol) inside which the referenced memory cell is found.

:data-type
:pointer The data type and pointer fields of the contents of the memory cell, at the time of the error. Both are fixnums.

The proceed type :no-action takes no argument. If the cell's contents are now valid, the program proceeds, using them. Otherwise the error happens again.

The proceed type :package-dwim looks for symbols with the same name in other packages; but only if the containing structure is a symbol.

Two other proceed types take one argument: :new-value and :store-new-value. The argument is used as the contents of the memory cell. :store-new-value also permanently stores the argument into the cell.

sys:unbound-variable (sys:cell-contents-error error) *Condition*

This condition name categorizes all errors of variables which are unbound.

sys:unbound-special-variable *Condition*
sys:unbound-closure-variable *Condition*
sys:unbound-instance-variable *Condition*

These condition names appear in addition to sys:unbound-variable to subcategorize the kind of variable reference that the error happened in.

sys:undefined-function (sys:cell-contents-error error) *Condition*

This condition name categorizes errors of function specs that are undefined.

sys:wrong-type-argument (error) *Condition*

This is signaled when a function checks the type of its argument and rejects it; for example, if you do **(car 1)**.

The condition instance supports these extra operations:

:arg-name — The name of the argument that was erroneous. This may be **nil** if there is no name, or if the system no longer remembers which argument it was.

:old-value — The value that was supplied for the argument.

:function — The function which received and rejected the argument.

:description — A symbol which says what sort of object was expected for this argument.

The proceed type :argument-value is provided; it expects one argument, which is a value to use instead of the erroneous value.

sys:throw-tag-not-seen (error) *Condition*

This is signaled when ***throw** (or ***unwind-stack**) is used and there is no ***catch** for the specified tag. The condition instance supports these extra operations:

:tag — The tag being thrown to.

:value — The value being thrown (the second argument to ***throw**).

:count
:action — The additional two arguments given to ***unwind-stack**, if that was used.

The error occurs in the environment of the ***throw**; no unwinding has yet taken place.

The proceed type :new-tag expects one argument, a tag to throw to instead.