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 self-evaluating symbol (nil, t or a keyword such as :foo), then *form* itself is the result.

If *form* is any other symbol, the result is the value of *form*, considered as a variable. If *form*'s value is void, an error is signaled. The way symbols are bound to values is explained in section 3.1, page 25 below.

If *form* is not any of the above types, and is not a list, *form* itself is the result.

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*. If the car is an explicit function such as a list starting with lambda, the form is a function form. If it is a symbol, things depend on the symbol's function definition, which may be a special form definition (see page 233), a macro definition, or an ordinary function.

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 233, but this hardly ever affects you.

If *form* is a macro form, then the macro is expanded as explained in chapter 18.

If *form* is a function form, it calls for the *application* of a function to *arguments*. The car of *form* is a function or the name of a function. The cdr of *form* is a list of subforms. The subforms are evaluated, sequentially, and each produces one argument for 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 25. A basic explanation of functions is in section 3.3, page 38. The way functions can return more than one value is explained in section 3.7, page 55. The description of all of the kinds of functions, and the means by which they are manipulated, is in chapter 11. Macros are explained in chapter 18. The evalhook facility, which lets you do something arbitrary whenever the evaluator is invoked, is explained in section 30.12, page 748. 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 a symbol is evaluated, its value as a variable is taken.

3.1.1 Variables and Bindings

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*). We say that a variable is *bound* (past participle of active verb) by the action of binding; we also say that the variable is *bound* (state of being) after a binding has been made. When a binding is made, the variable's old binding and value are hidden or *shadowed* by a new binding, which holds a new value. Setting a variable places a new value into the current binding; it does not change which binding is current. In addition, shadowed bindings' values are not affected by setting the variable. Binding a variable does not affect the value in the old current binding but that binding ceases to be current so the value no longer applies.

The action of binding is always followed eventually by the action of unbinding. This discards the current binding of the variable, with its value. The previous binding becomes current again, and the value in it—unchanged since the newer binding was made, in normal operation—is visible again.

Binding is normally done on entry to a function and by certain special forms (**let**, **do**, **prog** and others). The bindings are unbound on exit from the function or the special form, even nonlocal exit such as **go**, **return** or **throw**. The function or special form is said to be the *scope* of the bindings made therein.

Here is a simple example of making a binding, shadowing it, unshadowing it, examining it, and unbinding it. The inner, shadowing binding is made, examined, set, examined and unbound.

```
(let ((a 5))
  (print a)          ;prints 5
  (let ((a "foo"))
    (print a)        ;prints "foo"
    (setq a "bar")
    (print a))       ;prints "bar"
  (print a))         ;prints 5
```

Every symbol has one binding which was never made and is never unbound. This is the *global* binding. This binding is current whenever no other binding has been established that would shadow it. If you type (**setq** x 5) in the Lisp listen loop, you set the global binding of **x**. Programs often set global bindings permanently using **defvar** or one of its cousins (page 33). **setq-globally** and related functions can be used to set or refer to the global binding even when it is shadowed (page 35).

```
(defvar a 5)      ;sets the global binding

(let ((a t))
  (print a))      ;prints t

a => 5            ;the global binding is visible again
```

A binding does not need to have an actual value. It can be *void* instead. The variable is also called void. Actually, a void binding contains a weird internal value, which the system interprets as meaning "there is no value here". (This is the data type code dtp-null, page 271). Reference to a variable whose current binding is void signals an error. In fact, nearly all variables' global bindings are void; only those that you or the system have set are not void. variable-makunbound makes the current binding of a variable void again (page 31).

'Void' used to be called 'unbound', and most function names, error messages and documentation still use the term 'unbound'. The variable is also called 'unbound'. The term 'void' is being adopted because it is less ambiguous. 'Unbound' can mean 'void', or 'not bound' (no binding established), or the past participle of 'unbind'.

All bindings except global binding have a limited scope: one function or special form. This does not fully specify the scope, however: it may be *lexical* or *dynamic*. When a binding has lexical scope, it is visible only from code written within the function or special form that established it. Subroutines called from within the scope, but which are written elsewhere, never see the lexical binding. By contrast, a dynamic binding is visible the whole time it exists (except when it is shadowed, of course), which includes time spent in subroutines called from within the binding construct. The global binding of a symbol can be regarded as a dynamic binding that lasts from the beginning of the session to the end of the session.

Lexical and dynamic bindings are made by the same kinds of function definitions and special forms. By default, the bindings are lexical. You request a dynamic binding instead using a *special-declaration* at the beginning of the body of the function definition or special form. Also, some symbols are marked *globally special*; every binding of such a symbol is dynamic. This is what defvar, etc., do to a symbol. Dynamic bindings are also called *special bindings*, and the variable bound is called a *special variable*. Each use of a symbol as a variable (this includes setting as well as examining) is also marked as lexical or dynamic by the same declarations. A dynamic use sees only dynamic bindings, and a lexical use sees only lexical bindings.

In the examples above it makes no difference whether the bindings of a are lexical or dynamic, because all the code executed between the binding and unbinding is also written lexically within the let which made the binding. Here is an example where it makes a difference:

```
(defun foo ()
  (print a))

(let ((a 5))
  (foo))

>>Error: the variable A is used free but not special.
```

If the intention is that 5 be printed, a dynamic binding is required. A dynamic binding would remain visible for all the execution from the entry to the let to the exit from the let, including the execution of the definition of foo. Actually, the default is to do lexical binding. Since the binding of a is lexical, it is visible only for the evaluation of expressions written inside the let, which does not include the body of foo. In fact, an error happens when foo evaluates a, since a there is supposed to be lexical and no lexical binding is visible. If you compile foo, you get a compiler warning about a.

The use of a inside foo, not lexically within any binding of a, is called *free*, and a is called a *free variable* of foo. Free variables are erroneous unless they are special. Strictly speaking, it is erroneous to type (setq x 5) at top level in the Lisp listener if x has not been made globally special, but this is permitted as an exception because it is so often useful.

One way to make the example work is to make a globally special:

```
(defvar a)

(defun foo () (print a))

(let ((a 5))
  (foo))
```

prints 5. The global specialness of a tells let to make a dynamic binding and tells the evaluation of a in foo to look for one.

Another way is with declarations at the point of binding and the point of use:

```
(defun foo ()
  (declare (special a))
  (print a))

(let ((a 5))
  (declare (special a))
  (foo))
```

A declaration at the point of binding affects only that binding, not other bindings made within it to shadow it. Another way of stating this is that a binding is affected only by a declaration in the construct that makes the binding, not by declarations in surrounding constructs. Thus,

```
(let ((a 5))                    ;this binding is dynamic
  (declare (special a))
  (let ((a "foo"))              ;this binding is lexical
    no declaration here
    ... a ...                   ;this reference is lexical since
    ...                         ; the innermost binding is lexical
    (let ()
      (declare (special a))
      ... a ...                 ;this reference is dynamic, and sees value 5
    ...))
```

[Currently, for historical compatibility, bindings *are* affected by surrounding declarations. However, whenever this makes a difference, the compiler prints a warning to inform the programmer that the declaration should be moved.]

The classical case where dynamic binding is useful is for parameter variables like ***read-base***:

```
(let ((*read-base* 16.))
  (read))
```

reads an expression using hexadecimal numbers by default. ***read-base*** is globally special, and the subroutine of **read** that reads integers uses ***read-base*** free.

Here is an example where lexical bindings are desirable:

```
(let ((a nil))
  (mapatoms (function (lambda (symbol) (push symbol a))))
  a)
```

Because the reference to **a** from within the internal function is lexical, the only binding it can see is the one made by this **let**. **mapatoms** cannot interfere by binding **a** itself. Consider: if **mapatoms** makes a lexical binding of **a**, it is not visible here because this code is not written inside the definition of **mapatoms**. If **mapatoms** makes a dynamic binding of **a**, it is not visible here because the reference to **a** is not declared special and therefore sees only lexical bindings.

The fact that **function** is used to mark the internal function is crucial. It causes the lexical environment appropriate for the function to be combined with the code for the function in a *lexical closure*, which is passed to **mapatoms**.

The last example shows *downward* use of lexical closures. *Upward* use is also possible, in which a function is closed inside a lexical environment and then preserved after the binding construct has been exited.

```
(defun mycons (a d)
  (function (lambda (x)
              (cond ((eq x 'car) a)
                    ((eq x 'cdr) d)))))

(defun mycar (x) (funcall x 'car))
(defun mycdr (x) (funcall x 'cdr))

(setq mc (mycons 4 t))

(mycar mc)  =>  4
(mycdr mc)  =>  t
```

mycons returns an object that can be called as a function with one argument. This object retains a pointer to a lexical environment that has a binding for a and a binding for d. The function mycons that made those bindings has been exited, but this is irrelevant because the bindings were not dynamic. Since the code of the lambda-expression is lexically within the body of mycons, that function can see the lexical bindings made by mycons no matter when it is called. The function returned by mycons records two values and can deliver either of them when asked, and is therefore analogous to a cons cell.

Only lexical bindings are transferred automatically downward and upward, but dynamic bindings can be used in the same ways if explicitly requested through the use of the function closure. See chapter 12, page 250 for more information.

Dynamic bindings, including the global binding, are stored (unless shadowed) in a particular place: the symbol's *value cell*. This is a word at a fixed offset in the symbol itself. When a new dynamic binding is made, the value in the value cell is saved away on a stack called the *special pdl*. The new binding's value is placed in the value cell. When the new binding is unbound, the old binding's value is copied off of the special pdl, into the value cell again. The function symeval examines the value cell of a symbol chosen at run time; therefore, it sees the current dynamic binding of the symbol.

Lexical bindings are never stored in the symbol's value cell. The compiler stores them in fixed slots in stack frames. The interpreter stores them in alists that live in the stack. It should be noted that if the lexical binding is made by compiled code, then all code that ought to see the binding is necessarily also compiled; if the binding is made by interpreted code, then all code that ought to see the binding is necessarily interpreted. Therefore, it is safe for the compiler and interpreter to use completely different techniques for recording lexical bindings.

Lexical binding is the default because the compiler can find with certainty all the places where a lexical binding is used, and usually can use short cuts based on this certainty. For dynamic bindings slow but general code must always be generated.

3.1.2 Setting Variables

Here are the constructs 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*}... *Macro*

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
```

variable-location *symbol* *Special form*

Returns a locative to the cell in which the value of *symbol* is stored. *symbol* is an unevaluated argument, so the name of the symbol must appear explicitly in the code.

For a special variable, this is equivalent to

```
(value-cell-location 'symbol)
```

For a lexical variable, the place where the value is stored is a matter decided by the interpreter or the compiler, but in any case variable-location nevertheless returns a pointer to it.

In addition, if *symbol* is a special variable that is closed over, the value returned is an external value cell, the same as the value of locate-in-closure applied to the proper closure and *symbol*. This cell *always* contains the closure binding's value, which is *current* only inside the closure. See page 251.

variable-boundp *symbol* *Special form*

t if variable *symbol* is not void. It is equivalent to

```
(location-boundp (variable-location symbol))
```

symbol is not evaluated.

variable-makunbound *symbol* *Special form*

Makes *symbol*'s current binding void. It is equivalent to

```
(location-makunbound (variable-location symbol))
```

symbol is not evaluated.

3.1.3 Variable Binding Constructs

Here are the constructs used for binding variables.

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

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 let-if special form, typically written as

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

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.

The bindings are always dynamic, and it is the user's responsibility to put in appropriate declarations so that the body forms consider the variables dynamic.

let-globally ((*var value*)...) *body*... *Macro*

let-globally-if *condition* ((*var value*)...) *body*... *Macro*

let-globally is similar in form to let (see page 31). 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 82) to set them back. The important consequence is that, with let-globally, when the current stack group (see chapter 13, page 256) 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 in the current stack group. Therefore, let-globally can be used for communication between stack groups and between processes.

let-globally-if modifies and restores the variables only if the value of *condition* is non-nil. The *body* is executed in any case.

Since let-globally is based on setq, it makes sense for both lexical and dynamic variables. But its main application exists only for dynamic variables.

The globally in let-globally does not mean the same thing as the globally in setq-globally and related functions.

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 variables dynamically to a list of values, and then evaluates some forms. The lists of 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' bindings are made empty. 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. Assuming that the variables a, b, foo and bar are globally special, we can do:

```
(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 like progv except that it has a different way of deciding which variables to bind and what values to give them. Like progv, it always makes dynamic bindings.

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* dynamically 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. progw is useful mainly 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 797); break implements this by using progw.

See also %bind (page 284), which is a subprimitive that gives you maximal control over binding.

3.1.4 Defining Global Variables

Here are the constructs for defining global variables. Each makes the variable globally special, provides a value, records documentation, and allows the editor to find where all this was done.

defvar *variable* [*initial-value*] [*documentation*] *Macro*

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* globally special 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).

If *variable* is void, it is initialized to the result of evaluating the form *initial-value*. *initial-value* is evaluated only if it is to be used.

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 28.8, page 672) 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 void.

defconst *variable initial-value* [*documentation*] *Macro*
defparameter *variable initial-value* [*documentation*] *Macro*

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 gets 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* a 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**.

defconstant *symbol value* [*documentation*] *Macro*

Defines a true constant. The compiler is permitted to assume it will never change. Therefore, if a function that refers to *symbol*'s value is compiled, the compiled function may contain *value* merged into it and may not actually refer to *symbol* at run time.

You should not change the value of *symbol* except by reexecuting the **defconstant** with a new *value*. If you do this, it is necessary to recompile any compiled functions that refer to *symbol* 's value.

3.1.5 The Global Binding

This section describes functions which examine or set the global binding of a variable even when it is shadowed and cannot be accessed simply by evaluating the variable or setting it.

The primary use of these functions is for init files to set variables which are bound by the load function, such as package or base. (setq package (find-package 'foo)) executed from a file being loaded has no effect beyond the end of loading that file, since it sets the binding of package made by load. However, if you use setq-globally instead, the current binding in effect during loading is actually not changed, but when the load exits and the global binding is in effect again, foo will become the current package.

setq-globally {*symbol value*}... *Macro*
Sets each *symbol*'s global binding to the *value* that follows. The *value*'s are evaluated but the *symbol*'s are not.

set-globally *symbol value*
Sets the global binding of *symbol* to *value*.

makunbound-globally *symbol*
Makes the global binding of *symbol* be void.

boundp-globally *symbol*
Returns t if the global binding of *symbol* is not void.

symeval-globally *symbol*
symbol-value-globally *symbol*
Return the value of the global binding of *symbol*. An error is signaled if the global binding is void.

See also pkg-goto-globally (page 638), a "globally" version of pkg-goto. Note that let-globally is *not* analogous to these functions, as it modifies the current bindings of symbols rather than their global bindings. This is an unfortunate collision of naming conventions.

3.2 Generalized Variables

In Lisp, a variable is something that can remember one piece of data. The primary conceptual operations on a variable are to recover that piece of data and to change it. These might be called *access* and *update*. The concept of variables named by symbols, explained above, can be generalized to any storage location that can remember one piece of data, no matter how that location is named.

For each kind of generalized variable, there are typically three functions which implement the conceptual *access*, *update* and *locate* operations. For example, symeval accesses a symbol's value cell, set updates it, and value-cell-location returns the value cell's location. array-leader accesses the contents of an array leader element, store-array-leader updates it, and ap-leader returns the location of the leader element. car accesses the car of a cons, rplaca updates it, and car-location returns the location of the car.

Rather than thinking of this as two functions, which operate on a storage location somehow deduced from their arguments, we can shift our point of view and think of the access function as a *name* for the storage location. Thus (symeval 'foo) is a name for the value of foo, and (aref a 105) is a name for the 105th element of the array a. Rather than having to remember the update function associated with each access function, we adopt a uniform way of updating storage locations named in this way, using the setf special form. This is analogous to the way we use the setq special form to convert the name of a variable (which is also a form which accesses it) into a form that updates it. In fact, setf is an upward compatible generalization of setq. Similarly, the location of the generalized variable can be obtained using the locf construct.

3.2.1 setf

setf is the construct for storing a new value into a generalized variable which is identified by the form which would obtain the current value of the variable. For example,

```
(setf (car x) y)
```

stores the value of y into the car of the value of x.

setf is particularly useful in combination with structure-accessing macros, such as those created with defstruct, because the knowledge of the representation of the structure is embedded inside the macro, and the programmer shouldn't have to know what it is in order to alter an element of the structure.

setf is actually a macro which expands into the appropriate update code. It has a database, explained in section 18.10, page 345, that associates from access functions to update functions.

setf {*place value*}... *Macro*

Takes a form called *place* that *accesses* something and "inverts" the form to produce a corresponding form to *update* the thing. A setf expands into an update form, which stores the result of evaluating the form *value* into the place referenced by the *place*. If multiple *place*'s and *value*'s are specified, each one specifies an update, and each update is done before the following updates' arguments are computed.

Examples:

```
(setf (array-leader foo 3) 'bar)
                ==> (store-array-leader 'bar foo 3)
(setf a 3) ==> (setq a 3)
(setf (plist 'a) '(foo bar)) ==> (setplist 'a '(foo bar))
(setf (aref q 2) 56) ==> (sys:set-aref q 2 56)
(setf (cadr w) x) ==> (sys:setcdr (cdr w) x)
```

The value of a setf form is always the *value* stored by the last update it performs. Thus, (setf (cadr w) x) is not really the same as (rplaca (cdr w) x), because the setf returns x and the rplaca returns w. In fact, the expansion of setf of cdr uses an internal function si:setcdr which exists specifically for this purpose.

If *place* invokes a macro or a substitutable function, then setf expands the *place* and starts over again. This lets you use setf together with defstruct accessor macros.

sys:unknown-setf-reference (error) *Condition*
sys:unknown-locf-reference (error) *Condition*

These are signaled when setf or locf does not know how to expand the *place*. The :form operation on the condition instance returns the *access-form*.

psetf {*place value*}... *Macro*

Stores each *value* into the corresponding *place*, with the changes taking effect in parallel. Thus,

```
(psetf (car x) (cdr x) (cdr x) (car x))
```

interchanges the car and cdr of x.

The subforms of the *places*, and the *values*, are evaluated in order; thus, in

```
(psetf (aref a (tyi)) (tyi)
       (aref b (tyi)) (aref a (tyi)))
```

the first input character indexes a, the second is stored, the third indexes b, and the fourth indexes a. The parallel nature of psetf implies that, should the first and fourth characters be equal, the old value of that element of a is what is stored into the array b, rather than the new value which comes from the second character read.

shiftf *place...* *Macro*

Sets the first *place* from the second, the second from the third, and so on. The last *place* is not set, so it doesn't really need to be a setf'able place; it can be any form. The value of the shiftf form is the old value of the first *place*. Thus,

```
(shiftf x (car (foo)) b)
```

evaluates (foo), copies the car of that value into x, copies b into the car of that value, then returns the former value of x.

rotatef *place...* *Macro*

Sets the first *place* from the second, the second from the third, and so on, and sets the last *place* from the old value of the first *place*. Thus, the values of the *place*'s are permuted among the *place*'s in a cyclic fashion.

With only two place's, their values are exchanged:

```
(rotatef (car x) (cdr x))
```

is equivalent to the psetf example above.

swapf *place1 place2* *Macro*

Exchanges the contents of *place1* and *place2*. This is a special case of rotatef.

incf *place* [*amount*] *Macro*

Increments the value of a generalized variable. (incf *ref*) increments the value of *ref* by 1. (incf *ref amount*) adds *amount* to *ref* and stores the sum back into *ref*. The incf form returns the value after incrementation.

incf expands into a setf form, so *ref* can be anything that setf understands as its *place*.

incf is defined using define-modify-macro, page 349.

decf *place* [*amount*] *Macro*

Decrements the value of a generalized variable. Just like **incf** except that *amount* (or 1) is subtracted rather than added.

See also **push** (page 88), **pop** (page 88), **pushnew** (page 107), **getf** (page 115) and **remf** (page 115).

3.2.2 locf

Besides the *access* and *update* conceptual operations on generalized variables, there is a third basic operation, which we might call *locate*. Given the name of a storage cell, the *locate* operation returns the address of that cell as a locative pointer (see chapter 14, page 267). This locative pointer is a first-class Lisp data object which is a kind of reference to the cell. It can be passed as an argument to a function which operates on any cell, regardless of where the cell is found. It can be used to *bind* the contents of the cell, just as special variables are bound, using the %bind subprimitive (see page 284).

Of course, this can work only on generalized variables whose implementation is really to store their value in a memory cell. A generalized variable with an *update* operation that encrypts the value and an *access* operation that decrypts it could not have the *locate* operation, since the value per se is not actually stored anywhere.

locf *place* *Macro*

locf takes a form that *accesses* some cell, and produces a corresponding form to create a locative pointer to that cell.

Examples:

```
(locf (array-leader foo 3)) ==> (ap-leader foo 3)
(locf a) ==> (value-cell-location 'a)
(locf (plist 'a)) ==> (property-cell-location 'a)
(locf (aref q 2)) ==> (aloc q 2)
```

If *place* invokes a macro or a substitutable function, then **locf** expands the *place* and starts over again. This lets you use **locf** together with **defstruct** accessor macros.

3.3 Functions

In the description of evaluation on page 24, 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 11, page 223. Here we 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 signaled. 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 get an error because **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 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, more readable styles of call.

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 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; all keyword parameters are optional.

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 is 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 algorithm used to match up the arguments with the parameters:

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 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 each 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 is 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 eq against the allowed parameter keywords, which have by default the same names as the parameters but in the keyword package. (You can specify the keyword symbol explicitly in the lambda list if you must; see below.) Often the symbol arguments are constants in the program, and it is convenient for this usage that keywords all evaluate to themselves, but it is permissible for them to be computed by expressions.

If any keyword parameter has not received a value when all the arguments have been processed, the default-form for the parameter is evaluated and the parameter is bound to its value. All keyword parameters are optional.

There may be a keyword symbol among the arguments which does not match any keyword parameter name. By default this is an error, but the lambda list can specify that there should be no error using &allow-other-keys. Also, if one of the keyword symbols among the arguments is :allow-other-keys and the value that follows it is non-nil then there is no error. When there is no error, for either reason, the non-matching symbols and their associated values are simply 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 apply; that function will check for the keywords that concern it.

The way you express which parameters are required, optional, rest and keyword 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 zero, one, two or three arguments.

`(&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 keyword parameters. A typical call would look like

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

or

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

or

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

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

`(x &optional y &rest z &key a b)`
x is required positional, y is optional positional, z is rest, and a and b are 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 a b c &allow-other-keys)`
z is rest, and a, b and c are 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 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. Just by looking at the value one cannot distinguish between omitting it and passing the default value explicitly as an argument. 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 is bound to 3 and c is bound to nil. If the function is called with two arguments, b is bound to the value that was passed by the caller (which might be 3), and c is 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)) ((:b b) t))`

This is equivalent to (&key a (b t)).

`(&key ((:base base-value)))`

This defines an argument which callers specify with the keyword :base, but which within the function is referred to as the variable base-value so as to avoid binding the value of base, which is a synonym for *print-base* and controls how numbers are printed.

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 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.

You could, equivalently, use **(a &optional b &rest c)** as the lamda list and write **(let* (d (e 5) (f (cons a e))) ...)** around the body of the function.

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 5.9, page 112) 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 94), as you must always assume that it is one of these special lists. The system does 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 may 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 is unsafe in this case, while in the first case it would cause an error, since lists in the stack are impossible to **rplacd**.

lambda-parameters-limit *Constant*

Has as its value the limit on the number of parameters that a lambda list may have. The implementation limit on the number of parameters allowed is at least this many. There is no promise that this many is forbidden, but it is a promise that any number less than this many is permitted.

3.3.1 Lambda-List Keywords

This section documents all the keywords that may appear in the lambda list or argument list (see section 3.3, page 38) of a function, a macro, or a special form. Some of them are allowed everywhere, while others are only allowed in one of these contexts; those are so indicated. You need only know about **&optional**, **&key**, and **&rest** in order to understand the documentation of system functions in this manual.

lambda-list-keywords *Constant*

The value of this variable is a list of all of the allowed '&' keywords. A list of them follows.

&optional Separates the required arguments of a function from the optional arguments. See section 3.3, page 38.

&rest Separates the required and optional arguments of a function from the rest argument. There may be only one rest argument. See page 41 for full information about rest arguments. See section 3.3, page 38.

&key Separates the positional arguments and rest argument of a function from the keyword arguments. See section 3.3, page 38.

&allow-other-keys
In a function that accepts keyword arguments, says that keywords that are not recognized are allowed. They and the corresponding values are ignored, as far as keyword arguments are concerned, but they do become part of the rest argument, if there is one.

&aux Separates the arguments of a function from the auxiliary variables. Following **&aux** you can put entries of the form

(*variable initial-value-form*)

or just *variable* if you want it initialized to **nil** or don't care what the initial value is.

&special Declares the following arguments and/or auxiliary variables to be special within the scope of this function.

&local Turns off a preceding **&special** for the variables that follow.

&quote Declares that the following arguments are not to be evaluated. This is how you create a special function. See the caveats about special forms on page 233.

&eval Turns off a preceding **&quote** for the arguments which follow.

&list-of This is for macros defined by **defmacro** only. Refer to page 324.

&body This is for macros defined by **defmacro** only. It is similar to **&rest**, but declares to **grindef** and the code-formatting module of the editor that the body forms of a special form follow and should be indented accordingly. Refer to page 324.

&whole This is for macros defined by **defmacro** only. It means that the following argument is bound to the entire macro call form being expanded. Refer to page 324.

&environment This is for macros defined by **defmacro** only. It means that the following argument is bound to an environment structure which records the local **macrolet** macro definitions in effect for subforms of the macro call form. Refer to page 324.

3.3.2 Local Functions

The constructs flet and labels permit you to define a function name in a lexical context only. If the same name has a global function definition, it is shadowed temporarily. Function definitions established by flet (or labels) are to global definitions made with defun as lexical variable bindings made with let are to global bindings made with defvar. They always have lexical scope.

flet *local-functions body...* *Special form*

Executes *body* with local function definitions in effect according to *local-functions*.

local-functions should be a list of elements which look like

(*name lambda-list function-body...*)

just like the cdr of a defun form. The meaning of this element of *local-functions* is to define *name* locally with the indicated definition. Within the lexical scope of *body*, using *name* as a function name accesses the local definition.

Example:

```
(flet ((triple (x) (* x 3)))
  (print (triple -1))
  (mapcar (function triple) '(1 2 1.2)))
```

prints the number -3 and returns a list (3 6 3.6).

Each local function is closed in the environment outside the flet. As a result, the local functions cannot call each other.

```
(flet ((foo (x) (bar x t))
       (bar (y z) (list y z)))
  (foo t))
```

calls the local definition of foo, which calls the *global* definition of bar, because the body of foo is not within the scope of the local definition of bar.

Functions defined with flet inside of a compiled function can be referred to by name in a function spec of the form (:internal *outer-function-name flet-name*). See page 226.

labels *local-functions body...* *Special form*

Is like *flet* except that the local functions can call each other. They are closed in the environment inside the labels, so all the local function names are accessible inside the bodies of the local functions. labels is one of the most ancient Lisp constructs, but was typically not implemented in second generation Lisp systems in which no efficient form of closure existed.

```
(labels ((walk (x)
           (typecase x
             (cons (walk (car x)) (walk (cdr x)))
             (t (if (eq x 'haha) (print 'found-it))))))
  (walk foo))
```

allows walk to call itself recursively because walk's body is inside the scope of the definition of walk.

See also macrolet, an analogous construct for defining macros locally (page 329).

3.4 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 *form* &optional *nohook*

(eval *form*) evaluates *form*, and returns the result.

Example:

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

The dynamic bindings available at the time eval is called are visible for dynamic variables within the expression *x*. No lexical bindings are available for the evaluation of *x*.

It is unusual to call eval explicitly, 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, mathematics or games.

Also, if you are only interested in getting at the dynamic value of a symbol (that is, the contents of the symbol's value cell), then you should use the primitive function symeval (see page 129).

If the argument *nohook* is non-nil, execution of the evalhook is inhibited for *form*, but not for evaluation of the subforms of *form*. See evalhook, page 749. evalhook is also the way to evaluate in a specified lexical environment if you happen to have got your hands on one.

Note: in Maclisp, the second argument to eval is a "binding context pointer". There is no such thing in Zetalisp; closures are used instead (see chapter 12, page 250).

si:eval1 *form* &optional *nohook*

Within the definition of a special form, evaluates *form* in the *current* lexical environment.

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.

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.

apply *f* &rest *args*
lexpr-funcall *f* &rest *args*

apply is like funcall except that the last of *args* is really a list of arguments to give to *f* rather than a single argument. lexpr-funcall is a synonym for apply; formerly, apply was limited to the two argument case.

(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.

If there is more than one element of *args*, then all but the last of them are individual arguments to pass to *f*, while the last one is a list of arguments as above.
Examples:

```
(apply 'plus 1 1 1 '(1 1 1)) => 6

(defun report-error (&rest args)
   (apply 'format *error-output* args))
```

apply can also be used with a single argument. Then this argument is a list of a function and some arguments to pass it.
Example:

```
(apply '(car (a))) => a
      ;Not the same as (eval '(car (a)))
```

Note: in Maclisp, apply takes two or three arguments, and the third argument, when passed, is interpreted as a "binding context pointer". So the second argument always provides all the args to pass to the function. There are no binding context pointers in Zetalisp; true lexical scoping exists and is interfaced in other ways.

call-arguments-limit *Constant*

Has as its value the limit on the number of arguments that can be dealt with in a function call. There is no promise that this many is forbidden, but it is a promise that any smaller number is acceptable.

Note that if apply is used with exactly two arguments, the first one being a function that takes a rest argument, there is no limit except the size of memory on the number of elements in the second argument to apply.

call *function* &rest *argument-specifications*

Offers a very general way of controlling what arguments you pass to a function. You can provide either individual arguments as in **funcall** or lists of arguments as in **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** *object*) simply returns *object*. **quote** is used to include constants in a form. It is useful specifically because *object* is not evaluated; the **quote** is how you make a form that returns an arbitrary Lisp object.

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*

function has two distinct, though related, meanings.

If *f* is a symbol or any other function spec (see section 11.2, page 223), (**function** *f*) refers to the function definition of *f*. For example, in (**mapcar** (**function car**) **x**), the function definition of **car** is passed as the first argument to **mapcar**. **function** used this way is like **fdefinition** except that its argument is unevaluated, and so

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

f can also be an explicit function, or lambda-expression, a list such as (lambda (x) (* x x)) such as could be the function definition of a symbol. Then (function *f*) represents that function, suitably interfaced to execute in the lexical environment where it appears. To explain:

```
(let (a)
  (mapcar (lambda (x) (push x a)) l))
```

attempts to call the function lambda and evaluate (x) for its first argument. That is no way to refer to the function expressed by (lambda (x) (push x a)).

```
(let (a)
  (mapcar (quote (lambda (x) (push x a))) l))
```

passes to mapcar the list (lambda (x) (push x a)). This list does not in any way record the lexical environment where the quote form appeared, so it is impossible to make this environment, with its binding of a, available for the execution of (push x a). Therefore, the reference to a does not work properly.

```
(let (a)
  (mapcar (function (lambda (x) (push x a))) l))
```

passes mapcar a specially designed closure made from the function represented by (lambda (x) (push x a)). When mapcar calls this closure, the lexical environment of the function form is put again into effect, and the a in (push x a) refers properly to the binding made by this let.

In addition, the compiler knows that the argument to function should be compiled. The argument of quote cannot be compiled since it may be intended for other uses.

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. The last example could be written as

```
(let (a)
  (mapcar #'(lambda (x) (push x a)) l))
```

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, as that would be difficult to make sense of. Instead, write

```
(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 test function, there are two things you could say:

```
(setq x 'test)
```

or

```
(setq x #'test)
```

The former causes the value of x to be the symbol test, whereas the latter causes the value of x to be the function object found in the function cell of test. When the time comes to call the function (the program does (funcall x ...)), either expression works because calling a symbol as a function uses its function definition instead. Using 'test is insignificantly slower, because the function call has to indirect through the symbol, but it allows the function to be redefined, traced (see page 738), or advised (see page 742). Use of #' picks up the function definition out of the symbol test when the setq is done and does not see any later changes to it. #' should be used only if you wish specifically to prevent redefinition of the function from affecting this closure.

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. It is most useful for commenting out function definitions that are not needed or correct but worth preserving in the source. The #|...|# syntactic construct is an alternative method. For comments within code about the code, it is better to use semicolons.

Example:

```
(comment
;; This is brain-damaged.  Can someone figure out
;; how to do this right?
(defun foo (x)
  ...)
) ;End comment
;; prevents this definition of foo from being used.
```

3.5 Declarations

Declarations provide auxiliary information on how to execute a function or expression properly. The most important declarations are **special** declarations, which control the scope of variable names. Some declarations do not affect execution at all and only provide information about a function, for the sake of **arglist**, for example.

Declarations may apply to an entire function or to any expression within it. Declarations can be made around any subexpression by writing a **local-declare** around the subexpression or by writing a **declare** at the front of the body of certain constructs. Declarations can be made on an entire function by writing a **declare** at the front of the function's body.

local-declare (*declaration...*) *body...* *Special form*

A **local-declare** form looks like

```
(local-declare (decl1 decl2 ...)
  form1
  form2
  ...)
```

Each *decl* is in effect for the forms in the body of the **local-declare** form.

declare *declaration...* *Special form*

The special form **declare** is used for writing local declarations within the construct they apply to.

A *declare* inside a function definition, just after the argument list, is equivalent to putting a **local-declare** around the function definition. More specifically,

```
(defun foo (a b)
  (declare (special a b))
  (bar))
```

is equivalent to

```
(local-declare ((special a b))
(defun foo (a b)
  (bar)))
```

Note that

```
(defun foo (a b)
  (local-declare ((special a b))
    (bar)))
```

does not do the job, because the declaration is not in effect for the binding of the arguments of **foo**.

declare is preferable to local-declare in this sort of situation, because it allows the defuns themselves to be the top-level lists in the file. While local-declare might appear to have an advantage in that one local-declare may go around several defuns, it tends to cause trouble to use local-declare in that fashion.

declare has a similar meaning at the front of the body of a progn, prog, let, prog*, let*, or internal lambda. For example,

```
(prog (x)
      (declare (special x))
      ...)
```

is equivalent to

```
(local-declare ((special x))
  (prog (x)
        ...))
```

At top level in the file, (declare *forms...*) is equivalent to (eval-when (compile) *forms...*). This use of declare is nearly obsolete, and should be avoided. In Common Lisp, proclaim (below) is used for such purposes, with a different calling convention.

Elsewhere, declare's are ignored.

Here is a list of declarations that have system-defined meanings:

(special *var1 var2...*)
: The variables *var1*, *var2*, etc. will be treated as special variables in the scope of the declaration.

(unspecial *var1 var2...*)
: The variables *var1*, *var2*, etc. will be treated as lexical variables in the scope of the declaration, even if they are globally special.

(notinline *fun1 fun2...*)
: The functions *fun1*, *fun2* and so on will not be open coded or optimized by the compiler within the scope of the declaration.

(inline *fun1 fun2...*)
: The functions *fun1*, *fun2* and so on will be open coded or optimized by the compiler (to whatever extent it knows how) within the scope of the declaration. Merely issuing this declaration does not tell the compiler how to do any useful optimization or open coding of a function.

(ignore *var1 var2...*)
: Says that the variables *var1*, *var2*, etc., which are bound in the construct in which this declaration is found, are going to be ignored. This is currently significant only in a function being compiled; the compiler issues a warning if the variables are used, and refrains from its usual warning if the variables are ignored.

(declaration *decl1 decl2...*)
: Says that declarations *decl1*, *decl2*, etc. are going to be used, and prevents any warning about an unrecognized type of declaration. For example:

```
(defun hack ()
  (declare (declaration lose-method)
           (lose-method foo bar))
  ... (lose foo) ...)
```

might be useful if (lose foo) is a macro whose expander function does (getdecl 'foo 'lose-method) to see what to do. See page 307 for more information on getdecl and declarations.

```
(proclaim '(declaration lose-method))
```

might also be advisable if you expect widespread use of lose-method declarations.

The next two are used by the compiler and generally should not be written by users.

(def *name* . *definition*)

name will be defined for the compiler in the scope of the declaration. The compiler uses this automatically to keep track of macros and open-codable functions (defsubsts) defined in the file being compiled. Note that the cddr of this item is a function.

(*propname symbol value*)

(getdecl *symbol propname*) will return *value* in the scope of the declaration. This is how the compiler keeps track of defdecls.

These declarations are significant only when they apply to an entire defun.

(arglist . *arglist*)

Records *arglist* as the argument list of the function, to be used instead of its lambda list if anyone asks what its arguments are. This is purely documentation.

(values . *values*) or (:return-list . *values*)

Records *values* as the return values list of the function, to be used if anyone asks what values it returns. This is purely documentation.

(sys:function-parent *parent-function-spec*)

Records *parent-function-spec* as the parent of this function. If, in the editor, you ask to see the source of this function, and the editor doesn't know where it is, the editor will show you the source code for the parent function instead.

For example, the accessor functions generated by defstruct have no defuns of their own in the text of the source file. So defstruct generates them with sys:function-parent declarations giving the defstruct's name as the parent function spec. Visiting the accessor function with Meta-. sees the declaration and therefore visits the text of the defstruct.

(:self-flavor *flavorname*)

Instance variables of the flavor *flavorname*, in self, will be accessible in the function.

locally &body *body* *Macro*

Executes the *body*, recognizing declarations at the front of it. locally is synonymous with progn except that in Common Lisp a declare is allowed at the beginning of a locally and not at the beginning of a progn.

locally does differ from progn in one context: at top level in a file being compiled, progn causes each of its elements (including declarations, therefore) to be treated as if at top level. locally does not receive this treatment. The locally form is simply evaluated when the QFASL file is loaded.

proclaim &rest *declarations*

Each of *declarations* is put into effect globally. Currently only special and unspecial declarations mean anything in this way. proclaim's arguments are evaluated, and the values are expected to be declarations such as you could write in a declare. Thus, you would say (proclaim '(special x)) to make a special declaration globally.

Top-level special declarations are not the recommended way to make a variable special. Use defvar, defconstant or defparameter, so that you can give the variable documentation. Proclaiming the variable special should be done only when the variable is used in a file other than the one which defines it, to enable the file to be compiled without having to load the defining file first.

proclaim is fairly new. Until recently, top-level declare was the preferred way to make global special declarations when defvar, etc., could not be used. Such top-level declare's are still quite common. In them, the declaration would not be quoted; for example, (declare (special x)).

special *variables...* *Special form*

Equivalent to (proclaim (special *variables*...)), this declares each of the *variables* to be globally special. This function is obsolete.

unspecial *variables...* *Special form*

Removes any global special declarations of the *variables*. This function is obsolete.

the *type-specifier value-form* *Macro*

Is a Common Lisp construct effectively the same as *value-form*. It declares that the value of *value-form* is an object which of type *type-specifier*. This is to assist compilers in generating better code for conventional machine architectures. The Lisp Machine does not make use of type declarations so this is the same as writing just *value-form*. *type-specifier* is not evaluated.

If you want the type of an object to be checked at run time, with an error if it is not what it is supposed to be, use check-type (page 709).

3.6 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. 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 a function can do that can make it dangerous to discard its 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 stack frame to ensure proper execution. Some of these conditions occur in eval; as a result, interpreted code is never completely tail recursive.

3.7 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 returns three values. Many system functions produce multiple values, but they all do it via values.

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. Equivalent to (apply 'values *list*).

return and its variants can also be used, within a block, do or prog special form, to return multiple values. They are explained on page 77.

Here are the special forms for receiving multiple values.

multiple-value (*variable...*) *form* *Special form*
multiple-value-setq (*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 (setting nil is not allowed anyway).
Example:

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

In addition to its first value (the symbol), intern returns a second value, which is non-nil if an existing symbol was found, or else nil if intern had to create one. So if the symbol goo was already known, the variable already-there-p is set non-nil, otherwise it is set to nil. The third value returned by intern is ignored by this form of call since there is no third variable in the multiple-value.

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-setq is the Common Lisp name for this construct. The two names are equivalent.

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-call *function argforms...* *Special form*

Evaluates the argforms, saving all of their values, and then calls *function* with all those values as arguments. This differs from

```
(funcall function argforms...)
```

because that would get only one argument for *function* from each *argform*, whereas multiple-value-call gets as many args from each *argform* as the *argform* cares to return. This works by consing a list of all the values returned, and applying *function* to it. Example:

```
(multiple-value-call 'append
      (values '(a b) '(c d))
      '(e f))
  => (a b c d e f)
```

multiple-value-prog1 *form forms...* *Special form*

Evaluates *form*, saves its values, evaluates the *forms*, discards their values, then returns whatever values *form* produced. This does not cons. Example:

```
(multiple-value-prog1 (values 1 2)
      (print 'foo))
  => 1 2
```

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 is set to a list of three elements, the three values returned by intern.

nth-value *n form* *Special form*

Evaluates *form* and returns its value number *n*, *n* = 0 meaning the first value. For example, (nth-value 1 (foo)) returns the second of foo's values. nth-value operates without consing in compiled code if the first argument's value is known at compile time.

When one form finished by tail recursively evaluating a subform (see section 3.6, page 55), 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 is still called on only 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. To pass all returned values to another function, use multiple-value-call, above.

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 actually 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 and 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 only from the last subform of an and or an or form, not from previous subforms since the return is conditional. Remember that multiple values are only passed back when the value of a subform 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 is 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 is 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.

If a block form falls through the end, it returns all the values returned by the last expression in it. If return-from or return is used to exit a block form, then the values returned by the block form depend on the kind of return. If return is given two or more subforms, then block returns as many values as the return has subforms. However, if the return has only one

subform, then the block returns all of the values returned by that one subform.

prog behaves like block if it is exited with return (or return-from). If control falls through the end of a prog, it returns the single value nil. do also behaves like block with respect to return, but if it is exited through the exit test, 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 passes back multiple values from the last form in its body, if it exits normally. If a throw is done, multiple values are passed back from the value form in the throw.

multiple-values-limit *Constant*

The smallest number of values that might possibly fail to work. Returning a number of values less than this many cannot possibly run into trouble with an implementation limit on number of values returned.

3.8 Evaluation and Function Calling Errors

Here is a description of the error conditions that the evaluator can signal. Some can be signaled by calls to compiled functions also. This is for use by those who are writing condition handlers (section 30.2, page 700). The novice should skip this section.

sys:invalid-function (error) *Condition*

This is signaled when an object that is supposed to be applied to arguments 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:undefined-keyword-argument (error) *Condition*

This is signaled when a function that takes keyword arguments is given a keyword that it does not accept, if &allow-other-keys was not used in the function's definition and :allow-other-keys was not specified by the caller (see page 40). 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 references to void memory locations. 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 were found in the referenced cell when the error happened. This can be different from the original address in the case of dynamic 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 whose values are void.

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 erroneous argument. 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 type specifier 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.