symbolics™

Volume 3B
Lisp Language

Volume 3B. Lisp Language

#996032

Design: Schafer/LaCasse
Cover and title page typography: Litho Composition Co.
Text typography: Century Schoolbook and Helvetica produced on a Symbolics 3600 Lisp Machine from Bitstream, Inc., outlines; text master printed on Symbolics LGP-1 Laser Graphics Printer.

The first Lisp Machine system was a product of the efforts of many people at the M.I.T. Artificial Intelligence Laboratory, and of the unique environment there. Portions of earliest versions of many of the documents in this documentation set were written at the AI Lab.

symbolics™

Contents

symbolics™

FUNC Functions

Cambridge, Massachusetts

Functions

990073

February 1984

This document corresponds to Release 5.0.

This document was prepared by the Documentation Group of Symbolics, Inc.

Printing year and number: 87 86 85 84 9 8 7 6 5 4 3 2 1

Table of Contents

1. Functions

Functions are the basic building blocks of Lisp programs. This chapter describes the functions in Zetalisp that are used to manipulate functions. It also explains how to manipulate special forms and macros.

This chapter contains internal details intended for those writing programs to manipulate programs as well as material suitable for the beginner.

1.1 What is a Function?

There are many different kinds of functions in Zetalisp. Here are the printed representations of examples of some of them:

```
foo
(lambda (x) (car (last x)))
(named-lambda foo (x) (car (last (x))))
(subst (x) (car (last x)))
#<dtp-fef-pointer append 1424771>
#<dtp-u-entry last 270>
#<dtp-closure 1477464>
```

We will examine these and other types of functions in detail later. They all have one thing in common: a function is a Lisp object that can be applied to arguments. All of the above objects can be applied to some arguments and will return a value. Functions are Lisp objects and so can be manipulated in all the usual ways: you can pass them as arguments, return them as values, and make other Lisp objects refer to them.

1.2 Function Specs

The name of a function does not have to be a symbol. Various kinds of lists describe other places where a function can be found. A Lisp object that describes a place to find a function is called a *function spec*. ("Spec" is short for "specification".) Here are the printed representations of some typical function specs:

```
foo
(:property foo bar)
(:method tv:graphics-mixin :draw-line)
(:internal foo 1)
(:within foo bar)
(:location #<dtp-locative 7435216>)
```

Function specs have two purposes: they specify a place to *remember* a function, and they serve to *name* functions. The most common kind of function spec is a symbol, that specifies that the function cell of the symbol is the place to remember the function. We will see all the kinds of function specs, and what they mean, shortly. Function specs are not the same thing as functions. You cannot, in general, apply a function spec to arguments. The time to use a function spec is when you want to *do* something to the function, such as define it, look at its definition, or compile it.

Some kinds of functions remember their own names, and some do not. The "name" remembered by a function can be any kind of function spec, although it is usually a symbol. (See the section "What is a Function?".) In that section, the example starting with the symbol **named-lambda**, the one whose printed representation included **dtp-fef-pointer**, and the **dtp-u-entry** remembered names (the function specs **foo**, **append**, and **last** respectively). The others did not remember their names.

To *define a function spec* means to make that function spec remember a given function. This is done with the **fdefine** function; you give **fdefine** a function spec and a function, and **fdefine** remembers the function in the place specified by the function spec. The function associated with a function spec is called the *definition* of the function spec. A single function can be the definition of more than one function spec at the same time, or of no function specs.

To *define a function* means to create a new function, and define a given function spec as that new function. This is what the **defun** special form does. Several other special forms such as **defmethod** and **defselect** do this too.

These special forms that define functions usually take a function spec, create a function whose name is that function spec, and then define that function spec to be the newly created function. Most function definitions are done this way, and so usually if you go to a function spec and see what function is there, the function's name will be the same as the function spec. However, if you define a function named **foo** with **defun**, and then define the symbol **bar** to be this same function, the name of the function is unaffected; both **foo** and **bar** are defined to be the same function, and the name of that function is **foo**, not **bar**.

A function spec's definition in general consists of a *basic definition* surrounded by *encapsulations*. Both the basic definition and the encapsulations are functions, but of recognizably different kinds. What **defun** creates is a basic definition, and usually that is all there is. Encapsulations are made by function-altering functions such as **trace** and **advise**. When the function is called, the entire definition, which includes the tracing and advice, is used. If the function is "redefined" with **defun**, only the basic definition is changed; the encapsulations are left in place. See the section "Encapsulations".

A function spec is a Lisp object of one of the following types:

a symbol

The function is remembered in the function cell of the symbol. See the section "The Function Cell". Function cells and the primitive functions to manipulate them are explained in that section.

(:property *symbol property***)**
The function is remembered on the property list of the symbol; doing **(get** *symbol property***)** would return the function. Storing functions on property lists is a frequently used technique for dispatching (that is, deciding at run-time which function to call, on the basis of input data).

(:method *flavor-name message***)**

(:method *flavor-name method-type message***)**
The function is remembered inside internal data structures of the flavor system. See the document *Objects, Message Passing, and Flavors.*

(:select-method *function-spec message***)**
If the definition of *function-spec* is a select-method, this refers to the function to which the select method dispatches upon receiving *message*. It is an error if *function-spec* does not contain a select method or if the select method does not support *message*. **defselect** now defines its component functions using this function spec instead of creating a special symbol for each function.

(:handler *flavor-name message***)**
This is a name for the function actually called when a *message* message is sent to an instance of the flavor *flavor-name*. The difference between **:handler** and **:method** is that the handler may be a method inherited from some other flavor or a *combined method* automatically written by the flavor system. Methods are what you define in source files; handlers are not. Note that redefining or encapsulating a handler affects only the named flavor, not any other flavors built out of it. Thus **:handler** function specs are often used with **trace** and **advise**.

(:location *pointer***)**
The function is stored in the cdr of *pointer*, which may be a locative or a list. This is for pointing at an arbitrary place which there is no other way to describe. This form of function spec is not useful in **defun** (and related special forms) because the reader has no printed representation for locative pointers and always creates new lists; these function specs are intended for programs that manipulate functions. See the section "How Programs Manipulate Definitions".

(:within *within-function function-to-affect***)**
This refers to the meaning of the symbol *function-to-affect*, but only where it occurs in the text of the definition of *within-function*. If you define this function spec as anything but the symbol *function-to-affect* itself, then that symbol is replaced throughout the definition of *within-function* by a new symbol which is then defined as you specify. See the section "Encapsulations".

(:internal *function-spec number***)**

Some Lisp functions contain internal functions, created by **(function (lambda ...))** forms. These internal functions need names when compiled, but they do not have symbols as names; instead they are named by **:internal** function-specs. *function-spec* is the containing function. *number* is a sequence number; the first internal function the compiler comes across in a given function will be numbered 0, the next 1, and so on. Internal functions are remembered inside the **FEF** of their containing function.

Here is an example of the use of a function spec that is not a symbol:

```
(defun (:property foo bar-maker) (thing &optional kind)
  (set-the 'bar thing (make-bar 'foo thing kind)))
```

This puts a function on **foo's bar-maker** property. Now you can say:

```
(funcall (get 'foo 'bar-maker) 'baz)
```

Unlike the other kinds of function spec, a symbol *can* be used as a function. If you apply a symbol to arguments, the symbol's function definition is used instead. If the definition of the first symbol is another symbol, the definition of the second symbol is used, and so on, any number of times. But this is an exception; in general, you cannot apply function specs to arguments.

A keyword symbol that identifies function specs (may appear in the car of a list that is a function spec) is identified by a **sys:function-spec-handler** property whose value is a function which implements the various manipulations on function specs of that type. The interface to this function is internal and not documented in this manual.

For compatibility with Maclisp, the function-defining special forms **defun, macro,** and **defselect** (and other defining forms built out of them, such as **defunp** and **defmacro**) will also accept a list:

(*symbol property*)

as a function name. This is translated into:

(:property *symbol property*)

symbol must not be one of the keyword symbols which identifies a function spec, since that would be ambiguous.

1.3 Simple Function Definitions

defun *Special Form*

defun is the usual way of defining a function that is part of a program. A **defun** form looks like:

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

name is the function spec you wish to define as a function. The *lambda-list* is a list of the names to give to the arguments of the function. Actually, it is a little more general than that; it can contain *lambda-list keywords* such as **&optional** and **&rest**. (Keywords are explained in other sections. See the section "Functions: Evaluation". See the section "Lambda-list Keywords".) Additional syntactic features of **defun** are explained in another section. See the section "Function-defining Special Forms".

defun creates a list which looks like:

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

and puts it in the function cell of *name*. *name* is now defined as a function and can be called by other forms.

Examples:

```
(defun addone (x)
  (1+ x))

(defun foo (a &optional (b 5) c &rest e &aux j)
  (setq j (+ (addone a) b))
  (cond ((not (null c))
         (cons j e))
        (t j)))
```

addone is a function that expects a number as an argument, and returns a number one larger. **foo** is a complicated function that takes one required argument, two optional arguments, and any number of additional arguments that are given to the function as a list named **e**.

A declaration (a list starting with **declare**) can appear as the first element of the body. It is equivalent to a **local-declare** surrounding the entire **defun** form. For example:

```
(defun foo (x)
  (declare (special x))
  (bar))               ;bar uses x free.
```

is equivalent to and preferable to:

```
(local-declare ((special x))
  (defun foo (x)
    (bar)))
```

(It is preferable because the editor expects the open parenthesis of a top-level function definition to be the first character on a line, which isn't possible in the second form without incorrect indentation.)

A documentation string can also appear as the first element of the body (following the declaration, if there is one). (It shouldn't be the only thing in the body; otherwise it is the value returned by the function and so is not interpreted as documentation. A string as an element of a body other than the last element is only evaluated for side effect, and since evaluation of strings has no side effects, they are not useful in this position to do any computation, so they are interpreted as documentation.) This documentation string becomes part of the function's debugging info and can be obtained with the function **documentation**. The first line of the string should be a complete sentence that makes sense read by itself, since there are two editor commands to get at the documentation, one of which is "brief" and prints only the first line. Example:

```
(defun my-append (&rest lists)
   "Like append but copies all the lists.
This is like the Lisp function append, except that
append copies all lists except the last, whereas
this function copies all of its arguments
including the last one."
   ...)
```

defunp *Macro*

Usually when a function uses **prog**, the **prog** form is the entire body of the function; the definition of such a function looks like **(defun** *name arglist* **(prog** *varlist* **...))**. Although the use of **prog** is generally discouraged, **prog** fans may want to use this special form. For convenience, the **defunp** macro can be used to produce such definitions. A **defunp** form such as:

```
(defunp fctn (args)
    form1
    form2
    ...
    formn)
```

expands into:

```
(defun fctn (args)
  (prog ()
        form1
        form2
        ...
        (return formn)))
```

You can think of **defunp** as being like **defun** except that you can **return** out of the middle of the function's body.

See the section "Function-defining Special Forms". Information on defining functions, and other ways of doing so, are discussed in that section.

1.4 Operations the User Can Perform on Functions

Here is a list of the various things a user (as opposed to a program) is likely to want to do to a function. In all cases, you specify a function spec to say where to find the function.

To print out the definition of the function spec with indentation to make it legible, use **grindef**. This works only for interpreted functions. If the definition is a compiled function, it cannot be printed out as Lisp code, but its compiled code can be printed by the **disassemble** function.

To find out about how to call the function, you can ask to see its documentation, or its argument names. (The argument names are usually chosen to have mnemonic significance for the caller). Use **arglist** to see the argument names and **documentation** to see the documentation string. There are also editor commands for doing these things: the c-sh-D and m-sh-D commands are for looking at a function's documentation, and c-sh-A is for looking at an argument list. c-sh-A does not ask for the function name; it acts on the function that is called by the innermost expression that the cursor is inside. Usually this is the function that will be called by the form you are in the process of writing.

You can see the function's debugging info alist by means of the function **debugging-info**.

When you are debugging, you can use **trace** to obtain a printout or a break loop whenever the function is called. You can customize the definition of the function, either temporarily or permanently, using **advise**.

1.5 Kinds of Functions

There are many kinds of functions in Zetalisp. This section briefly describes each kind of function. Note that a function is also a piece of data and can be passed as an argument, returned, put in a list, and so forth.

Before we start classifying the functions, we will first discuss something about how the evaluator works. When the evaluator is given a list whose first element is a symbol, the form may be a function form, a special form, or a macro form. If the definition of the symbol is a function, then the function is just applied to the result of evaluating the rest of the subforms. If the definition is a cons whose car is **macro**, then it is a macro form. See the document *Macros*. What about special forms?

Conceptually, the evaluator knows specially about all special forms (hence their name). However, the Zetalisp implementation actually uses the definition of symbols that name special forms as places to hold pieces of the evaluator. The definitions of such symbols as **prog**, **do**, **and**, and **or** actually hold Lisp objects, which we will call *special functions*. Each of these functions is the part of the Lisp interpreter that knows how to deal with that special form. Normally you do not have to know about this; it is just part of how the evaluator works. However, if you try to add encapsulations to **and** or something like that, knowing this will help you understand the behavior you will get.

Special functions are written like regular functions except that the keywords **&quote** and **&eval** are used to make some of the arguments be "quoted" arguments. See the section "Lambda-list Keywords". The evaluator looks at the pattern in which arguments to the special function are "quoted" or not, and it calls the special function in a special way: for each regular argument, it passes the result of evaluating the corresponding subform, but for each "quoted" argument, it passes the subform itself without evaluating it first. For example, **cond** works by having a special function that takes a "quoted" **&rest** argument; when this function is called it is passed a list of **cond** clauses as its argument.

If you **apply** or **funcall** a special function yourself, you have to understand what the special form is going to do with its arguments; it is likely to call **eval** on parts of them. This is different from applying a regular function, which is passed argument values rather than Lisp expressions.

Defining your own special form, by using **&quote** yourself, can be done; it is a way to extend the Lisp language. Macros are another way of extending the Lisp language. It is preferable to implement language extensions as macros rather than special forms, because macros directly define a Lisp-to-Lisp translation and therefore can be understood by both the interpreter and the compiler. Special forms, on the other hand, only extend the interpreter. The compiler has to be modified to understand each new special form so that code using it can be compiled. Since all real programs are eventually compiled, writing your own special functions is strongly discouraged.

(In fact, many of the special forms in Zetalisp are actually implemented as macros, rather than as special functions. They are implemented this way because it is easier to write a macro than to write both a new special function and a new *ad hoc* module in the compiler. However, they are sometimes documented in this set as special forms, rather than macros, because you should not in any way *depend* on the way they are implemented; they might get changed in the future to be special functions, if there was some reason to do so.)

There are four kinds of functions, classified by how they work.

First, there are *interpreted* functions: you define them with **defun**, they are represented as list structure, and they are interpreted by the Lisp evaluator.

Secondly, there are *compiled* functions: they are defined by **compile** or by loading a bin file, they are represented by a special Lisp data type, and they are executed directly by the microcode. Similar to compiled functions are microcode functions, which are written in microcode (either by hand or by the micro-compiler) and executed directly by the hardware.

Thirdly, there are various types of Lisp object which can be applied to arguments, but when they are applied they dig up another function somewhere and apply it instead. These include **dtp-select-method**, closures, instances, and entities.

Finally, there are various types of Lisp object which, when used as functions, do something special related to the specific data type. These include arrays and stack groups.

1.5.1 Interpreted Functions

An interpreted function is a piece of list structure that represents a program according to the rules of the Lisp interpreter. Unlike other kinds of functions, an interpreted function can be printed out and read back in (it has a printed representation that the reader understands), and it can be pretty-printed. See the section "Formatting Lisp Code". It can also be opened up and examined with the usual functions for list-structure manipulation.

There are four kinds of interpreted functions: **lambdas**, **named-lambdas**, **substs**, and **named-substs**. A **lambda** function is the simplest kind. It is a list that looks like this:

(lambda *lambda-list form1 form2...*)

The symbol **lambda** identifies this list as a **lambda** function. *lambda-list* is a description of what arguments the function takes. See the section "Functions: Evaluation". The *forms* make up the body of the function. When the function is called, the argument variables are bound to the values of the arguments as described by *lambda-list*, and then the forms in the body are evaluated, one by one. The value of the function is the value of its last form.

A **named-lambda** is like a **lambda** but contains an extra element in which the system remembers the function's name, documentation, and other information. Having the function's name there allows the Debugger and other tools to give the user more information. This is the kind of function that **defun** creates. A **named-lambda** function looks like this:

(named-lambda *name lambda-list body forms...*)

If the *name* slot contains a symbol, it is the function's name. Otherwise it is a list whose car is the name and whose cdr is the function's debugging information alist. See **debugging-info**. Note that the name need not be a symbol; it can be any function spec. For example:

```
(defun (foo bar) (x)
  (car (reverse x)))
```

will give **foo** a **bar** property whose value is:

```
(named-lambda ((:property foo bar)) (x) (car (reverse x)))
```

A **subst** is just like a **lambda** as far as the interpreter is concerned. It is a list that looks like this:

```
(subst lambda-list form1 form2...)
```

The difference between a **subst** and a **lambda** is the way they are handled by the compiler. A call to a normal function is compiled as a *closed subroutine*; the compiler generates code to compute the values of the arguments and then apply the function to those values. A call to a **subst** is compiled as an *open subroutine*; the compiler incorporates the body forms of the **subst** into the function being compiled, substituting the argument forms for references to the variables in the **subst**'s *lambda-list*. This is a simple but useful facility for *open* or *in-line coded* functions. It is simple because the argument forms can be evaluated multiple times or out of order, and so the semantics of a **subst** may not be the same in the interpreter and the compiler. **substs** are described more fully in the section that explains **defsubst**. See the section "Substitutable Functions".

A **named-subst** is the same as a **subst** except that it has a name just as a **named-lambda** does. It looks like:

```
(named-subst name lambda-list form1 form2 ...)
```

where *name* is interpreted the same way as in a **named-lambda**.

1.5.2 Compiled Functions

There are two kinds of compiled functions: *macrocoded* functions and *microcoded* functions. The Lisp compiler converts **lambda** and **named-lambda** functions into macrocoded functions. A macrocoded function's printed representation looks like:

```
#<dtp-fef-pointer append 1424771>
```

This type of Lisp object is also called a "Function Entry Frame", or "FEF" for short. Like "car" and "cdr", the name is historical in origin and does not really mean anything. The object contains Lisp Machine machine code that does the computation expressed by the function; it also contains a description of the arguments accepted, any constants required, the name, documentation, and other things. Unlike Maclisp "subr-objects", macrocoded functions are full-fledged objects and can be passed as arguments, stored in data structure, and applied to arguments.

The printed representation of a microcoded function looks like:

```
#<dtp-u-entry last 270>
```

Most microcompiled functions are basic Lisp primitives or subprimitives written in Lisp Machine microcode. You can also convert your own macrocode functions into microcode functions in some circumstances, using the microcompiler.

The compiler now records, as part of its debugging-info property, which top-level macros were expanded in the process of compiling it. This information is used by **who-calls** and similar functions. Thus you can now use **who-calls** for macros. **who-calls** can also find callers of open-coded functions, such as substitutable functions. Functions compiled in earlier versions of the system have not recorded this information; hence **who-calls** will not be able to find them until those sources have been recompiled.

1.5.3 Other Kinds of Functions

A closure is a kind of function that contains another function and a set of special variable bindings. When the closure is applied, it puts the bindings into effect and then applies the other function. When that returns, the closure bindings are removed. Closures are made with the function **closure**. See the section "Closures". Entities are slightly different from closures. See the section "Entities: Closures".

A select-method (**dtp-select-method**) is an alist of symbols and functions. When one is called the first argument is looked up in the alist to find the particular function to be called. This function is applied to the rest of the arguments. The alist may have a list of symbols in place of a symbol, in which case the associated function is called if the first argument is any of the symbols on the list. If **cdr** of **last** of the alist is not **nil**, it is a *default handler* function, which gets called if the message key is not found in the alist. Select-methods can be created with the **defselect** special form.

An instance is a message-receiving object which has some state and a table of message-handling functions (called *methods*). See the document *Objects, Message Passing, and Flavors.*

An array can be used as a function. The arguments to the array are the indices and the value is the contents of the element of the array. This works this way for Maclisp compatibility and is not recommended usage. Use **aref** instead.

A stack group can be called as a function. This is one way to pass control to another stack group. See the section "Stack Groups: Internals".

1.6 Function-defining Special Forms

defun is a special form that is put in a program to define a function. **defsubst** and **macro** are others. This section explains how these special forms work, how they relate to the different kinds of functions, and how they connect to the rest of the function-manipulation system.

Function-defining special forms typically take as arguments a function spec and a description of the function to be made, usually in the form of a list of argument names and some forms which constitute the body of the function. They construct a function, give it the function spec as its name, and define the function spec to be the new function. Different special forms make different kinds of functions. **defun** makes a **named-lambda** function, and **defsubst** makes a **named-subst** function. **macro** makes a macro; though the macro definition is not really a function, it is like a function as far as definition handling is concerned.

These special forms are used in writing programs because the function names and bodies are constants. Programs that define functions usually want to compute the functions and their names, so they use **fdefine**.

All of these function-defining special forms alter only the basic definition of the function spec. Encapsulations are preserved. See the section "Encapsulations".

The special forms only create interpreted functions. There is no special way of defining a compiled function. Compiled functions are made by compiling interpreted ones. The same special form which defines the interpreted function, when processed by the compiler, yields the compiled function. See the document *The Compiler*.

Note that the editor understands these and other "defining" special forms (for example, **defmethod, defvar, defmacro,** and **defstruct**) to some extent, so that when you ask for the definition of something, the editor can find it in its source file and show it to you. The general convention is that anything which is used at top level (not inside a function) and starts with **def** should be a special form for defining things and should be understood by the editor. **defprop** is an exception.

The **defun** special form (and the **defunp** macro that expands into a **defun**) are used for creating ordinary interpreted functions. See the section "Simple Function Definitions".

For Maclisp compatibility, a *type* symbol may be inserted between *name* and *lambda-list* in the **defun** form. The following types are understood:

expr	The same as no type.
fexpr	**&quote** and **&rest** are prefixed to the lambda list.
macro	A macro is defined instead of a normal function.

If *lambda-list* is a non-**nil** symbol instead of a list, the function is recognized as a Maclisp *lexpr* and it is converted in such a way that the **arg, setarg**, and **listify** functions can be used to access its arguments.

The **defsubst** special form is used to create substitutable functions. It is used just like **defun** but produces a list starting with **named-subst** instead of one starting with **named-lambda**. The **named-subst** function acts just like the corresponding **named-lambda** function when applied, but it can also be open-coded (incorporated into its callers) by the compiler. See the section "Substitutable Functions".

The **macro** special form is the primitive means of creating a macro. It gives a function spec a definition which is a macro definition rather than a actual function. A macro is not a function because it cannot be applied, but it *can* appear as the car of a form to be evaluated. Most macros are created with the more powerful **defmacro** special form. See the document *Macros*.

The **defselect** special form defines a select-method function.

Unlike the above special forms, the next two (**deff** and **def**) do not create new functions. They simply serve as hints to the editor that a function is being stored into a function spec here, and therefore if someone asks for the source code of the definition of that function spec, this is the place to look for it.

def *Special Form*

If a function is created in some strange way, wrapping a **def** special form around the code that creates it informs the editor of the connection. The form:

```
(def function-spec
  form1 form2...)
```

simply evaluates the forms *form1*, *form2*, and so on. It is assumed that these forms will create or obtain a function somehow, and make it the definition of *function-spec*.

Alternatively, you could put **(def** *function-spec***)** in front of or anywhere near the forms which define the function. The editor only uses it to tell which line to put the cursor on.

deff *function-spec definition-creator* *Special Form*

deff is a simplified version of **def**. It evaluates the form *definition-creator*, which should produce a function, and makes that function the definition of *function-spec*, which is not evaluated. **deff** is used for giving a function spec a definition that is not obtainable with the specific defining forms such as **defun** and **macro**. For example:

```
(deff foo 'bar)
```

will make **foo** equivalent to **bar**, with an indirection so that if **bar** changes, **foo** will likewise change;

```
(deff foo (function bar))
```

copies the definition of **bar** into **foo** with no indirection, so that further changes to **bar** will have no effect on **foo**.

@define *Macro*

This macro turns into **nil**, doing nothing. It exists for the sake of the @ listing generation program, which uses it to declare names of special forms which define objects (such as functions) that @ should cross-reference.

defselect *Special Form*

defselect defines a function that is a select-method. This function contains a table of subfunctions; when it is called, the first argument, a symbol on the keyword package called the *message name*, is looked up in the table to determine which subfunction to call. Each subfunction can take a different number of arguments, and have a different pattern of **&optional** and **&rest** arguments. **defselect** is useful for a variety of "dispatching" jobs. By analogy with the more general message passing facilities described in the *Objects, Message Passing, and Flavors* document, the subfunctions are sometimes called *methods* and the first argument is sometimes called a *message*.

The special form looks like:

```
(defselect (function-spec default-handler no-which-operations)
  (message-name (args...)
       body...)
  (message-name (args...)
       body...)
  ...)
```

function-spec is the name of the function to be defined. *default-handler* is optional; it must be a symbol and is a function that gets called if the select-method is called with an unknown message. If *default-handler* is unsupplied or **nil**, then an error occurs if an unknown message is sent. If *no-which-operations* is non-**nil**, the **:which-operations** method that would normally be supplied automatically is suppressed. The **:which-operations** method takes no arguments and returns a list of all the message names in the **defselect**.

The **:operation-handled-p** and **:send-if-handles** methods are automatically supplied. See the message **:operation-handled-p**. See the message **:send-if-handles**.

If *function-spec* is a symbol, and *default-handler* and *no-which-operations* are not supplied, then the first subform of the **defselect** may be just *function-spec* by itself, not enclosed in a list.

The remaining subforms in a **defselect** define methods. *message-name* is the message name, or a list of several message names if several messages are to be handled by the same subfunction. *args* is a lambda-list; it should not include the first argument, which is the message name. *body* is the body of the function.

A method subform can instead look like:

```
(message-name . symbol)
```

In this case, *symbol* is the name of a function that is to be called when the *message-name* message is received. It will be called with the same arguments as the select-method, including the message symbol itself.

1.7 Lambda-list Keywords

This section documents all the keywords that may appear in the lambda-list (argument list) of a function, a macro, or a special form. See the section "Functions: Evaluation". Some of them are allowed everywhere, while others are only allowed in one of these contexts; those are so indicated.

lambda-list-keywords *Variable*

The value of this variable is a list of all of the allowed "&" keywords. Some of these are obsolete and do not do anything; the remaining ones are listed below.

&optional	Separates the required arguments of a function from the optional arguments. See the section "Functions: Evaluation".
&rest	Separates the required and optional arguments of a function from the rest argument. There may be only one rest argument. See the section "Functions: Evaluation". That section contains full information about rest arguments.
&key	Separates the positional arguments and rest argument of a function from the keyword arguments. See the section "Functions: Evaluation".
&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 do not 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.
&functional	Preceding an argument, tells the compiler that the value of this argument will be a function. When a caller of this function is compiled, if it passes a quoted constant argument that looks like a function (a list beginning with the symbol **lambda**) the compiler will know that it is intended to be a function rather than a list that happens to start with that symbol, and will compile it.
&quote	Declares that the following arguments are not to be evaluated. This is how you create a special function. The caveats about special forms are in another section. See the section "Kinds of Functions".

&eval	Turns off a preceding **&quote** for the arguments which follow.
&list-of	This is for macros defined by **defmacro** only. See the section "Advanced Features of **defmacro**".
&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. See the section "Advanced Features of **defmacro**".

1.8 How Programs Manipulate Definitions

fdefine *function-spec definition* &optional *(carefully* **nil***) (no-query* **nil***)* *Function*

This is the primitive that **defun** and everything else in the system use to change the definition of a function spec. If *carefully* is non-**nil**, which it usually should be, then only the basic definition is changed, the previous basic definition is saved if possible (see **undefun**), and any encapsulations of the function such as tracing and advice are carried over from the old definition to the new definition. *carefully* also causes the user to be queried if the function spec is being redefined by a file different from the one that defined it originally. However, this warnings is suppressed if either the argument *no-query* is non-**nil**, or if the global variable **inhibit-fdefine-warnings** is **t**.

If **fdefine** is called while a file is being loaded, it records what file the function definition came from so that the editor can find the source code.

If *function-spec* was already defined as a function, and *carefully* is non-**nil**, the function-spec's **:previous-definition** property is used to save the previous definition. If the previous definition is an interpreted function, it is also saved on the **:previous-expr-definition** property. These properties are used by the **undefun** function, which restores the previous definition, and the **uncompile** function, which restores the previous interpreted definition. The properties for different kinds of function specs are stored in different places; when a function spec is a symbol its properties are stored on the symbol's property list.

defun and the other function-defining special forms all supply **t** for *carefully* and **nil** or nothing for *no-query*. Operations that construct encapsulations, such as **trace**, are the only ones that use **nil** for *carefully*.

inhibit-fdefine-warnings *Variable*

This variable is normally **nil**. Setting it to **t** prevents **fdefine** from warning you and asking about questionable function definitions such as a function being redefined by a different file than defined it originally, or a symbol that belongs to one package being defined by a file that belongs to a different package. Setting it to **:just-warn** allows the warnings to be printed out, but prevents the queries from happening; it assumes that your answer is "yes", that is, that it is all right to redefine the function.

record-source-file-name *function-spec* &optional *(type* **'defun**) *no-query* — *Function*

record-source-file-name associates the definition of a function with its source files, so that tools such as Edit Definition (m-.) can find the source file of a function. It also detects when two different files both try to define the same function, and warns the user.

record-source-file-name is called automatically by **defun**, **defmacro**, **defstruct**, **defflavor**, and other such defining special forms. Normally you do not invoke it explicitly. If you have your own defining macro, however, that does not expand into one of the above, then you can make its expansion include a **record-source-file-name** form.

function-spec	The function spec for the entity being defined.
type	The type of entity being defined, with **defun** as the default. *type* can be any symbol, typically the name of the corresponding special form for defining the entity. Some standard examples: defun defvar defflavor defstruct Both macros and substs are subsumed under the type defun, because you cannot have a function named **x** in one file and a macro named **x** in another file.
no-query	Controls queries about redefinitions. **t** means to suppress queries about redefining. The default value of *no-query* depends on the value of **inhibit-fdefine-warnings**. When **inhibit-fdefine-warnings** is **t**, *no-query* is **t**; otherwise it is **nil**. Regardless of the value for *no-query*, queries are suppressed when the definition is happening in a patch file.

You cannot specify the source file name with this function. The function is always associated with the pathname for the file being loaded (**fdefine-file-pathname**).

When redefining functions, some users try to avoid redefinition warnings and queries by using the form **(remprop** *symbol* **':source-file-name)**. The preferred way to do this is to use the form **(record-source-file-name** *function-spec* **'defun t)**. The former method causes the system to forget both the original definition and other definitions for the same symbol (as a variable, flavor, structure, and so forth). **record-source-file-name** lets the system know that the function is defined in two places, and it avoids redefinition warnings and queries.

Of course, if you are redefining something other than a function, use the appropriate definition type symbol instead of **defun** as the second argument to **record-source-file-name**. For example, if you are redefining a flavor, use **defflavor** as the second argument.

sys:fdefine-file-pathname *Variable*

While loading a file, this is the generic-pathname for the file. The rest of the time it is **nil**. **fdefine** uses this to remember what file defines each function.

sys:function-parent *function-spec* *Function*

When a symbol's definition is produced as the result of macro expansion of a source definition, so that the symbol's definition does not appear textually in the source, the editor cannot find it. The accessor, constructor, and alterant macros produced by a **defstruct** are an example of this. The **sys:function-parent** declaration can be inserted in the source definition to record the name of the outer definition of which it is a part.

The declaration consists of the following:

(sys:function-parent *name type***)**

name is the name of the outer definition. *type* is its type, which defaults to **defun.** (This is the same type as in **record-source-file-name**; it is usually the name of the defining special form.)

You can define the type of an entity being defined:

```
(defprop feature "Feature" si:definition-type-name)
(defprop defun "Function" si:definition-type-name)
```

sys:function-parent is a function related to the declaration. It takes a function spec and returns **nil** or another function spec. The first function spec's definition is contained inside the second function spec's definition. The second value is the type of definition.

Two examples:

```
(defsubst foo (x y)
  (declare (sys:function-parent bar))
  ...)

(defmacro defxxx (name ...)
  '(local-declare ((sys:function-parent ,name defxxx))
     (defmacro ...)
     (defmacro ...)
  ))
```

Using the sys:function-parent declaration

A *definition* is a Lisp expression that appears in a source program file and has a name by which a user would like to refer to it. Definitions come in a variety of types. The main point of definition types is that two definitions with the same name and different types can exist simultaneously, but two definitions with the same name and the same type redefine each other when evaluated. Some examples of definition type symbols and special forms that define such definitions are:

Type symbol	*Type name in English*	*Special form names*
defun	function	**defun**, **defmacro**, **defmethod**
defvar	variable	**defvar**, **defconst**, **defconstant**
defflavor	flavor	**defflavor**
defstruct	structure	**defstruct**

Things to note: More than one special form can define a given kind of definition. The name of the most representative special form is typically chosen as the type symbol. This symbol typically has a **si:definition-type-name** property of a string that acts as a prettier form of the name for people to read. **record-source-file-name** and related functions take a name and a type symbol as arguments. The editor understands certain definition-making special forms, and knows how to parse them to get out the name and the type. This mechanism has not yet been made user-extensible. Currently the editor assumes that any top-level form it does not know about that starts with "(def" must be defining a function (a definition of type **defun**) and assumes that the cadr of that form is the name of the function. Heuristics appropriate for **defun** are applied to this name if it is a list. In general, a definition whose name is not a symbol and whose type is not **defun** does not work properly. This will be fixed in a future release.

The declaration **sys:function-parent** is of interest to users. The function with the same name is probably not of interest to users; it is part of the mechanism by which the Zmacs command Edit Definition (m-.) figures out what file to look in.

Example:

We have functions called "frobulators" that are stored on the property list of symbols and require some special bindings wrapped around their bodies. Frobulator definitions are not considered function definitions, because the name of the frobulator does not become defined as a Lisp function. Indeed, we could have a frobulator named **list** and Lisp's **list** function would continue to work. Instead we make a new definition type.

```
(defmacro define-frobulator (name arg-list &body body)
  `(progn 'compile
          (add-to-list-of-known-frobulators ',name)
          (record-source-file-name ',name 'define-frobulator)
          (defun (:property ,name frobulator) (self ,@arg-list)
            (declare (sys:function-parent ,name define-frobulator))
            (let (,(make-frobulator-bindings name arg-list))
              ,@body))))

(defprop define-frobulator "Frobulator" si:definition-type-name)
```

Here we would tell the editor how to parse **define-frobulator** if its parser were user-extensible. Because it is not, we rely on its heuristics to make m-. work adequately for frobulators.

Next we define a frobulator. This is not an interesting definition, for we do not actually know what the word "frobulate" means. We could always recast this example as a symbolic differentiator: We would define the + frobulator to return a list of + and the frobulations of the arguments, the * frobulator to return sums of products of factors and derivatives of factors, and so forth.

```
(define-frobulator list ()
  (frobulate-any-number-of-args self))
```

In **define-frobulator**, we call **record-source-file-name** so that when a file containing frobulator definitions is loaded, we will know what file those definitions came from. Inside the function that is generated, we include a function-parent declaration because no definition of that function is apparent in any source file. The system will take care of doing
(record-source-file-name '(:property list frobulator) 'defun), as it always does when a function definition is loaded. Suppose an error occurs in a frobulator function — in the **list** example above, we might try to call **frobulate-any-number-of-args**, which is not defined — and we use the Debugger c-E command to edit the source. This will be trying to edit **(:property list frobulator)**, the function in which we were executing. The definition that defines this function does not have that name; rather, it is named **list** and has type **define-frobulator**. The **sys:function-parent** declaration enables the editor to know that fact.

If your definition-making special form and your definition type symbol do not have the same name, you should define the special form's **zwei:definition-function-spec** property to be the definition type symbol. This helps the editor parse such special forms.

For another example, more complicated but real, use **mexp** or the Zmacs command Macro Expand Expression (c-sh-M) to look at the macro expansion of:

```
(defstruct (foo :conc-name) one two)
```

The macro **sys:defsubst-with-parent** that it calls is just **defsubst** with a **sys:function-parent** declaration inside. It exists only because of a bug in an old implementation of **defsubst** that made doing it the straightforward way not work.

fset-carefully *symbol definition* &optional *force-flag* *Function*

This function is obsolete. It is equivalent to:

(fdefine *symbol definition* t *force-flag*)

fdefinedp *function-spec* *Function*

This returns **t** if *function-spec* has a definition, or **nil** if it does not.

fdefinition *function-spec* *Function*

This returns *function-spec*'s definition. If it has none, an error occurs.

sys:fdefinition-location *function-spec* *Function*

This returns a locative pointing at the cell that contains *function-spec*'s definition. For some kinds of function specs, though not for symbols, this can cause data structure to be created to hold a definition. For example, if *function-spec* is of the **:property** kind, then an entry may have to be added to the property list if it isn't already there. In practice, you should write **(locf (fdefinition** *function-spec***))** instead of calling this function explicitly.

fundefine *function-spec* *Function*

Removes the definition of *function-spec*. For symbols this is equivalent to **fmakunbound**. If the function is encapsulated, **fundefine** removes both the basic definition and the encapsulations. Some types of function specs (**:location** for example) do not implement **fundefine**. **fundefine** on a **:within** function spec removes the replacement of *function-to-affect*, putting the definition of *within-function* back to its normal state. **fundefine** on a **:method** function spec removes the method completely, so that future messages will be handled by some other method. See the document *Objects, Message Passing, and Flavors.*

si:function-spec-get *function-spec indicator* *Function*

Returns the value of the *indicator* property of *function-spec*, or **nil** if it doesn't have such a property.

si:function-spec-putprop *function-spec value indicator* *Function*

Gives *function-spec* an *indicator* property whose value is *value*.

undefun *function-spec* *Function*

If *function-spec* has a saved previous basic definition, this interchanges the current and previous basic definitions, leaving the encapsulations alone. This undoes the effect of a **defun, compile,** and so on. (See the function **uncompile.**)

1.9 How Programs Examine Functions

These functions take a function as argument and return information about that function. Some also accept a function spec and operate on its definition. The others do not accept function specs in general but do accept a symbol as standing for its definition. (Note that a symbol is a function as well as a function spec).

documentation *function* *Function*

Given a function or a function spec, this finds its documentation string, which is stored in various different places depending on the kind of function. If there is no documentation, **nil** is returned.

debugging-info *function* *Function*

This returns the debugging info alist of *function*, or **nil** if it has none.

arglist *function* &optional *real-flag* *Function*

arglist is given a function or a function spec, and returns its best guess at the nature of the function's lambda-list. It can also return a second value which is a list of descriptive names for the values returned by the function.

If *function* is a symbol, **arglist** of its function definition is used.

If the *function* is an actual lambda-expression, its cadr, the lambda-list, is returned. But if *function* is compiled, **arglist** attempts to reconstruct the lambda-list of the original definition, using whatever debugging information was saved by the compiler. Sometimes the actual names of the bound variables are not available, and **arglist** uses the symbol **si:*unknown*** for these. Also, sometimes the initialization of an optional parameter is too complicated for **arglist** to reconstruct; for these it returns the symbol **si:*hairy***.

Some functions' real argument lists are not what would be most descriptive to a user. A function may take a &rest argument for technical reasons even though there are standard meanings for the first element of that argument. For such cases, the definition of the function can specify, with a local declaration, a value to be returned when the user asks about the argument list. Example:

```
(defun foo (&rest rest-arg)
  (declare (arglist x y &rest z))
  .....)
```

real-flag allows the caller of **arglist** to say that the real argument list should be used even if a declared argument list exists. Note that while normally **declares** are only for the compiler's benefit, this kind of **declare** affects all functions, including interpreted functions.

arglist cannot be relied upon to return the exactly correct answer, since some of the information may have been lost. Programs interested in how many and what kind of arguments there are should use **args-info** instead. In general **arglist** is to be used for documentation purposes, not for reconstructing the original source code of the function.

When a function returns multiple values, it is useful to give the values names so that the caller can be reminded which value is which. By means of a **return-list** declaration in the function's definition, entirely analogous to the **arglist** declaration above, you can specify a list of mnemonic names for the returned values. This list will be returned by **arglist** as the second value.

```
(arglist 'arglist)
  => (function &optional real-flag) and (arglist return-list)
```

args-info *function* *Function*

args-info returns a fixnum called the "numeric argument descriptor" of the *function*, which describes the way the function takes arguments. This descriptor is used internally by the microcode, the evaluator, and the compiler. *function* can be a function or a function spec.

The information is stored in various bits and byte fields in the fixnum, which are referenced by the symbolic names shown below. By the usual Lisp Machine convention, those starting with a single "%" are bit-masks (meant to be **logand**ed or **bit-test**ed with the number), and those starting with "%%" are byte descriptors (meant to be used with **ldb** or **ldb-test**).

Here are the fields:

%%arg-desc-min-args
This is the minimum number of arguments that may be passed to this function, that is, the number of "required" parameters.

%%arg-desc-max-args
This is the maximum number of arguments that may be passed to this function, that is, the sum of the number of "required" parameters and the number of "optional" parameters. If there is a rest argument, this is not really the maximum number of arguments that may be passed; an arbitrarily large number of arguments is permitted, subject to limitations on the maximum size of a stack frame (about 200 words).

%arg-desc-evaled-rest
If this bit is set, the function has a "rest" argument, and it is not "quoted".

%arg-desc-quoted-rest
If this bit is set, the function has a "rest" argument, and it is "quoted". Most special forms have this bit.

%arg-desc-fef-quote-hair

If this bit is set, there are some quoted arguments other than the "rest" argument (if any), and the pattern of quoting is too complicated to describe here. The ADL (Argument Description List) in the FEF should be consulted. This is only for special forms.

%arg-desc-interpreted
This function is not a compiled-code object, and a numeric argument descriptor cannot be computed. Usually **args-info** will not return this bit, although **%args-info** will.

%arg-desc-fef-bind-hair
There is argument initialization, or something else too complicated to describe here. The ADL (Argument Description List) in the FEF should be consulted.

Note that **%arg-desc-quoted-rest** and **%arg-desc-evaled-rest** cannot both be set.

%args-info *function* *Function*

This is an internal function; it is like **args-info** but does not work for interpreted functions. Also, *function* must be a function, not a function spec. It exists because it has to be in the microcode anyway, for **apply** and the basic function-calling mechanism.

1.10 Encapsulations

The definition of a function spec actually has two parts: the *basic definition*, and *encapsulations*. The basic definition is what functions like **defun** create, and encapsulations are additions made by **trace** or **advise** to the basic definition. The purpose of making the encapsulation a separate object is to keep track of what was made by **defun** and what was made by **trace**. If **defun** is done a second time, it replaces the old basic definition with a new one while leaving the encapsulations alone.

Only advanced users should ever need to use encapsulations directly via the primitives explained in this section. The most common things to do with encapsulations are provided as higher-level, easier-to-use features: **trace** and **advise**.

The way the basic definition and the encapsulations are defined is that the actual definition of the function spec is the outermost encapsulation; this contains the next encapsulation, and so on. The innermost encapsulation contains the basic definition. The way this containing is done is as follows. An encapsulation is actually a function whose debugging info alist contains an element of the form:

`(si:encapsulated-definition` *uninterned-symbol encapsulation-type*`)`

The presence of such an element in the debugging info alist is how you recognize a function to be an encapsulation. An encapsulation is usually an interpreted function (a list starting with **named-lambda**) but it can be a compiled function also, if the application that created it wants to compile it.

uninterned-symbol's function definition is the thing that the encapsulation contains, usually the basic definition of the function spec. Or it can be another encapsulation, which has in it another debugging info item containing another uninterned symbol. Eventually you get to a function that is not an encapsulation; it does not have the sort of debugging info item that encapsulations all have. That function is the basic definition of the function spec.

Literally speaking, the definition of the function spec is the outermost encapsulation, period. The basic definition is not the definition. If you are asking for the definition of the function spec because you want to apply it, the outermost encapsulation is exactly what you want. But the basic definition can be found mechanically from the definition, by following the debugging info alists. So it makes sense to think of it as a part of the definition. In regard to the function-defining special forms such as **defun**, it is convenient to think of the encapsulations as connecting between the function spec and its basic definition.

An encapsulation is created with the macro **si:encapsulate**.

si:encapsulate *Macro*

A call to **si:encapsulate** looks like:

```
(si:encapsulate function-spec outer-function type
                body-form
                extra-debugging-info)
```

All the subforms of this macro are evaluated. In fact, the macro could almost be replaced with an ordinary function, except for the way *body-form* is handled.

function-spec evaluates to the function spec whose definition the new encapsulation should become. *outer-function* is another function spec, which should often be the same one. Its only purpose is to be used in any error messages from **si:encapsulate**.

type evaluates to a symbol that identifies the purpose of the encapsulation; it says what the application is. For example, it could be **advise** or **trace**. The list of possible types is defined by the system because encapsulations are supposed to be kept in an order according to their type. See the variable **si:encapsulation-standard-order**. *type* should have an **si:encapsulation-grind-function** property that tells **grindef** what to do with an encapsulation of this type.

body-form is a form that evaluates to the body of the encapsulation-definition, the code to be executed when it is called. Backquote is typically used for this expression. See the section "Backquote". **si:encapsulate** is a macro because, while *body* is being evaluated, the variable **si:encapsulated-function** is bound to a list of the form **(function** *uninterned-symbol***)**, referring to the uninterned symbol used to hold the prior definition of *function-spec*. If **si:encapsulate** were a function, *body-form* would just get evaluated normally by the evaluator before **si:encapsulate** ever got invoked, and so there would be no opportunity to bind **si:encapsulated-function**. The form *body-form* should contain **(apply si:encapsulated-function arglist)** somewhere if the encapsulation is to live up to its name and truly serve to encapsulate the original definition. (The variable **arglist** is bound by some of the code that the **si:encapsulate** macro produces automatically. When the body of the encapsulation is run, **arglist**'s value will be the list of the arguments that the encapsulation received.)

extra-debugging-info evaluates to a list of extra items to put into the debugging info alist of the encapsulation function (besides the one starting with **si:encapsulated-definition** that every encapsulation must have). Some applications find this useful for recording information about the encapsulation for their own later use.

When a special function is encapsulated, the encapsulation is itself a special function with the same argument quoting pattern. (Not all quoting patterns can be handled; if a particular special form's quoting pattern cannot be handled, **si:encapsulate** signals an error.) Therefore, when the outermost encapsulation is started, each argument has been evaluated or not as appropriate. Because each encapsulation calls the prior definition with **apply**, no further evaluation takes place, and the basic definition of the special form also finds the arguments evaluated or not as appropriate. The basic definition may call **eval** on some of these arguments or parts of them; the encapsulations should not.

Macros cannot be encapsulated, but their expander functions can be; if the definition of *function-spec* is a macro, then **si:encapsulate** automatically encapsulates the expander function instead. In this case, the definition of the uninterned symbol is the original macro definition, not just the original expander function. It would not work for the encapsulation to apply the macro definition. So during the evaluation of *body-form*, **si:encapsulated-function** is bound to the form **(cdr (function** *uninterned-symbol***))**, which extracts the expander function from the prior definition of the macro.

Because only the expander function is actually encapsulated, the encapsulation does not see the evaluation or compilation of the expansion itself. The value returned by the encapsulation is the expansion of the macro call, not the value computed by the expansion.

It is possible for one function to have multiple encapsulations, created by different subsystems. In this case, the order of encapsulations is independent of the order in which they were made. It depends instead on their types. All possible encapsulation types have a total order and a new encapsulation is put in the right place among the existing encapsulations according to its type and their types.

si:encapsulation-standard-order *Variable*

The value of this variable is a list of the allowed encapsulation types, in the order that the encapsulations are supposed to be kept in (innermost encapsulations first). If you want to add new kinds of encapsulations, you should add another symbol to this list. Initially its value is:

```
(advise trace si:rename-within)
```

advise encapsulations are used to hold advice. **trace** encapsulations are used for implementing tracing. **si:rename-within** encapsulations are used to record the fact that function specs of the form **(:within** *within-function altered-function***)** have been defined. The encapsulation goes on *within-function*. See the section "Rename-within Encapsulations".

Every symbol used as an encapsulation type must be on the list **si:encapsulation-standard-order**. In addition, it should have an **si:encapsulation-grind-function** property whose value is a function that **grindef** will call to process encapsulations of that type. This function need not take care of printing the encapsulated function, because **grindef** will do that itself. But it should print any information about the encapsulation itself that the user ought to see. Refer to the code for the grind function for **advise** to see how to write one. See the special form **advise**.

To find the right place in the ordering to insert a new encapsulation, it is necessary to parse existing ones. This is done with the function **si:unencapsulate-function-spec**.

si:unencapsulate-function-spec *function-spec* &optional *encapsulation-types* *Function*

This takes one function spec and returns another. If the original function spec is undefined, or has only a basic definition (that is, its definition is not an encapsulation), then the original function spec is returned unchanged.

If the definition of *function-spec* is an encapsulation, then its debugging info is examined to find the uninterned symbol that holds the encapsulated definition, and also the encapsulation type. If the encapsulation is of a type that is to be skipped over, the uninterned symbol replaces the original function spec and the process repeats.

The value returned is the uninterned symbol from inside the last encapsulation skipped. This uninterned symbol is the first one that does not have a definition that is an encapsulation that should be skipped. Or the value can be *function-spec* if *function-spec*'s definition is not an encapsulation that should be skipped.

The types of encapsulations to be skipped over are specified by *encapsulation-types*. This can be a list of the types to be skipped, or **nil**, meaning skip all encapsulations (this is the default). Skipping all encapsulations means returning the uninterned symbol that holds the basic definition of *function-spec*. That is, the *definition* of the function spec returned is the *basic definition* of the function spec supplied. Thus:

```
(fdefinition (si:unencapsulate-function-spec 'foo))
```

returns the basic definition of **foo**, and

```
(fdefine (si:unencapsulate-function-spec 'foo) 'bar)
```

sets the basic definition (just like using **fdefine** with *carefully* supplied as **t**).

encapsulation-types can also be a symbol, which should be an encapsulation type; then we skip all types that are supposed to come outside of the specified type. For example, if *encapsulation-types* is **trace**, then we skip all types of encapsulations that come outside of **trace** encapsulations, but we do not skip **trace** encapsulations themselves. The result is a function spec that is where the **trace** encapsulation ought to be, if there is one. Either the definition of this function spec is a **trace** encapsulation, or there is no **trace** encapsulation anywhere in the definition of *function-spec*, and this function spec is where it would belong if there were one. For example:

```
(let ((tem (si:unencapsulate-function-spec spec 'trace)))
  (and (eq tem (si:unencapsulate-function-spec tem '(trace)))
       (si:encapsulate tem spec 'trace '(...body...))))
```

finds the place where a **trace** encapsulation ought to go, and makes one unless there is already one there.

```
(let ((tem (si:unencapsulate-function-spec spec 'trace)))
  (fdefine tem (fdefinition (si:unencapsulate-function-spec
                                tem '(trace)))))
```

eliminates any **trace** encapsulation by replacing it by whatever it encapsulates. (If there is no **trace** encapsulation, this code changes nothing.)

These examples show how a subsystem can insert its own type of encapsulation in the proper sequence without knowing the names of any other types of encapsulations. Only the **si:encapsulation-standard-order** variable, which is used by **si:unencapsulate-function-spec**, knows the order.

1.10.1 Rename-within Encapsulations

One special kind of encapsulation is the type **si:rename-within**. This encapsulation goes around a definition in which renamings of functions have been done.

How is this used?

If you define, advise, or trace **(:within foo bar)**, then **bar** gets renamed to **altered-bar-within-foo** wherever it is called from **foo**, and **foo** gets a **si:rename-within** encapsulation to record the fact. The purpose of the encapsulation is to enable various parts of the system to do what seems natural to the user. For example, **grindef** notices the encapsulation, and so knows to print **bar** instead of **altered-bar-within-foo**, when grinding the definition of **foo**.

Also, if you redefine **foo**, or trace or advise it, the new definition gets the same renaming done (**bar** replaced by **altered-bar-within-foo**). To make this work, everyone who alters part of a function definition should pass the new part of the definition through the function **si:rename-within-new-definition-maybe**.

si:rename-within-new-definition-maybe *function-spec new-structure* *Function*

Given *new-structure* that is going to become a part of the definition of *function-spec*, perform on it the replacements described by the **si:rename-within** encapsulation in the definition of *function-spec*, if there is one. The altered (copied) list structure is returned.

It is not necessary to call this function yourself when you replace the basic definition because **fdefine** with *carefully* supplied as **t** does it for you. **si:encapsulate** does this to the body of the new encapsulation. So you only need to call **si:rename-within-new-definition-maybe** yourself if you are rplac'ing part of the definition.

For proper results, *function-spec* must be the outer-level function spec. That is, the value returned by **si:unencapsulate-function-spec** is *not* the right thing to use. It will have had one or more encapsulations stripped off, including the **si:rename-within** encapsulation if any, and so no renamings will be done.

2. Closures

A *closure* is a type of Lisp functional object useful for implementing certain advanced access and control structures. Closures give you more explicit control over the environment, by allowing you to save the environment created by the entering of a dynamic contour (that is, a **lambda**, **do**, **prog**, **progv**, **let**, or any of several other special forms), and then use that environment elsewhere, even after the contour has been exited.

2.1 What a Closure is

We will use a particular view of lambda-binding in this section because it makes it easier to explain what closures do. In this view, when a variable is bound, a new value cell is created for it. The old value cell is saved away somewhere and is inaccessible. Any references to the variable will get the contents of the new value cell, and any **setq**'s will change the contents of the new value cell. When the binding is undone, the new value cell goes away, and the old value cell, along with its contents, is restored.

For example, consider the following sequence of Lisp forms:

```
(setq a 3)

(let ((a 10))
  (print (+ a 6)))

(print a)
```

Initially there is a value cell for **a**, and the **setq** form makes the contents of that value cell be **3**. Then the lambda-combination is evaluated. **a** is bound to **10**: the old value cell, which still contains a **3**, is saved away, and a new value cell is created with **10** as its contents. The reference to **a** inside the **lambda** expression evaluates to the current binding of **a**, which is the contents of its current value cell, namely **10**. So **16** is printed. Then the binding is undone, discarding the new value cell, and restoring the old value cell, which still contains a **3**. The final **print** prints out a **3**.

The form **(closure** *var-list function***)**, where *var-list* is a list of variables and *function* is any function, creates and returns a closure. When this closure is applied to some arguments, all the value cells of the variables on *var-list* are saved away, and the value cells that those variables had *at the time* **closure** *was called* (that is, at the time the closure was created) are made to be the value cells of the symbols. Then *function* is applied to the arguments.

Here is another, lower level explanation. The closure object stores several things inside of it. First, it saves the *function*. Secondly, for each variable in *var-list*, it remembers what that variable's value cell was when the closure was created. Then when the closure is called as a function, it first temporarily restores the value cells it has remembered inside the closure, and then applies *function* to the same arguments to which the closure itself was applied. When the function returns, the value cells are restored to be as they were before the closure was called.

Now, if we evaluate the form:

```
(setq a
      (let ((x 3))
        (closure '(x) 'frob)))
```

what happens is that a new value cell is created for **x**, and its contents is a fixnum **3**. Then a closure is created, which remembers the function **frob**, the symbol **x**, and that value cell. Finally the old value cell of **x** is restored, and the closure is returned. Notice that the new value cell is still around, because it is still known about by the closure. When the closure is applied, say by doing **(funcall a 7)**, this value cell will be restored and the value of **x** will be **3** again. If **frob** uses **x** as a free variable, it will see **3** as the value.

A closure can be made around any function, using any form that evaluates to a function. The form could evaluate to a lambda expression, as in **'(lambda () x)**, or to a compiled function, as would **(function (lambda () x))**. In the example above, the form is **'frob** and it evaluates to the symbol **frob**. A symbol is also a good function. It is usually better to close around a symbol that is the name of the desired function, so that the closure points to the symbol. Then, if the symbol is redefined, the closure will use the new definition. If you actually prefer that the closure continue to use the old definition that was current when the closure was made, then close around the definition of the symbol rather than the symbol itself. In the above example, that would be done by:

```
(closure '(x) (function frob))
```

Because of the way closures are implemented, the variables to be closed over must not get turned into "local variables" by the compiler. Therefore, all such variables must be declared special. This can be done with an explicit **declare**, with a special form such as **defvar**, or with **let-closed**. In simple cases, a **local-declare** around the binding will do the job. Usually the compiler can tell when a special declaration is missing, but in the case of making a closure the compiler detects this after already acting on the assumption that the variable is local, by which time it is too late to fix things. The compiler will warn you if this happens.

In Zetalisp's implementation of closures, lambda-binding never really allocates any storage to create new value cells. Value cells are created only by the **closure** function itself, when they are needed. Thus, implementors of large systems need not worry about storage allocation overhead from this mechanism if they are not using closures.

Zetalisp closures are not closures in the true sense, as they do not save the whole variable-binding environment; however, most of that environment is irrelevant, and the explicit declaration of which variables are to be closed allows the implementation to have high efficiency. They also allow you to explicitly choose for each variable whether it is to be bound at the point of call or bound at the point of definition (for example, creation of the closure), a choice which is not conveniently available in other languages. In addition, the program is clearer because the intended effect of the closure is made manifest by listing the variables to be affected.

The implementation of closures (which is not usually necessary for you to understand) involves two kinds of value cells. Every symbol has an *internal value cell*, which is where its value is normally stored. When a variable is closed over by a closure, the variable gets an *external value cell* to hold its value. The external value cells behave according to the lambda-binding model used earlier in this section. The value in the external value cell is found through the usual access mechanisms (such as evaluating the symbol, calling **symeval**, and so on), because the internal value cell is made to contain an invisible pointer to the external value cell currently in effect. A symbol will use such an invisible pointer whenever its current value cell is a value cell that some closure is remembering; at other times, there will not be an invisible pointer, and the value will just reside in the internal value cell.

2.2 Examples of the Use of Closures

One thing we can do with closures is to implement a *generator*, which is a kind of function that is called successively to obtain successive elements of a sequence. We will implement a function **make-list-generator**, which takes a list and returns a generator that will return successive elements of the list. When it gets to the end it should return **nil**.

The problem is that in between calls to the generator, the generator must somehow remember where it is up to in the list. Since all of its bindings are undone when it is exited, it cannot save this information in a bound variable. It could save it in a global variable, but the problem is that if we want to have more than one list generator at a time, they will all try to use the same global variable and get in each other's way.

Here is how we can use closures to solve the problem:

```
(defun make-list-generator (l)
    (declare (special l))
    (closure '(l)
            (function (lambda ()
                        (prog1 (car l)
                               (setq l (cdr l)))))))
```

Now we can make as many list generators as we like; they will not get in each other's way because each has its own (external) value cell for **l**. Each of these value cells was created when the **make-list-generator** function was entered, and the value cells are remembered by the closures.

The following form uses closures to create an advanced accessing environment:

```
(declare (special a b))

(defun foo ()
   (setq a 5))

(defun bar ()
   (cons a b))

(let ((a 1)
      (b 1))
   (setq x (closure '(a b) 'foo))
   (setq y (closure '(a b) 'bar)))
```

When the **let** is entered, new value cells are created for the symbols **a** and **b**, and two closures are created that both point to those value cells. If we do **(funcall x)**, the function **foo** will be run, and it will change the contents of the remembered value cell of **a** to **5**. If we then do **(funcall y)**, the function **bar** will return **(5 . 1)**. This shows that the value cell of **a** seen by the closure **y** is the same value cell seen by the closure **x**. The top-level value cell of **a** is unaffected.

2.3 Closure-manipulating Functions

closure *var-list function* *Function*

This creates and returns a closure of *function* over the variables in *var-list*. Note that all variables on *var-list* must be declared special if the function is to compile correctly.

To test whether an object is a closure, use the **closurep** predicate. See the section "Predicates". The **typep** function will return the symbol **closure** if given a closure. **(typep** *x* **'closure)** is equivalent to **(closurep** *x***)**.

symeval-in-closure *closure symbol* *Function*

This returns the binding of *symbol* in the environment of *closure*; that is, it returns what you would get if you restored the value cells known about by *closure* and then evaluated *symbol*. This allows you to "look around inside" a closure. If *symbol* is not closed over by *closure*, this is just like **symeval**.

set-in-closure *closure symbol x* *Function*

This sets the binding of *symbol* in the environment of *closure* to *x*; that is, it does what would happen if you restored the value cells known about by *closure* and then set *symbol* to *x*. This allows you to change the contents of the value cells known about by a closure. If *symbol* is not closed over by *closure*, this is just like **set**.

locate-in-closure *closure symbol* *Function*
This returns the location of the place in *closure* where the saved value of *symbol* is stored. An equivalent form is
(locf (symeval-in-closure *closure symbol***))**.

closure-alist *closure* *Function*
Returns an alist of **(***symbol* **.** *value***)** pairs describing the bindings which the closure performs when it is called. This list is not the same one that is actually stored in the closure; that one contains pointers to value cells rather than symbols, and **closure-alist** translates them back to symbols so you can understand them. As a result, clobbering part of this list will not change the closure.

closure-function *closure* *Function*
Returns the closed function from *closure*. This is the function that was the second argument to **closure** when the closure was created.

let-closed *((variable value)...) function* *Special Form*
When using closures, it is very common to bind a set of variables with initial values, and then make a closure over those variables. Furthermore, the variables must be declared as "special" for the compiler. **let-closed** is a special form that does all of this. It is best described by example:

```
(let-closed ((a 5) b (c 'x))
   (function (lambda () ...)))
```

macro-expands into

```
(local-declare ((special a b c))
   (let ((a 5) b (c 'x))
      (closure '(a b c)
         (function (lambda () ...)))))
```

copy-closure *closure* *Function*
Creates and returns a new closure by copying *closure*, which should be a closure or an entity. **copy-closure** generates new external value cells for each variable in the closure and initializes their contents from the external value cells of *closure*.

closure-variables *closure* *Function*
Creates and returns a list of all of the variables in *closure*, which should be a closure or an entity. It returns a copy of the list that was passed as the first argument to **closure** when *closure* was created.

boundp-in-closure *closure symbol* *Function*

Returns **t** if *symbol* is bound in the environment of *closure*; that is, it does what **boundp** would do if you restored the value cells known about by *closure*. If *symbol* is not closed over by *closure*, this is just like **boundp**.

makunbound-in-closure *closure symbol* *Function*

Makes *symbol* be unbound in the environment of *closure*; that is, it does what **makunbound** would do if you restored the value cells known about by *closure*. If *symbol* is not closed over by *closure*, this is just like **makunbound**.

A note about all of the *xxx*-**in-closure** functions (**set-**, **symeval-**, **boundp-**, and **makunbound-**): if the variable is not directly closed over, the variable's value cell from the global environment is used. That is, if closure A closes over closure B, *xxx*-**in-closure** of A does not notice any variables closed over by B.

A note about **closure-alist**: if any variable in the closure is unbound, this function signals an error. It has been changed to return the variables in forward order, rather than in reverse order as it used to. Use of this function is not recommended.

2.4 Entities

An entity is almost the same thing as a closure; the data type is nominally different but an entity behaves just like a closure when applied. The difference is that some system functions, such as **print**, operate on them differently. When **print** sees a closure, it prints the closure in a standard way. When **print** sees an entity, it calls the entity to ask the entity to print itself.

To some degree, entities are made obsolete by flavors. See the document *Objects, Message Passing, and Flavors*. The use of entities as message-receiving objects is explained in another section. See the section "Entities".

entity *variable-list function* *Function*

(LM-2 only) Returns a newly constructed entity. This function is just like the function **closure** except that it returns an entity instead of a closure.

To test whether an object is an entity, use the **entityp** predicate. See the section "Predicates". The functions **symeval-in-closure**, **closure-alist**, **closure-function**, and so on also operate on entities.

Index

symbolics™

MAC Macros

Cambridge, Massachusetts

Macros

# 990068

March 1984

This document corresponds to Release 5.0.

This document was prepared by the Documentation Group of Symbolics, Inc.

Printing year and number: 87 86 85 84 9 8 7 6 5 4 3 2 1

Table of Contents

1. Introduction to Macros

If **eval** is handed a list whose car is a symbol, then **eval** inspects the definition of the symbol to find out what to do. If the definition is a cons, and the car of the cons is the symbol **macro**, then the definition (that is, that cons) is called a *macro*. The cdr of the cons should be a function of one argument. **eval** applies the function to the form it was originally given, takes whatever is returned, and evaluates that in lieu of the original form.

Here is a simple example. Suppose the definition of the symbol **first** is:

```
(macro lambda (x)
       (list 'car (cadr x)))
```

This thing is a macro: it is a cons whose car is the symbol **macro**. What happens if we try to evaluate a form **(first '(a b c))**? Well, **eval** sees that it has a list whose car is a symbol (namely, **first**), so it looks at the definition of the symbol and sees that it is a cons whose car is **macro**; the definition is a macro.

eval takes the cdr of the cons, which is supposed to be the macro's *expander function*, and calls it providing as an argument the original form that **eval** was handed. So it calls **(lambda (x) (list 'car (cadr x)))** with argument **(first '(a b c))**. Whatever this returns is the *expansion* of the macro call. It will be evaluated in place of the original form.

In this case, **x** is bound to **(first '(a b c))**, **(cadr x)** evaluates to **'(a b c)**, and **(list 'car (cadr x))** evaluates to **(car '(a b c))**, which is the expansion. **eval** now evaluates the expansion. **(car '(a b c))** returns **a**, and so the result is that **(first '(a b c))** returns **a**.

What have we done? We have defined a macro called **first**. What the macro does is to *translate* the form to some other form. Our translation is very simple — it just translates forms that look like **(first** *x***)** into **(car** *x***)**, for any form *x*. We can do much more interesting things with macros, but first we will show how to define a macro.

macro *Special Form*

The primitive special form for defining macros is **macro**. A macro definition looks like this:

```
(macro name (arg)
  body)
```

name can be any function spec. *arg* must be a variable. *body* is a sequence of Lisp forms that expand the macro; the last form should return the expansion.

To define our **first** macro, we would say:

```
(macro first (x)
    (list 'car (cadr x)))
```

Here are some more simple examples of macros. Suppose we want any form that looks like **(addone** *x***)** to be translated into **(plus 1** *x***)**. To define a macro to do this we would say

```
(macro addone (x)
   (list 'plus '1 (cadr x))))
```

Now say we wanted a macro that would translate **(increment** *x***)** into **(setq** *x* **(1+** *x***)**. This would be:

```
(macro increment (x)
    (list 'setq (cadr x) (list '1+ (cadr x)))
```

Of course, this macro is of limited usefulness. The reason is that the form in the *cadr* of the **increment** form had better be a symbol. If you tried **(increment (car x))**, it would be translated into **(setq (car x) (1+ (car x)))**, and **setq** would complain. (If you're interested in how to fix this problem: See the macro **setf**. However, this is irrelevant to how macros work.)

You can see from this discussion that macros are very different from functions. A function would not be able to tell what kind of subforms are around in a call to itself; they get evaluated before the function ever sees them. However, a macro gets to look at the whole form and see just what is going on there. Macros are *not* functions; if **first** is defined as a macro, it is not meaningful to apply **first** to arguments. A macro does not take arguments at all; its expander function takes a *Lisp form* and turns it into another *Lisp form*.

The purpose of functions is to *compute*; the purpose of macros is to *translate*. Macros are used for a variety of purposes, the most common being extensions to the Lisp language. For example, Lisp is powerful enough to express many different control structures, but it does not provide every control structure anyone might ever possibly want. Instead, if you want some kind of control structure with a syntax that is not provided, you can translate it into some form that Lisp *does* know about.

For example, you might want a limited iteration construct that increments a variable by one until it exceeds a limit (like the FOR statement of the BASIC language). You might want it to look like:

```
(for a 1 100 (print a) (print (* a a)))
```

To get this, you could write a macro to translate it into:

```
(do a 1 (1+ a) (> a 100) (print a) (print (* a a)))
```

A macro to do this could be defined with:

```
(macro for (x)
  (cons 'do
        (cons (cadr x)
              (cons (caddr x)
                    (cons (list '1+ (cadr x))
                          (cons (list '> (cadr x) (cadddr x))
                                (cddddr x)))))))
```

Now you have defined your own new control structure primitive, and it will act just as if it were a special form provided by Lisp itself.

1.1 Lambda Macros

Lambda macros are similar to regular Lisp macros, except that regular Lisp macros replace, and expand into, Lisp forms, whereas lambda macros replace, and expand into, Lisp functions. They are an advanced feature, used only for certain special language extensions or embedded programming systems.

To understand what lambda macros do, consider how regular Lisp macros work. When the evaluator is given a Lisp form to evaluate, it inspects the car of the form to figure out what to do. If the car is the name of a function, the function is called. But if the car is the name of a macro, the macro is expanded, and the result of the expansion is considered to be a Lisp form and is evaluated. Lambda macros work analogously, but in a different situation. When the evaluator finds that the car of a form is a list, it looks at the car of this list to figure out what to do. If this car is the symbol **lambda**, the list is an ordinary function, and it is applied to its arguments. But if this car is the name of a lambda macro, the lambda macro is expanded, and the result of the expansion is considered to be a Lisp function and is applied to the arguments.

Like regular macros, lambda macros are named by symbols and have a body, which is a function of one argument. To expand the lambda macro, the evaluator applies this body to the entire lambda macro function (the list whose car is the name of the lambda macro), and expects the body to return another function as its value.

Several special forms are provided for dealing with lambda macros. The primitive for defining a new lambda macro is **lambda-macro**; it is analogous with the **macro** special form. For convenience, **deflambda-macro** and **deflambda-macro-displace** are defined; these work like **defmacro** to provide easy parsing of the function into its component parts. The special form **deffunction** creates a new Lisp function that uses a named lambda macro instead of **lambda** in its definition.

lambda-macro *name lambda-list* &body *body* *Special Form*

Analogously with **macro**, defines a lambda macro to be called *name*. *lambda-list* should be a list of one variable, which is bound to the function being expanded. The lambda macro must return a function. Example:

```
(lambda-macro ilisp (x)
  `(lambda (&optional ,@(second x) &rest ignore) . ,(cddr x)))
```

This defines a lambda macro called **ilisp**. After it has been defined, the following list is a valid Lisp function:

```
(ilisp (x y z) (list x y z))
```

The above function takes three arguments and returns a list of them, but all of the arguments are optional and any extra arguments are ignored. (This shows how to make functions that imitate Interlisp functions, in which all arguments are always optional and extra arguments are always ignored.) So, for example:

```
(funcall #'(ilisp (x y z) (list x y z)) 1 2)  =>  (1 2 nil)
```

deflambda-macro *Special Form*

deflambda-macro is like **defmacro**, but defines a lambda macro instead of a normal macro.

deflambda-macro-displace *Special Form*

deflambda-macro-displace is like **defmacro-displace**, but defines a displacing lambda macro instead of a displacing normal macro.

deffunction *function-spec lambda-macro-name lambda-list* &body *body* *Special Form*

Defines a function using an arbitrary lambda macro in place of **lambda**. A **deffunction** form is like a **defun** form, except that the the function spec is immediately followed by the name of the lambda macro to be used. **deffunction** expands the lambda macro immediately, so the lambda macro must already be defined before **deffunction** is used. For example, suppose the **ilisp** lambda macro were defined, as in the example above. Then the following example would define a function called **new-list**, that would use the lambda macro called **ilisp**:

```
(deffunction new-list ilisp (x y z)
  (list x y z))
```

new-list's arguments are optional, and any extra arguments are ignored. Examples:

```
(new-list 1 2) => (1 2 nil)
(new-list 1 2 3 4) -> (1 2 3)
```

Lambda macros can be accessed with the (**:lambda-macro** *name*) function spec.

2. Aids for Defining Macros

The main problem with the definition for the **for** macro is that it is verbose and clumsy. See the section "Introduction to Macros". If it is that hard to write a macro to do a simple specialized iteration construct, one would wonder how anyone could write macros of any real sophistication.

There are two things that make the definition so inelegant. One is that you must write things like "**(cadr x)**" and "**(cddddr x)**" to refer to the parts of the form you want to do things with. The other problem is that the long chains of calls to the **list** and **cons** functions are very hard to read.

Two features are provided to solve these two problems. The **defmacro** macro solves the former, and the "backquote" (‘) reader macro solves the latter.

2.1 defmacro

Instead of referring to the parts of our form by "**(cadr x)**" and such, we would like to give names to the various pieces of the form, and somehow have the **(cadr x)** automatically generated. This is done by a macro called **defmacro**. It is easiest to explain what **defmacro** does by showing an example. Here is how you would write the **for** macro using **defmacro**:

```
(defmacro for (var lower upper . body)
  (cons 'do
        (cons var
              (cons lower
                    (cons (list '1+ var)
                          (cons (list '> var upper)
                                body))))))
```

The **(var lower upper . body)** is a *pattern* to match against the body of the form (to be more precise, to match against the cdr of the argument to the macro's expander function). If **defmacro** tries to match the following two lists:

```
(var lower upper . body)
(a 1 100 (print a) (print (* a a)))
```

var will get bound to the symbol **a**, **lower** to the fixnum **1**, **upper** to the fixnum **100**, and **body** to the list **((print a) (print (* a a)))**. Then inside the body of the **defmacro**, **var**, **lower**, **upper**, and **body** are variables, bound to the matching parts of the macro form.

defmacro *Macro*

defmacro is a general-purpose macro-defining macro. A **defmacro** form looks like:

(defmacro *name pattern* . *body*)

The *pattern* may be anything made up out of symbols and conses. It is matched against the body of the macro form; both *pattern* and the form are car'ed and cdr'ed identically, and whenever a non-**nil** symbol is hit in *pattern*, the symbol is bound to the corresponding part of the form. All of the symbols in *pattern* can be used as variables within *body*. *name* is the name of the macro to be defined; it can be any function spec. See the section "Function Specs". *body* is evaluated with these bindings in effect, and its result is returned to the evaluator as the expansion of the macro.

defmacro could have been defined in terms of **destructuring-bind** as follows:

```
(defmacro defmacro (name pattern &body body)
  `(macro ,name (form)
     (destructuring-bind ,pattern (cdr form)
       . ,body)))
```

See the special form **destructuring-bind**.

Note that the pattern need not be a list the way a lambda-list must. In the above example, the pattern was a "dotted list", since the symbol **body** was supposed to match the cddddr of the macro form. If we wanted a new iteration form, like **for** except that our example would look like:

```
(for a (1 100) (print a) (print (* a a)))
```

(just because we thought that was a nicer syntax), then we could do it merely by modifying the pattern of the **defmacro** above; the new pattern would be **(var (lower upper) . body)**.

Here is how we would write our other examples using **defmacro**:

```
(defmacro first (the-list)
    (list 'car the-list))

(defmacro addone (form)
   (list 'plus '1 form))

(defmacro increment (symbol)
   (list 'setq symbol (list '1+ symbol)))
```

All of these were very simple macros and have very simple patterns, but these examples show that we can replace the **(cadr x)** with a readable mnemonic name such as **the-list** or **symbol**, which makes the program clearer, and enables documentation facilities such as the **arglist** function to describe the syntax of the special form defined by the macro.

There is another version of **defmacro** which defines displacing macros. See the section "Displacing Macros". **defmacro** has other, more complex features. See the section "Advanced Features of **defmacro**".

2.2 Backquote

Now we deal with the other problem: the long strings of calls to **cons** and **list**. This problem is relieved by introducing some new characters that are special to the Lisp reader. Just as the single-quote character makes it easier to type things of the form **(quote** *x***)**, so will some more new special characters make it easier to type forms that create new list structure. The functionality provided by these characters is called the *backquote* facility.

The backquote facility is used by giving a backquote character (‘), followed by a form. If the form does not contain any use of the comma character, the backquote acts just like a single quote: it creates a form that, when evaluated, produces the form following the backquote. For example:

```
'(a b c) => (a b c)
`(a b c) => (a b c)
```

So in the simple cases, backquote is just like the regular single-quote macro. The way to get it to do interesting things is to include a comma somewhere inside the form following the backquote. The comma is followed by a form, and that form gets evaluated even though it is inside the backquote. For example:

```
(setq b 1)
`(a b c)  => (a b c)
`(a ,b c) => (a 1 c)
`(abc ,(+ b 4) ,(- b 1) (def ,b)) => (abc 5 0 (def 1))
```

In other words, backquote quotes everything *except* things preceded by a comma; those things get evaluated.

A list following a backquote can be thought of as a template for some new list structure. The parts of the list that are preceded by commas are forms that fill in slots in the template; everything else is just constant structure that will appear in the result. This is usually what you want in the body of a macro; some of the form generated by the macro is constant, the same thing on every invocation of the macro. Other parts are different every time the macro is called, often being functions of the form that the macro appeared in (the "arguments" of the macro). The latter parts are the ones for which you would use the comma. Several examples of this sort of use follow.

When the reader sees the **`(a ,b c)** it is actually generating a form such as **(list 'a b 'c)**. The actual form generated may use **list**, **cons**, **append**, or whatever might be a good idea; you should never have to concern yourself with what it actually turns into. All you need to care about is what it evaluates to. Actually, it does not use the regular functions **cons**, **list**, and so forth, but uses special ones instead so that the grinder can recognize a form that was created with the backquote syntax, and print it using backquote so that it looks like what you typed in. You should never write any program that depends on this, anyway, because backquote makes no guarantees about how it does what it does. In particular, in some circumstances it may decide to create constant forms that will cause sharing of list structure at run time, or it may decide to create forms that will create new list structure at run time. For example, if the reader sees **'(r . ,nil)**, it may produce the same thing as **(cons 'r nil)**, or **'(r . nil)**. Be careful that your program does not depend on which of these it does.

This is generally found to be pretty confusing by most people; the best way to explain further seems to be with examples. Here is how we would write our three simple macros using both the **defmacro** and backquote facilities.

```
(defmacro first (the-list)
    `(car ,the-list))

(defmacro addone (form)
   `(plus 1 ,form))

(defmacro increment (symbol)
   `(setq ,symbol (1+ ,symbol)))
```

Finally, to demonstrate how easy it is to define macros with these two facilities, here is the final form of the **for** macro.

```
(defmacro for (var lower upper . body)
  `(do ,var ,lower (1+ ,var) (> ,var ,upper) . ,body))
```

Look at how much simpler that is than the original definition. Also, look how closely it resembles the code it is producing. The functionality of the **for** really stands right out when written this way.

If a comma inside a backquote form is followed by an "at-sign" character (@), it has a special meaning. The ",@" should be followed by a form whose value is a *list*; then each of the elements of the list is put into the list being created by the backquote. In other words, instead of generating a call to the **cons** function, backquote generates a call to **append**. For example, if **a** is bound to **(x y z)**, then **`(1 ,a 2)** would evaluate to **(1 (x y z) 2)**, but **`(1 ,@a 2)** would evaluate to **(1 x y z 2)**.

Here is an example of a macro definition that uses the ",@" construction. Suppose you wanted to extend Lisp by adding a kind of special form called **repeat-forever**, which evaluates all of its subforms repeatedly. One way to implement this would be to expand:

```
(repeat-forever form1 form2 form3)
```

into:

```
(prog ()
    a form1
      form2
      form3
      (go a))
```

You could define the macro by:

```
(defmacro repeat-forever body
       `(prog ()
            a ,@body
              (go a)))
```

A similar construct is ",." (comma, dot). This means the same thing as ",@" except that the list which is the value of the following form may be freely smashed; backquote uses **nconc** rather than **append**. This should of course be used with caution.

Backquote does not make any guarantees about what parts of the structure it shares and what parts it copies. You should not do destructive operations such as **nconc** on the results of backquote forms such as:

```
`(,a b c d)
```

since backquote might choose to implement this as:

```
(cons a '(b c d))
```

and **nconc** would smash the constant. On the other hand, it would be safe to **nconc** the result of:

```
`(a b ,c ,d)
```

since there is nothing this could expand into that does not involve making a new list, such as:

```
(list 'a 'b c d)
```

Backquote of course guarantees not to do any destructive operations (**rplaca**, **rplacd**, **nconc**) on the components of the structure it builds, unless the ",." syntax is used.

Advanced macro writers sometimes write "macro-defining macros": forms that expand into forms which, when evaluated, define macros. In such macros it is often useful to use nested backquote constructs. The following example illustrates the use of nested backquotes in the writing of macro-defining macros.

This example is a very simple version of **defstruct**. You should first understand the basic description of **defstruct** before proceeding with this example. The **defstruct** below does not accept any options, and only allows the simplest kind of items; that is, it only allows forms like:

```
(defstruct (name)
    item1
    item2
    item3
    item4
    ...)
```

We would like this form to expand into:

```
(progn 'compile
 (defmacro item1 (x)
      `(aref ,x 0))
 (defmacro item2 (x)
      `(aref ,x 1))
 (defmacro item3 (x)
      `(aref ,x 2))
 (defmacro item4 (x)
      `(aref ,x 3))
 ...)
```

The meaning of the **(progn 'compile ...)** is discussed in another section. See the section "Macros Expanding Into Many Forms". Here is the macro to perform the expansion:

```
(defmacro defstruct ((name) . items)
     (do ((item-list items (cdr item-list))
          (ans nil)
          (i 0 (1+ i)))
         ((null item-list)
          `(progn 'compile . ,(nreverse ans)))
       (setq ans
             (cons `(defmacro ,(car item-list) (x)
                             `(aref ,x ,',i))
                   ans))))
```

The interesting part of this definition is the body of the (inner) **defmacro** form:

```
`(aref ,x ,',i)
```

Instead of using this backquote construction, we could have written:

```
(list 'aref x ,i)
```

That is, the ",'," acts like a comma that matches the outer backquote, while the "," preceding the "**x**" matches with the inner backquote. Thus, the symbol **i** is evaluated when the **defstruct** form is expanded, whereas the symbol **x** is evaluated when the accessor macros are expanded.

Backquote can be useful in situations other than the writing of macros. Whenever there is a piece of list structure to be consed up, most of which is constant, the use of backquote can make the program considerably clearer.

3. Substitutable Functions

A substitutable function is a function that is open-coded by the compiler. It is like any other function when applied, but it can be expanded instead, and in that regard resembles a macro.

defsubst *Special Form*

defsubst is used for defining substitutable functions. It is used just like **defun**:

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

and does almost the same thing. It defines a function that executes identically to the one that a similar call to **defun** would define. The difference comes when a function that *calls* this one is compiled. Then, the call will be open-coded by substituting the substitutable function's definition into the code being compiled. The function itself looks like **(named-subst** *name lambda-list* **.** *body***)**. Such a function is called a **subst**. For example, if we define:

```
(defsubst square (x) (* x x))

(defun foo (a b) (square (+ a b)))
```

then if **foo** is used interpreted, **square** will work just as if it had been defined by **defun**. If **foo** is compiled, however, the squaring will be substituted into it and it will compile just like:

```
(defun foo (a b) (* (+ a b) (+ a b)))
```

square's definition would be:

```
(named-subst square (x) (* x x))
```

See the section "Interpreted Functions". The internal formats of **substs** and **named-substs** are explained in that section.

defsubst now creates compiled functions, where they were previously only interpreted. Thus a subst is now faster when invoked as a function. As a consequence of changes to the implementation of subst, it is now possible to get new compiler warnings when compiling files containing **defsubst**. For example, the following **defsubst** would not have previously gotten a warning, even though **x** is free in **add-with-x**.

```
(defsubst add-with-x (y) (+ x y))
```

The current implementation would issue a warning because substs are now implemented with compiled code objects. (This example would still work if expanded in an environment which lexically contained **x**.)

A similar **square** could be defined as a macro, with:

```
(defmacro square (x) `(* ,x ,x))
```

In general, anything that is implemented as a **subst** can be reimplemented as a macro, just by changing the **defsubst** to a **defmacro** and putting in the appropriate backquote and commas. The disadvantage of macros is that they are not functions, and so cannot be applied to arguments. Their advantage is that they can do much more powerful things than **substs** can. This is also a disadvantage since macros provide more ways to get into trouble. If something can be implemented either as a macro or as a **subst,** it is generally better to make it a **subst.**

You will notice that the substitution performed is very simple and takes no care about the possibility of computing an argument twice when it really ought to be computed once. For instance, the functions:

```
(defsubst reverse-cons (x y) (cons y x))
(defsubst in-order (a b c) (and (< a b) (< b c)))
```

present problems. When compiled, because of the substitution a call to **reverse-cons** would evaluate its arguments in the wrong order, and a call to **in-order** could evaluate its second argument twice. For this reason the writer of **defsubsts** must be cautious. Also all occurrences of the argument names in the body are replaced with the argument forms, wherever they appear. Thus an argument name should not be used in the body for anything else, such as a function name or a symbol in a constant.

As with **defun,** *name* can be any function spec.

4. Symbol Macros

A symbol macro translates a symbol into a substitute form. When the Lisp evaluator is given a symbol, it checks whether the symbol has been defined as a symbol macro. If so, it evaluates the symbol's replacement form instead of the symbol itself. Use **define-symbol-macro** to define a symbol macro.

define-symbol-macro *name form* *Special Form*

This special form defines a symbol macro. *name* is a symbol to be defined as a symbol macro. *form* is a Lisp form to be substituted for the symbol when the symbol is evaluated. A symbol macro is more like a subst than a macro: *form* is the form to be substituted for the symbol, not a form whose evaluation results in the substitute form.

A symbol defined as a symbol macro cannot be used in the context of a variable. You cannot use **setq** on it, and you cannot bind it. You can use **setf** on it: **setf** substitutes the replacement form, which should access something, and expands into the appropriate update function. Example:

```
(define-symbol-macro foo (+ 3 bar))
(setq bar 2)
foo => 5
```

Here is a more complex example. Suppose you want to define some new instance variables and methods for a flavor. You want to test the methods using existing instances of the flavor. For testing purposes, you might use hash tables to simulate the instance variables, using one hash table per instance variable with the instance as the key. You could then implement an instance variable **x** as a symbol macro:

```
(defvar x-hash-table (make-hash-table))
(define-symbol-macro x (send x-hash-table ':get-hash self))
```

To simulate setting a new value for **x**, you could use **(setf x** *value***)**, which would expand into **(send x-hash-table ':put-hash self** *value***)**.

5. Hints to Macro Writers

There are many useful techniques for writing macros. Over the years, Lisp programmers have discovered techniques that most programmers find useful, and have identified pitfalls that must be avoided. This section discusses some of these techniques, and illustrates them with examples.

The most important thing to keep in mind as you learn to write macros is that you should first figure out what the macro form is supposed to expand into, and only then should you start to actually write the code of the macro. If you have a firm grasp of what the generated Lisp program is supposed to look like, from the start, you will find the macro much easier to write.

In general any macro that can be written as a substitutable function should be written as one, not as a macro, for several reasons:

- Substitutable functions are easier to write and to read.
- They can be passed as functional arguments (for example, you can pass them to **mapcar**).
- There are some subtleties that can occur in macro definitions that need not be worried about in substitutable functions.

See the section "Substitutable Functions". A macro can be a substitutable function only if it has exactly the semantics of a function, rather than a special form. The macros we will see in this section are not semantically like functions; they must be written as macros.

5.1 Name Conflicts

One of the most common errors in writing macros is best illustrated by example. Suppose we wanted to write **dolist** as a macro that expanded into a **do**. The first step, as always, is to figure out what the expansion should look like. Let's pick a representative example form, and figure out what its expansion should be. Here is a typical **dolist** form.

```
(dolist (element (append a b))
  (push element *big-list*)
  (foo element 3))
```

We want to create a **do** form that does the thing that the above **dolist** form says to do. That is the basic goal of the macro: it must expand into code that does the same thing that the original code says to do, but it should be in terms of existing Lisp constructs. The **do** form might look like this:

```
(do ((list (append a b) (cdr list))
     (element))
    ((null list))
  (setq element (car list))
  (push element *big-list*)
  (foo element 3))
```

Now we could start writing the macro that would generate this code, and in general convert any **dolist** into a **do**, in an analogous way. However, there is a problem with the above scheme for expanding the **dolist**. The above expansion works fine. But what if the input form had been the following:

```
(dolist (list (append a b))
  (push list *big-list*)
  (foo list 3))
```

This is just like the form we saw above, except that the user happened to decide to name the looping variable **list** rather than **element**. The corresponding expansion would be:

```
(do ((list (append a b) (cdr list))
     (list))
    ((null list))
  (setq list (car list))
  (push list *big-list*)
  (foo list 3))
```

This does not work at all! In fact, this is not even a valid program, since it contains a **do** that uses the same variable in two different iteration clauses.

Here is another example that causes trouble:

```
(let ((list nil))
  (dolist (element (append a b))
    (push element list)
    (foo list 3)))
```

If you work out the expansion of this form, you will see that there are two variables named **list**, and that the user meant to refer to the outer one but the generated code for the **push** actually uses the inner one.

The problem here is an accidental name conflict. This can happen in any macro that has to create a new variable. If that variable ever appears in a context in which user code might access it, then you have to worry that it might conflict with some other name that the user is using for his own program.

One way to avoid this problem is to choose a name that is very unlikely to be picked by the user, simply by choosing an unusual name. This will probably work, but it is inelegant since there is no guarantee that the user will not just happen to choose the same name. The only sure way to avoid the name conflict is to use an uninterned symbol as the variable in the generated code. The function **gensym** is useful for creating such symbols.

Here is the expansion of the original form, using an uninterned symbol created by **gensym**.

```
(do ((g0005 (append a b) (cdr g0005))
     (element))
    ((null g0005))
  (setq element (car g0005))
  (push element *big-list*)
  (foo element 3))
```

This is the right kind of thing to expand into. Now that we understand how the expansion works, we are ready to actually write the macro. Here it is:

```
(defmacro dolist ((var form) . body)
  (let ((dummy (gensym)))
    '(do ((,dummy ,form (cdr ,dummy))
          (,var))
         ((null ,dummy))
       (setq ,var (car ,dummy))
       . ,body)))
```

Many system macros do not use **gensym** for the internal variables in their expansions. Instead they use symbols whose print names begin and end with a dot. This provides meaningful names for these variables when looking at the generated code and when looking at the state of a computation in the Debugger. However, this convention means that users should avoid naming variables this way.

5.2 Prog-context Conflicts

A related problem occurs when you write a macro that expands into a **prog** (or a **do**, or something that expands into **prog** or **do**) behind the user's back (unlike **dolist**, which is documented to be like **do**).

Consider the **error-restart** special form; suppose we wanted to implement it as a macro that expands into a **prog**. If it expanded into a standard **prog**, then the following (contrived) Lisp program would not behave correctly:

```
(prog ()
   (setq a 3)
   (error-restart
     (cond ((> a 10)
            (return 5))
           ((> a 4)
            (cerror nil t 'lose "You lose."))))
   (setq b 7))
```

The problem is that the **return** would return from the **error-restart** instead of the **prog**. The way to avoid this problem is to use a named **prog** whose name is **t**. The name **t** is special in that it is invisible to the **return** function. If we write

error-restart as a macro that expands into a **prog** named **t**, then the **return** will pass right through the **error-restart** form and return from the **prog**, as it ought to.

In general, when a macro expands into a **prog** or a **do** around the user's code, the **prog** or **do** should be named **t** so that **return** forms in the user code will return to the right place, unless the macro is documented as generating a **prog/do**-like form that may be exited with **return**.

5.3 Macros Expanding Into Many Forms

Sometimes a macro wants to do several different things when its expansion is evaluated. Another way to say this is that sometimes a macro wants to expand into several things, all of which should happen sequentially at run time (not macro-expand time). For example, suppose you wanted to implement **defconst** as a macro. **defconst** must do two things: declare the variable to be special, and set the variable to its initial value. (We will implement a simplified **defconst** that only does these two things, and does not have any options.) What should a **defconst** form expand into? Well, what we would like is for an appearance of:

```
(defconst a (+ 4 b))
```

in a file to be the same thing as the appearance of the following two forms:

```
(declare (special a))
(setq a (+ 4 b))
```

However, because of the way that macros work, they expand into only one form, not two. So we need to have a **defconst** form expand into one form that is just like having two forms in the file.

There is such a form. It looks like this:

```
(progn 'compile
       (declare (special a))
       (setq a (+ 4 b)))
```

In interpreted Lisp, it is easy to see what happens here. This is a **progn** special form, and so all its subforms are evaluated, in turn. First the form **'compile** is evaluated. The result is the symbol **compile**; this value is not used, and evaluation of **'compile** has no side effects, so the **'compile** subform is effectively ignored. Then the **declare** form and the **setq** form are evaluated, and so each of them happens, in turn. So far, so good.

The interesting thing is the way this form is treated by the compiler. The compiler specially recognizes any **progn** form at top level in a file whose first subform is **'compile**. When it sees such a form, it processes each of the remaining subforms of the **progn** just as if that form had appeared at top level in the file. So the compiler behaves exactly as if it had encountered the **declare** form at top level, and then encountered the **setq** form at top level, even though neither of those forms was actually at top level (they were both inside the **progn**). This feature of the compiler is provided specifically for the benefit of macros that want to expand into several things.

Here is the macro definition:

```
(defmacro defconst (variable init-form)
  `(progn 'compile
          (declare (special ,variable))
          (setq ,variable ,init-form)))
```

Here is another example of a form that wants to expand into several things. We will implement a special form called **define-command**, which is intended to be used in order to define commands in some interactive user subsystem. For each command, there are two things provided by the **define-command** form: a function that executes the command, and a text string that contains the documentation for the command (in order to provide an online interactive documentation feature). This macro is a simplified version of a macro that is actually used in the Zwei editor. Suppose that in this subsystem, commands are always functions of no arguments, documentation strings are placed on the **help** property of the name of the command, and the names of all commands are put onto a list. A typical call to **define-command** would look like:

```
(define-command move-to-top
   "This command moves you to the top."
   (do ()
       ((at-the-top-p))
     (move-up-one)))
```

This could expand into:

```
(progn 'compile
       (defprop
         move-to-top
         "This command moves you to the top."
         help)
       (push 'move-to-top *command-name-list*)
       (defun move-to-top ()
         (do ()
             ((at-the-top-p))
           (move-up-one)))
       )
```

The **define-command** expands into three forms. The first one sets up the documentation string and the second one puts the command name onto the list of all command names. The third one is the **defun** that actually defines the function itself. Note that the **defprop** and **push** happen at load time (when the file is loaded); the function, of course, also gets defined at load time. (See the description of **eval-when** for more discussion of the differences among compile time, load time, and eval time.)

This technique makes Lisp a powerful language in which to implement your own language. When you write a large system in Lisp, frequently you can make things much more convenient and clear by using macros to extend Lisp into a customized language for your application. In the above example, we have created a little language extension: a new special form that defines commands for our system. It lets the writer of the system put documentation strings right next to the code that they document, so that the two can be updated and maintained together. The way that the Lisp environment works, with load-time evaluation able to build data structures, lets the documentation data base and the list of commands be constructed automatically.

5.4 Macros That Surround Code

There is a particular kind of macro that is very useful for many applications. This is a macro that you place "around" some Lisp code, in order to make the evaluation of that code happen in some context. For a very simple example, we could define a macro called **with-output-in-base**, that executes the forms within its body with any output of numbers that is done defaulting to a specified base.

```
(defmacro with-output-in-base ((base-form) &body body)
   '(let ((base ,base-form))
      . ,body))
```

A typical use of this macro might look like:

```
(with-output-in-base (*default-base*)
   (print x)
   (print y))
```

that would expand into:

```
(let ((base *default-base*))
  (print x)
  (print y))
```

This example is too trivial to be very useful; it is intended to demonstrate some stylistic issues. There are some special forms in Zetalisp that are similar to this macro. See the special form **with-open-file**. See the special form **with-input-from-string**. The really interesting thing, of course, is that you can define your own such special forms for your own specialized applications. One very powerful application of this technique was used in a system that manipulates and solves the Rubik's cube puzzle. The system heavily uses a special form called **with-front-and-top**, whose meaning is "evaluate this code in a context in which this specified face of the cube is considered the front face, and this other specified face is considered the top face".

The first thing to keep in mind when you write this sort of macro is that you can make your macro much clearer to people who might read your program if you conform to a set of loose standards of syntactic style. By convention, the names of such special forms start with "**with-**". This seems to be a clear way of expressing the concept that we are setting up a context; the meaning of the special form is "do this stuff *with* the following things true". Another convention is that any "parameters" to the special form should appear in a list that is the first subform of the special form, and that the rest of the subforms should make up a body of forms that are evaluated sequentially with the last one returned. All of the examples cited above work this way. In our **with-output-in-base** example, there was one parameter (the base), which appears as the first (and only) element of a list that is the first subform of the special form. The extra level of parentheses in the printed representation serves to separate the "parameter" forms from the "body" forms so that it is textually apparent which is which; it also provides a convenient way to provide default parameters (a good example is the **with-input-from-string** special form, which takes two required and two optional "parameters"). Another convention/technique is to use the **&body** keyword in the **defmacro** to tell the editor how to correctly indent the special form. See the section "Advanced Features of **defmacro**".

The other thing to keep in mind is that control can leave the special form either by the last form's returning, or by a nonlocal exit (that is, something doing a ***throw**). You should write the special form in such a way that everything will be cleaned up appropriately no matter which way control exits. In our **with-output-in-base** example, there is no problem, because nonlocal exits undo lambda-bindings. However, in even slightly more complicated cases, an **unwind-protect** form is needed: the macro must expand into an **unwind-protect** that surrounds the body, with "cleanup" forms that undo the context-setting-up that the macro did. For example, **using-resource** is implemented as a macro that does an **allocate-resource** and then performs the body inside of an **unwind-protect** that has a **deallocate-resource** in its "cleanup" forms. This way the allocated resource item will be deallocated whenever control leaves the **using-resource** special form.

5.5 Multiple and Out-of-order Evaluation

In any macro, you should always pay attention to the problem of multiple or out-of-order evaluation of user subforms. Here is an example of a macro with such a problem. This macro defines a special form with two subforms. The first is a reference, and the second is a form. The special form is defined to create a cons whose car and cdr are both the value of the second subform, and then to set the reference to be that cons. Here is a possible definition:

```
(defmacro test (reference form)
   `(setf ,reference (cons ,form ,form)))
```

Simple cases will work all right:

```
(test foo 3) ==>
  (setf foo (cons 3 3))
```

But a more complex example, in which the subform has side effects, can produce surprising results:

```
(test foo (setq x (1+ x))) ==>
  (setf foo (cons (setq x (1+ x))
                  (setq x (1+ x))))
```

The resulting code evaluates the **setq** form twice, and so **x** is increased by two instead of by one. A better definition of **test** that avoids this problem is:

```
(defmacro test (reference form)
   (let ((value (gensym)))
     `(let ((,value ,form))
        (setf ,reference (cons ,value ,value)))))
```

With this definition, the expansion works as follows:

```
(test foo (setq x (1+ x))) ==>
  (let ((g0005 (setq x (1+ x))))
    (setf foo (cons g0005 g0005)))
```

In general, when you define a new special form that has some forms as its subforms, you have to be careful about just when those forms get evaluated. If you are not careful, they can get evaluated more than once, or in an unexpected order, and this can be semantically significant if the forms have side effects. There is nothing fundamentally wrong with multiple or out-of-order evaluation if that is really what you want and if it is what you document your special form to do. However, it is very common for special forms to simply behave like functions, and when they are doing things like what functions do, it is natural to expect them to be function-like in the evaluation of their subforms. Function forms have their subforms evaluated, each only once, in left-to-right order, and special forms that are similar to function forms should try to work that way too for clarity and consistency.

There is a tool that makes it easier for you to follow the principle explained above. It is a macro called **once-only**. It is most easily explained by example. The way you would write **test** using **once-only** is as follows:

```
(defmacro test (reference form)
  (once-only (form)
    `(setf ,reference (cons ,form ,form))))
```

This defines **test** in such a way that the **form** is only evaluated once, and references to **form** inside the macro body refer to that value. **once-only** automatically introduces a lambda-binding of a generated symbol to hold the value of the form. Actually, it is more clever than that; it avoids introducing the lambda-binding for forms whose evaluation is trivial and may be repeated without harm or cost, such as numbers, symbols, and quoted structure. This is just an optimization that helps produce more efficient code.

The **once-only** macro makes it easier to follow the principle, but it does not completely nor automatically solve the problems of multiple and out-of-order evaluation. It is just a tool that can solve some of the problems some of the time; it is not a panacea.

The following description attempts to explain what **once-only** does, but it is a lot easier to use **once-only** by imitating the example above than by trying to understand **once-only**'s rather tricky definition.

once-only *Macro*

A **once-only** form looks like:

```
(once-only var-list
  form1
  form2
  ...)
```

var-list is a list of variables. The *forms* are a Lisp program that presumably uses the values of those variables. When the form resulting from the expansion of the **once-only** is evaluated, the first thing it does is to inspect the values of each of the variables in *var-list*; these values are assumed to be Lisp forms. For each of the variables, it binds that variable either to its current value, if the current value is a trivial form, or to a generated symbol. Next, **once-only** evaluates the *forms*, in this new binding environment, and when they have been evaluated it undoes the bindings. The result of the evaluation of the last *form* is presumed to be a Lisp form, typically the expansion of a macro. If all of the variables had been bound to trivial forms, then *once-only* just returns that result. Otherwise, **once-only** returns the result wrapped in a lambda-combination that binds the generated symbols to the result of evaluating the respective nontrivial forms.

The effect is that the program produced by evaluating the **once-only** form is coded in such a way that it only evaluates each form once, unless evaluation of the form has no side effects, for each of the forms that were the values of variables in *var-list*. At the same time, no unnecessary lambda-binding appears in this program, but the body of the **once-only** is not cluttered up with extraneous code to decide whether or not to introduce lambda-binding in the program it constructs.

Note well: while **once-only** attempts to prevent multiple evaluation, it does *not* necessarily preserve the *order* of evaluation of the forms! Since it generates the new bindings, the evaluation of complex forms (for which a new variable needs to be created) may be moved ahead of the evaluation of simple forms (such as variable references). **once-only** does not solve all of the problems mentioned in this section.

Caution! A number of system macros, **setf** for example, fail to follow this convention. Unexpected multiple evaluation and out-of-order evaluation can occur with them. This was done for the sake of efficiency, is prominently mentioned in the documentation of these macros, and will be fixed in the future. It would be best not to compromise the semantic simplicity of your own macros in this way.

5.6 Nesting Macros

A useful technique for building language extensions is to define programming constructs that employ two special forms, one of which is used inside the body of the other. Here is a simple example. There are two special forms; the outer one is called **with-collection**, and the inner one is called **collect**. **collect** takes one subform, which it evaluates; **with-collection** just has a body, whose forms it evaluates sequentially. **with-collection** returns a list of all of the values that were given to **collect** during the evaluation of the **with-collection**'s body. For example:

```
(with-collection
  (dotimes (i 5)
    (collect i)))

  => (1 2 3 4 5)
```

Remembering the first piece of advice we gave about macros, the next thing to do is to figure out what the expansion looks like. Here is how the above example could expand:

```
(let ((g0005 nil))
  (dotimes (i 5)
    (push i g0005))
  (nreverse g0005))
```

Now, how do we write the definition of the macros? Well, **with-collection** is pretty easy:

```
(defmacro with-collection (&body body)
  (let ((var (gensym)))
    `(let ((,var nil))
       ,@body
       (nreverse ,var))))
```

The hard part is writing **collect**. Let's try it:

```
(defmacro collect (argument)
  `(push ,argument ,var))
```

Note that something unusual is going on here: **collect** is using the variable **var** freely. It is depending on the binding that takes place in the body of **with-collection** in order to get access to the value of **var**. Unfortunately, that binding took place when **with-collection** got expanded; **with-collection**'s expander function bound **var**, and it got unbound when the expander function was done. By the time the **collect** form gets expanded, **var** has long since been unbound. The macro definitions above do not work. Somehow the expander function of **with-collection** has to communicate with the expander function of **collect** to pass over the generated symbol.

The only way for **with-collection** to convey information to the expander function of **collect** is for it to expand into something that passes that information. What we can do is to define a special variable (which we will call ***collect-variable***), and have **with-collection** expand into a form that binds this variable to the name of the variable that the **collect** should use. Now, consider how this works in the interpreter. The evaluator will first see the **with-collection** form, and call in the expander function to expand it. The expander function creates the expansion, and returns to the evaluator, which then evaluates the expansion. The expansion includes in it a **let** form to bind ***collect-variable*** to the generated symbol. When the evaluator sees this **let** form during the evaluation of the expansion of the **with-collection** form, it will set up the binding and recursively evaluate the body of the **let**. Now, during the evaluation of the body of the **let**, our special variable is bound, and if the expander function of **collect** gets run, it will be able to see the value of **collection-variable** and incorporate the generated symbol into its own expansion.

Writing the macros this way is not quite right. It works fine interpreted, but the problem is that it does not work when we try to compile Lisp code that uses these special forms. When code is being compiled, there is no interpreter to do the binding in our new **let** form; macro expansion is done at compile time, but generated code does not get run until the results of the compilation are loaded and run. The way to fix our definitions is to use **compiler-let** instead of **let**. **compiler-let** is a special form that exists specifically to do the sort of thing we are trying to do here. **compiler-let** is identical to **let** as far as the interpreter is concerned, so changing our **let** to a **compiler-let** will not affect the behavior in the interpreter; it will continue to work. When the compiler encounters a **compiler-let**, however, it actually performs the bindings that the **compiler-let** specifies, and proceeds to compile the body of the **compiler-let** with all of those bindings in effect. In other words, it acts as the interpreter would.

Here is the right way to write these macros:

```
(defvar *collect-variable*)

(defmacro with-collection (&body body)
  (let ((var (gensym)))
    `(let ((,var nil))
       (compiler-let ((*collect-variable* ',var))
         . ,body)
       (nreverse ,var))))

(defmacro collect (argument)
  `(push ,argument ,*collect-variable*))
```

5.7 Functions Used During Expansion

The technique of defining functions to be used during macro expansion deserves explicit mention here. It may not occur to you, but a macro expander function is a Lisp program like any other Lisp program, and it can benefit in all the usual ways by being broken down into a collection of functions that do various parts of its work. Usually macro expander functions are pretty simple Lisp programs that take things apart and put them together slightly differently and such, but some macros are quite complex and do a lot of work. Several features of Zetalisp, including flavors, **loop**, and **defstruct**, are implemented using very complex macros, which, like any complex well-written Lisp program, are broken down into modular functions. You should keep this in mind if you ever invent an advanced language extension or ever find yourself writing a five-page expander function.

A particular thing to note is that any functions used by macro-expander functions must be available at compile time. You can make a function available at compile time by surrounding its defining form with an **(eval-when (compile load eval) ...)**. See the special form **eval-when**. Doing this means that at compile time the definition of the function will be interpreted, not compiled, and hence will run more slowly. Another approach is to separate macro definitions and the functions they call during expansion into a separate file, often called a "defs" (definitions) file. This file defines all the macros but does not use any of them. It can be separately compiled and loaded up before compiling the main part of the program, which uses the macros. The *system* facility helps keep these various files straight, compiling and loading things in the right order. See the document *Maintaining Large Systems*.

6. Aid for Debugging Macros

mexp *Function*

mexp goes into a loop in which it reads forms and sequentially expands them, printing out the result of each expansion (using the grinder to improve readability). See the section "Formatting Lisp Code". It terminates when it reads an atom (anything that is not a cons). If you type in a form that is not a macro form, there will be no expansions and so it will not type anything out, but just prompt you for another form. This allows you to see what your macros are expanding into, without actually evaluating the result of the expansion.

7. Displacing Macros

Every time the the evaluator sees a macro form, it must call the macro to expand the form. If this expansion always happens the same way, then it is wasteful to expand the whole form every time it is reached; why not just expand it once? A macro is passed the macro form itself, and so it can change the car and cdr of the form to something else by using **rplaca** and **rplacd**! This way the first time the macro is expanded, the expansion will be put where the macro form used to be, and the next time that form is seen, it will already be expanded. A macro that does this is called a *displacing macro*, since it displaces the macro form with its expansion.

The major problem with this is that the Lisp form gets changed by its evaluation. If you were to write a program that used such a macro, call **grindef** to look at it, then run the program and call **grindef** again, you would see the expanded macro the second time. Presumably the reason the macro is there at all is that it makes the program look nicer; we would like to prevent the unnecessary expansions, but still let **grindef** display the program in its more attractive form. This is done with the function **displace**.

Another thing to worry about with displacing macros is that if you change the definition of a displacing macro, then your new definition will not take effect in any form that has already been displaced. If you redefine a displacing macro, an existing form using the macro will use the new definition only if the form has never been evaluated.

displace *form expansion* *Function*

form must be a list. **displace** replaces the car and cdr of *form* so that it looks like:

(si:displaced *original-form expansion*)

original-form is equal to *form* but has a different top-level cons so that the replacing mentioned above does not affect it. **si:displaced** is a macro, which returns the caddr of its own macro form. So when the **si:displaced** form is given to the evaluator, it "expands" to *expansion*. **displace** returns *expansion*.

The grinder knows specially about **si:displaced** forms, and will grind such a form as if it had seen the original form instead of the **si:displaced** form.

So if we wanted to rewrite our **addone** macro as a displacing macro, instead of writing:

```
(macro addone (x)
   (list 'plus '1 (cadr x)))
```

we would write:

```
(macro addone (x)
  (displace x (list 'plus '1 (cadr x))))
```

Of course, we really want to use **defmacro** to define most macros. Since there is no way to get at the original macro form itself from inside the body of a **defmacro**, another version of it is provided:

defmacro-displace *Macro*

defmacro-displace is just like **defmacro** except that it defines a displacing macro, using the **displace** function.

Now we can write the displacing version of **addone** as:

```
(defmacro-displace addone (val)
  (list 'plus '1 val))
```

All we have changed in this example is the **defmacro** into **defmacro-displace**. **addone** is now a displacing macro.

8. Advanced Features of defmacro

The pattern in a **defmacro** is more like the **lambda**-list of a normal function than revealed above. It is allowed to contain certain **&**-keywords.

&optional is followed by *variable*, (*variable*), (*variable default*), or (*variable default present-p*), exactly the same as in a function. Note that *default* is still a form to be evaluated, even though *variable* is not being bound to the value of a form. *variable* does not have to be a symbol; it can be a pattern. In this case the first form is disallowed because it is syntactically ambiguous. The pattern must be enclosed in a singleton list. If *variable* is a pattern, *default* can be evaluated more than once.

Using **&rest** is the same as using a dotted list as the pattern, except that it may be easier to read and leaves a place to put **&aux**.

&aux is the same in a macro as in a function, and has nothing to do with pattern matching.

defmacro has a couple of additional keywords not allowed in functions.

&body is identical to **&rest** except that it informs the editor and the grinder that the remaining subforms constitute a "body" rather than "arguments" and should be indented accordingly.

&list-of *pattern* requires the corresponding position of the form being translated to contain a list (or **nil**). It matches *pattern* against each element of that list. Each variable in *pattern* is bound to a list of the corresponding values in each element of the list matched by the **&list-of**. This may be clarified by an example. Suppose we want to be able to say things like:

```
(send-commands (aref turtle-table i)
  (forward 100)
  (beep)
  (left 90)
  (pen 'down 'red)
  (forward 50)
  (pen 'up))
```

We could define a **send-commands** macro as follows:

```
(defmacro send-commands (object
                &body &list-of (command . arguments))
  `(let ((o ,object))
     . ,(mapcar #'(lambda (com args) `(send o ',com . ,args))
                command arguments)))
```

Note that this example uses **&body** together with **&list-of**, so you do not see the list itself; the list is just the rest of the macro-form.

You can combine **&optional** and **&list-of**. Consider the following example:

```
(defmacro print-let (x &optional &list-of ((vars vals)
                                           '((base 10.)
                                             (*nopoint t))))
  `((lambda (,@vars) (print ,x))
    ,@vals))

(print-let foo)  ==>
((lambda (base *nopoint)
   (print foo))
 12
 t)

(print-let foo ((bar 3)))  ==>
((lambda (bar)
   (print foo))
 3)
```

In this example we are not using **&body** or anything like it, so you do see the list itself; that is why you see parentheses around the **(bar 3)**.

9. Functions to Expand Macros

The following two functions are provided to allow the user to control expansion of macros; they are often useful for the writer of advanced macro systems, and in tools that want to examine and understand code which may contain macros.

macroexpand-1 *form* *Function*

If *form* is a macro form, this expands it (once) and returns the expanded form. Otherwise it just returns *form*. **macroexpand-1** expands **defsubst** function forms as well as macro forms.

macroexpand *form* *Function*

If *form* is a macro form, this expands it repeatedly until it is not a macro form, and returns the final expansion. Otherwise, it just returns *form*. **macroexpand** expands **defsubst** function forms as well as macro forms.

10. Generalized Variables

In Lisp, a variable is something that can remember one piece of data. The main 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 can be generalized to any storage location that can remember one piece of data, no matter how that location is named. See the section "Variables: Evaluation".

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

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 that accesses it) into a form that updates it.

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 you should not have to know what it is in order to alter an element of the structure.

setf is actually a macro that expands into the appropriate update function. It has a database, explained below, that associates from access functions to update functions.

setf *access-form value* *Macro*

setf takes a form that *accesses* something, and "inverts" it 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 *access-form*. 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) ==> (aset 56 q 2)
(setf (cadr w) x) ==> (rplaca (cdr w) x)
```

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

For the sake of efficiency, the code produced by **setf** does not preserve order of evaluation of the argument forms. This is only a problem if the argument forms have interacting side-effects. For example, if you evaluate:

```
(setq x 3)
(setf (aref a x) (setq x 4))
```

then the form might set element **3** or element **4** of the array. We do not guarantee which one it will do; do not just try it and see and then depend on it, because it is subject to change without notice.

Furthermore, the value produced by **setf** depends on the structure type and is not guaranteed; **setf** should be used for side effect only.

Besides the *access* and *update* conceptual operations on variables, there is a third basic operation, which we might call *locate*. Given the name of a storage cell, the *locate* operation will return the address of that cell as a locative pointer. See the section "Locatives". This locative pointer is a kind of name for the variable that is a first-class Lisp data object. It can be passed as an argument to a function that operates on any kind of variable, regardless of how it is named. It can be used to *bind* the variable, using the **bind** subprimitive.

Of course this can only work on variables whose implementation is really to store their value in a memory cell. A 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 as such is not actually stored anywhere.

locf *access-form* *Macro*

locf takes a form which *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 *access-form* invokes a macro or a substitutable function, then **locf** expands the *access-form* and starts over again. This lets you use **locf** together with **defstruct** accessor macros.

If *access-form* is **(cdr** *list***)**, **locf** returns the list itself instead of a locative.

Both **setf** and **locf** work by means of property lists. When the form **(setf (aref q 2) 56)** is expanded, **setf** looks for the **setf** property of the symbol **aref**. The value of the **setf** property of a symbol should be a cons whose car is a pattern to be matched with the *access-form*, and whose **cdr** is the corresponding *update-form*, with the symbol **si:val** in place of the value to be stored. The **setf** property of **aref** is a cons whose car is **(aref array . subscripts)** and whose cdr is **(aset si:val array . subscripts)**. If the transformation that **setf** is to do cannot be expressed as a simple pattern, an arbitrary function may be used: When the form **(setf (foo bar) baz)** is being expanded, if the **setf** property of **foo** is a symbol, the function definition of that symbol will be applied to two arguments, **(foo bar)** and **baz**, and the result will be taken to be the expansion of the **setf**.

Similarly, the **locf** function uses the **locf** property, whose value is analogous. For example, the **locf** property of **aref** is a cons whose car is **(aref array . subscripts)** and whose cdr is **(aloc array . subscripts)**. There is no **si:val** in the case of **locf**.

incf *access-form [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*.

incf expands into a **setf** form, so *ref* can be anything that **setf** understands as its *access-form*. This also means that you should not depend on the returned value of an **incf** form.

You must take great care with **incf** because it may evaluate parts of *ref* more than once. For example:

```
(incf (car (mumble))) ==>
(setf (car (mumble)) (1+ (car (mumble)))) ==>
(rplaca (mumble) (1+ (car (mumble))))
```

The **mumble** function is called more than once, which may be significantly inefficient if **mumble** is expensive, and which may be downright wrong if **mumble** has side effects. The same problem can come up with the **decf**, **swapf**, **push**, and **pop** macros.

decf *access-form [amount]* *Macro*

Decrements the value of a generalized variable. **(decf** *ref***)** decrements the value of *ref* by 1. **(decf** *ref amount***)** subtracts *amount* from *ref* and stores the difference back into *ref*.

decf expands into a **setf** form, so *ref* can be anything that **setf** understands as its *access-form*. This also means that you should not depend on the returned value of a **decf** form.

swapf *a b* *Macro*

Exchanges the value of one generalized variable with that of another. *a* and *b* are access-forms suitable for **setf**. The returned value is not defined. All the caveats that apply to **incf** apply to **swapf** as well: Forms within *a* and *b* may be evaluated more than once. Examples:

```
(swapf a b)
   ==> (setf a (prog1 b (setf b a)))
   ==> (setq a (prog1 b (setq b a)))

(swapf (car (foo)) (car (bar)))
   ==> (setf (car (foo)) (prog1 (car (bar)) (setf (car (bar)) (car (foo)))))
   ==> (rplaca (foo) (prog1 (car (bar)) (rplaca (bar) (car (foo)))))
```

Note that in the second example the functions **foo** and **bar** are called twice.

push *item access-form* *Macro*

Adds an item to the front of a list which is stored in a generalized variable. **(push** *item ref***)** creates a new cons whose car is the result of evaluating *item* and whose cdr is the contents of *ref*, and stores the new cons into *ref*.

The form:

```
(push (hairy-function x y z) variable)
```

replaces the commonly used construct:

```
(setq variable (cons (hairy-function x y z) variable))
```

and is intended to be more explicit and esthetic.

All the caveats that apply to **incf** apply to **push** as well: forms within *ref* may be evaluated more than once. The returned value of **push** is not defined.

pop *access-form* *Macro*

Removes an element from the front of a list which is stored in a generalized variable. **(pop** *ref***)** finds the cons in *ref*, stores the cdr of the cons back into *ref*, and returns the car of the cons. Example:

```
(setq x '(a b c))
(pop x) => a
x => (b c)
```

All the caveats that apply to **incf** apply to **pop** as well: forms within *ref* may be evaluated more than once.

Index

symbolics™

DEFS Defstruct

Cambridge, Massachusetts

Defstruct

# 990057

February 1984

This document corresponds to Release 5.0.

This document was prepared by the Documentation Group of Symbolics, Inc.

Printing year and number: 87 86 85 84 9 8 7 6 5 4 3 2 1

Table of Contents

1. Introduction to Structure Macros

defstruct provides a facility in Lisp for creating and using aggregate data types with named elements. These are like "structures" in PL/I, or "records" in PASCAL. In this chapter we see how macros can be used to extend Lisp's data structures. (To see how to use macros to extend the control structures of Lisp: See the document *Macros*. See the section "Loop".

To explain the basic idea, assume you were writing a Lisp program that dealt with space ships. In your program, you want to represent a space ship by a Lisp object of some kind. The interesting things about a space ship, as far as your program is concerned, are its position (X and Y), velocity (X and Y), and mass. How do you represent a space ship?

Well, the representation could be a list of the x-position, y-position, and so on. Equally well it could be an array of five elements, the zeroth being the x-position, the first being the y-position, and so on. The problem with both of these representations is that the "elements" (such as x-position) occupy places in the object that are quite arbitrary, and hard to remember (Hmm, was the mass the third or the fourth element of the array?). This would make programs harder to write and read. It would not be obvious when reading a program that an expression such as **(cadddr ship1)** or **(aref ship2 3)** means "the y component of the ship's velocity", and it would be very easy to write **caddr** in place of **cadddr**.

What we would like to see are names, easy to remember and to understand. If the symbol **foo** were bound to a representation of a space ship, then:

```
(ship-x-position foo)
```

could return its x-position, and:

```
(ship-y-position foo)
```

its y-position, and so forth. The **defstruct** facility does just this.

defstruct itself is a macro that defines a structure. For the space ship example above, we might define the structure by saying:

```
(defstruct (ship)
     ship-x-position
     ship-y-position
     ship-x-velocity
     ship-y-velocity
     ship-mass)
```

This says that every **ship** is an object with five named components. (This is a very simple case of **defstruct**; we will see the general form later.) The evaluation of this form does several things. First, it defines **ship-x-position** to be a function that, given a ship, returns the x component of its position. This is called an *accessor* *function*, because it *accesses* a component of a structure. **defstruct** defines the other four accessor functions analogously.

defstruct will also define **make-ship** to be a macro that expands into the necessary Lisp code to create a **ship** object. So **(setq x (make-ship))** will make a new ship, and set **x** to it. This macro is called the *constructor macro*, because it constructs a new structure.

We also want to be able to change the contents of a structure. To do this, we use the **setf** macro as follows (for example):

```
(setf (ship-x-position x) 100)
```

Here **x** is bound to a ship, and after the evaluation of the **setf** form, the **ship-x-position** of that ship will be 100. Another way to change the contents of a structure is to use the alterant macro. See the section "Alterant Macros".

How does all this map into the familiar primitives of Lisp? In this simple example, we left the choice of implementation technique up to **defstruct**; it will choose to represent a ship as an array. The array has five elements, which are the five components of the ship. The accessor functions are defined thus:

```
(defun ship-x-function (ship)
  (aref ship 0))
```

The constructor macro **(make-ship)** expands into **(make-array 5)**, which makes an array of the appropriate size to be a ship. Note that a program that uses ships need not contain any explicit knowledge that ships are represented as five-element arrays; this is kept hidden by **defstruct**.

The accessor functions are not actually ordinary functions; instead they are **substs**. See the section "Interpreted Functions". This difference has two implications: it allows **setf** to understand the accessor functions, and it allows the compiler to substitute the body of an accessor function directly into any function that uses it, making compiled programs that use **defstruct** exactly equal in efficiency to programs that "do it by hand." Thus writing **(ship-mass s)** is exactly equivalent to writing **(aref s 4)**, and writing **(setf (ship-mass s) m)** is exactly equivalent to writing **(aset m s 4)**, when the program is compiled. It is also possible to tell **defstruct** to implement the accessor functions as macros; this is not normally done in Zetalisp, however.

We can now use the **describe-defstruct** function to look at the **ship** object, and see what its contents are:

```
(describe-defstruct x 'ship) =>

#<art-q-5 17073131> is a ship
   ship-x-position:            100
   ship-y-position:            nil
   ship-x-velocity:            nil
   ship-y-velocity:            nil
   ship-mass:                  nil
#<art-q-5 17073131>
```

See the function **describe-defstruct**.

By itself, this simple example provides a powerful structure definition tool. But, in fact, **defstruct** has many other features. First of all, we might want to specify what kind of Lisp object to use for the "implementation" of the structure. The example above implemented a "ship" as an array, but **defstruct** can also implement structures as array-leaders, lists, and other things. (For array-leaders, the accessor functions call **array-leader**, for lists, **nth**, and so on.)

Most structures are implemented as arrays. Lists take slightly less storage, but elements near the end of a long list are slower to access. Array leaders allow you to have a homogeneous aggregate (the array) and a heterogeneous aggregate with named elements (the leader) tied together into one object.

defstruct allows you to specify to the constructor macro what the various elements of the structure should be initialized to. It also lets you give, in the **defstruct** form, default values for the initialization of each element.

The **defstruct** in Zetalisp also works in various dialects of Maclisp, and so it has some features that are not useful in Zetalisp. When possible, the Maclisp-specific features attempt to do something reasonable or harmless in Zetalisp, to make it easier to write code that will run equally well in Zetalisp and Maclisp. (Note that this **defstruct** is not necessarily the default one installed in Maclisp!)

2. How to Use defstruct

defstruct *Macro*

A call to **defstruct** looks like:

```
(defstruct (name option-1 option-2 ...)
           slot-description-1
           slot-description-2
           ...)
```

name must be a symbol; it is the name of the structure. It is given a **si:defstruct-description** property that describes the attributes and elements of the structure; this is intended to be used by programs that examine Lisp programs, that want to display the contents of structures in a helpful way. *name* is used for other things, described below.

Each *option* may be either a symbol, which should be one of the recognized option names, or a list, whose car should be one of the option names and the rest of which should be "arguments" to the option. See the document *Objects, Message Passing, and Flavors*. Some options have arguments that default; others require that arguments be given explicitly.

Each *slot-description* may be in any of three forms:

```
(1) slot-name
(2) (slot-name default-init)
(3) ((slot-name-1 byte-spec-1 default-init-1)
     (slot-name-2 byte-spec-2 default-init-2)
           ...)
```

Each *slot-description* allocates one element of the physical structure, even though in form (3) several slots are defined.

Each *slot-name* must always be a symbol; an accessor function is defined for each slot.

In form (1), *slot-name* simply defines a slot with the given name. An accessor function will be defined with the name *slot-name*. See the option **:conc-name**. Form (2) is similar, but allows a default initialization for the slot. See the section "Constructor Macros". Initialization is explained further in that section. Form (3) lets you pack several slots into a single element of the physical underlying structure, using the byte field feature of **defstruct**. See the section "Byte Fields".

Because evaluation of a **defstruct** form causes many functions and macros to be defined, you must take care not to define the same name with two different **defstruct** forms. A name can only have one function definition at a time; if it is redefined, the latest definition is the one that takes effect, and the earlier definition is clobbered. (This is no different from the requirement that each **defun** that is intended to define a distinct function must have a distinct name.)

To systematize this necessary carefulness, as well as for clarity in the code, it is conventional to prefix the names of all of the accessor functions with some text unique to the structure. In the space ship example, all the names started with **ship-**. See the section "Introduction to Structure Macros". The **:conc-name** option can be used to provide such prefixes automatically. Similarly, the conventional name for the constructor macro in the space ship example was **make-ship**, and the conventional name for the alterant macro was **alter-ship**. See the section "Alterant Macros".

The **describe-defstruct** function lets you examine an instance of a structure.

describe-defstruct *instance* &optional *name* *Function*

describe-defstruct takes an *instance* of a structure, and prints out a description of the instance, including the contents of each of its slots. *name* should be the name of the structure; you must provide the name of the structure so that **describe-defstruct** can know what structure *instance* is an instance of, and therefore figure out what the names of the slots of *instance* are.

If *instance* is a named structure, you do not have to provide *name*, since it is just the named structure symbol of *instance*. Normally the **describe** function calls **describe-defstruct** if it is asked to describe a named structure; however, some named structures have their own idea of how to describe themselves. See the section "Named Structures".

3. Options to defstruct

This section explains each of the options that can be given to **defstruct**.

Here is an example that shows the typical syntax of a call to **defstruct** that gives several options.

```
(defstruct (foo (:type :array)
                (:make-array (:type 'art-8b :leader-length 3))
                :conc-name
                (:size-symbol foo))
  a
  b)
```

:type The **:type** option specifies what kind of Lisp object will be used to implement the structure. It must be given one argument, which must be one of the symbols enumerated below, or a user-defined type. If the option itself is not provided, the type defaults to **:array**. You can define your own types. See the section "Extensions to **defstruct**".

:array	Use an array, storing components in the body of the array.
:named-array	Like **:array**, but make the array a named structure using the name of the structure as the named structure symbol. See the section "Named Structures". Element 0 of the array will hold the named structure symbol and so will not be used to hold a component of the structure.
:array-leader	Use an array, storing components in the leader of the array. (See the option **:make-array**.)
:named-array-leader	Like **:array-leader**, but make the array a named structure using the name of the structure as the named structure symbol. See the section "Named Structures". Element 1 of the leader will hold the named structure symbol and so will not be used to hold a component of the structure.
:list	Use a list.
:named-list	Like **:list**, but the first element of the list will hold the symbol that is the name of the structure and so will not be used as a component.
:fixnum-array	Like **:array**, but the type of the array is **art-32b**.
:flonum-array	Like **:array**, but the type of the array is **art-float**.
:tree	The structure is implemented out of a binary tree of conses, with the leaves serving as the slots.

:fixnum	This unusual type implements the structure as a single fixnum. The structure may only have one slot. This is only useful with the byte field feature; it lets you store a bunch of small numbers within fields of a fixnum, giving the fields names. See the section "Byte Fields".
:grouped-array	See the section "Grouped Arrays". This option is described there.

:constructor

This option takes one argument, which specifies the name of the constructor macro. If the argument is not provided or if the option itself is not provided, the name of the constructor is made by concatenating the string "**make-**" to the name of the structure. If the argument is provided and is **nil**, no constructor is defined. See the section "Constructor Macros". Use of the constructor macro is explained in that section.

:alterant

This option takes one argument, which specifies the name of the alterant macro. If the argument is not provided or if the option itself is not provided, the name of the alterant is made by concatenating the string "**alter-**" to the name of the structure. If the argument is provided and is **nil**, no alterant is defined. See the section "Alterant Macros". Use of the alterant macro is explained in that section.

:default-pointer

Normally, the accessors defined by **defstruct** expect to be given exactly one argument. However, if the **:default-pointer** argument is used, the argument to each accessor is optional. If you use an accessor in the usual way it will do the usual thing, but if you invoke it without its argument, it will behave as if you had invoked it on the result of evaluating the form that is the argument to the **:default-pointer** argument. Here is an example:

```
(defstruct (room (:default-pointer *default-room*))
   room-name
   room-contents)

(room-name x) ==> (aref x 0)
(room-name)   ==> (aref *default-room* 0)
```

If the argument to the **:default-pointer** argument is not given, it defaults to the name of the structure.

:conc-name

It is conventional to begin the names of all the accessor functions of a structure with a specific prefix, usually the name of the structure followed by a hyphen. The **:conc-name** option allows you to specify this prefix and have it concatenated onto the front of all the slot names to make the names of the accessor functions. The argument should be a symbol; its print-name is used as the prefix. If **:conc-name** is specified without an argument, the prefix will be the name of the structure followed by a hyphen. If you do not specify the **:conc-name** option, the names of the accessors are the same as the slot names, and it is up to you to name the slots according to some suitable convention.

The constructor and alterant macros are given slot names, not accessor names. It is important to keep this in mind when using **:conc-name**, since it causes the slot and accessor names to be different. Here is an example:

```
(defstruct (door :conc-name)
   knob-color
   width)

(setq d (make-door knob-color 'red width 5.0))

(door-knob-color d) ==> red
```

:include

This option is used for building a new structure definition as an extension of an old structure definition. Suppose you have a structure called **person** that looks like this:

```
(defstruct (person :conc-name)
   name
   age
   sex)
```

Now suppose you want to make a new structure to represent an astronaut. Since astronauts are people too, you would like them to also have the attributes of name, age, and sex, and you would like Lisp functions that operate on **person** structures to operate just as well on **astronaut** structures. You can do this by defining **astronaut** with the **:include** option, as follows:

```
(defstruct (astronaut (:include person))
   helmet-size
   (favorite-beverage 'tang))
```

The **:include** option inserts the slots of the included structure at the front of the list of slots for this structure. That is, an **astronaut** will have five slots; first the three defined in **person,** and then after those the two defined in **astronaut** itself. The accessor functions defined by the **person** structure can be applied to instances of the **astronaut** structure, and they will work correctly. The following examples illustrate how you can use **astronaut** structures:

```
(setq x (make-astronaut name 'buzz
                        age 45.
                        sex t
                        helmet-size 17.5))

(person-name x) => buzz
(favorite-beverage x) => tang
```

Note that the **:conc-name** option was *not* inherited from the included structure; it only applies to the accessor functions of **person** and not to those of **astronaut**. Similarly, the **:default-pointer** and **:but-first** options, as well as the **:conc-name** option, only apply to the accessor functions for the structure in which they are enclosed; they are not inherited if you **:include** a structure that uses them.

The argument to the **:include** option is required, and must be the name of some previously defined structure of the same type as this structure. **:include** does not work with structures of type **:tree** or of type **:grouped-array**.

The following is an advanced feature. Sometimes, when one structure includes another, the default values for the slots that came from the included structure are not what you want. The new structure can specify different default values for the included slots than the included structure specifies, by giving the **:include** option as:

`(:include` *name new-init-1 ... new-init-n*`)`

Each *new-init* is either the name of an included slot or a list of the form (*name-of-included-slot init-form*). If it is just a slot name, then in the new structure the slot will have no initial value. Otherwise its initial value form will be replaced by the *init-form*. The old (included) structure is unmodified.

For example, if we had wanted to define **astronaut** so that the default age for an astronaut is **45.**, then we could have said:

```
(defstruct (astronaut (:include person (age 45.)))
   helmet-size
   (favorite-beverage 'tang))
```

:named
This means that you want to use one of the "named" types. If you specify a type of **:array**, **:array-leader**, or **:list**, and give the **:named** option, then the **:named-array**, **:named-array-leader**, or **:named-list** type is used instead. Asking for type **:array** and giving the **:named** option as well is the same as asking for the type **:named-array**; the only difference is stylistic.

:make-array
If the structure being defined is implemented as an array, this option may be used to control those aspects of the array that are not otherwise constrained by **defstruct**. For example, you might want to control the area in which the array is allocated. Also, if you are creating a structure of type

:array-leader, you almost certainly want to specify the dimensions of the array to be created, and you may want to specify the type of the array. Of course, this option is only meaningful if the structure is, in fact, being implemented by an array.

The argument to the **:make-array** option should be a list of alternating keyword symbols to the **make-array** function, and forms whose values are the arguments to those keywords. See the function **make-array**. For example, **(:make-array (:type 'art-16b))** would request that the type of the array be **art-16b**. Note that the keyword symbol is *not* evaluated.

defstruct overrides any of the **:make-array** options that it needs to. For example, if your structure is of type **:array**, then **defstruct** supplies the size of that array regardless of what you say in the **:make-array** option.

Constructor macros for structures implemented as arrays all allow the keyword **:make-array** to be supplied. Attributes supplied therein override any **:make-array** option attributes supplied in the original **defstruct** form. If some attribute appears in neither the invocation of the constructor nor in the **:make-array** option to **defstruct**, then the constructor chooses appropriate defaults.

The **:make-array** option lets you control the initialization of arrays created by **defstruct** as instances of structures. **make-array** initializes the array before the constructor code does. Therefore, any initial value supplied via the new **:initial-value** keyword for **make-array** is overwritten in any slots where you gave **defstruct** an explicit initialization.

If a structure is of type **:array-leader**, you probably want to specify the dimensions of the array. The dimensions of an array are given to **:make-array** as a position argument rather than a keyword argument, so there is no way to specify them in the above syntax. To solve this problem, you can use the keyword **:dimensions** or the keyword **:length** (they mean the same thing), with a value that is anything acceptable as **make-array**'s first argument.

:times This option is used for structures of type **:grouped-array** to control the number of repetitions of the structure that will be allocated by the constructor macro. See the section "Grouped Arrays". The constructor macro also allows **:times** to be used as a keyword that overrides the value given in the original **defstruct** form. If **:times** appears in neither the invocation of the constructor nor in the **:make-array** option to **defstruct**, then the constructor only allocates one instance of the structure.

:size-symbol
The **:size-symbol** option allows you to specify a global variable whose value will be the "size" of the structure; this variable is declared with **defconst**. The exact meaning of the size varies, but in general this number is the one you would need to know if you were going to allocate one of these structures yourself. The symbol has this value both at compile time and at run time.

If this option is present without an argument, then the name of the structure is concatenated with "**-size**" to produce the symbol.

:size-macro
This is similar to the **:size-symbol** option. A macro of no arguments is defined that expands into the size of the structure. The name of this macro defaults as with **:size-symbol**.

:initial-offset
This allows you to tell **defstruct** to skip over a certain number of slots before it starts allocating the slots described in the body. This option requires an argument (which must be a fixnum) that is the number of slots you want **defstruct** to skip. To use this option, you must have some familiarity with how **defstruct** is implementing your structure; otherwise, you will be unable to make use of the slots that **defstruct** has left unused.

:but-first
This option is best explained by example:

```
(defstruct (head (:type :list)
                 (:default-pointer person)
                 (:but-first person-head))
  nose
  mouth
  eyes)
```

The accessors expand like this:

```
(nose x)      ==> (car (person-head x))
(nose)        ==> (car (person-head person))
```

The idea is that **:but-first**'s argument will be an accessor from some other structure, and it is never expected that this structure will be found outside that slot of that other structure. Actually, you can use any one-argument function, or a macro that acts like a one-argument function. It is an error for the **:but-first** option to be used without an argument.

:callable-accessors
This option controls whether accessors are really functions, and therefore "callable", or whether they are really macros. With an argument of **t**, or with no argument, or if the option is not provided, then the accessors are really functions. Specifically, they are **substs**, so that they have all the efficiency of macros in compiled programs, while still being function objects that can be manipulated (passed to **mapcar**, and so on). If the argument is **nil** then the accessors will really be macros.

:eval-when
Normally the functions and macros defined by **defstruct** are defined at eval time, compile time, and load time. This option allows you to control this behavior. The argument to the **:eval-when** option is just like the list that is the first subform of an **eval-when** special form. For example: **(:eval-when (:eval :compile))** causes the functions and macros to be defined only when the code is running interpreted or inside the compiler.

:property

For each structure defined by defstruct, a property list is maintained for the recording of arbitrary properties about that structure. (That is, there is one property list per structure definition, not one for each instantiation of the structure.)

The **:property** option can be used to give a **defstruct** an arbitrary property. **(:property** *property-name value***)** gives the **defstruct** a *property-name* property of *value*. Neither argument is evaluated. To access the property list, the user will have to look inside the **defstruct-description** structure. See the section "The **si:defstruct-description** Structure".

:print The **:print** option gives you implementation-independent control over the printed representation of a structure.

```
(defstruct (foo :named
                (:print "#<Foo ~S ~S>" (foo-a foo) (foo-b foo)))
  foo-a
  foo-b)
```

The **:print** option takes a format string and its arguments. The arguments are evaluated in an environment in which the name symbol for the structure is bound to the structure instance being printed.

People used to use a **named-structure-invoke** handler to define **:print** handlers. This is no longer necessary; the **:print** option does it for you.

:predicate

The **:predicate** option causes **defstruct** to generate a predicate that recognizes instances of the structure. The first example defines a single-argument function, **foo-p,** that returns **t** only for instances of structure **foo.** The second example defines a function called **is-it-a-foo?**.

```
(defstruct (foo :named :predicate)
  foo-a
  foo-b)
(defstruct (foo :named (:predicate is-it-a-foo?))
  foo-a
  foo-b)
```

The **:predicate** option has one optional argument, the name for the function being generated. The default name for the generated function is formed by appending **-p** to the structure name.

The **:predicate** option is restricted to work only for named types.

:copier

The **:copier** option causes **defstruct** to generate a function for copying instances of the structure.

```
(defstruct (foo (:type list) :copier)
  foo-a
  foo-b)
```

This example would generate a function named copy-foo, with a definition approximately like this:

```
(defun copy-foo (x)
  (list (car x) (cadr x)))
```

type In addition to the options listed above, any currently defined type (any legal argument to the **:type** option) can be used as an option. This is mostly for compatibility with the old version of **defstruct**. It allows you to say just *type* instead of **(:type** *type***)**. It is an error to give an argument to one of these options.

other Finally, if an option is not found among those listed above, **defstruct** checks the property list of the name of the option to see if it has a non-**nil** **:defstruct-option** property. If it does have such a property, then if the option was of the form **(***option-name value***)**, it is treated just like **(:property** *option-name value***)**. That is, the **defstruct** is given an *option-name* property of *value*. It is an error to use such an option without a value.

This provides a primitive way for you to define your own options to **defstruct**, particularly in connection with user-defined types. See the section "Extensions to **defstruct**". Several of the options listed above are actually implemented using this mechanism.

4. Using the Constructor and Alterant Macros

After you have defined a new structure with **defstruct**, you can create instances of this structure using the constructor macro, and you can alter the values of its slots using the alterant macro. By default, **defstruct** defines both the constructor and the alterant, forming their names by concatenating "**make-**" and "**alter-**", respectively, onto the name of the structure. You can specify the names yourself by passing the name you want to use as the argument to the **:constructor** or **:alterant** options, or specify that you do not want the macro created at all by passing **nil** as the argument.

4.1 Constructor Macros

A call to a constructor macro, in general, has the form:

```
(name-of-constructor-macro
        symbol-1 form-1
        symbol-2 form-2
        ...)
```

Each *symbol* may be either the name of a *slot* of the structure, or a specially recognized keyword. All the *forms* are evaluated.

If *symbol* is the name of a *slot* (*not* the name of an accessor), then that element of the created structure will be initialized to the value of *form*. If no *symbol* is present for a given slot, then the slot will be initialized to the result of evaluating the default initialization form specified in the call to **defstruct**. (In other words, the initialization form specified to the constructor overrides the initialization form specified to **defstruct**.) If the **defstruct** itself also did not specify any initialization, the element's initial value is undefined. You should always specify the initialization, either in the **defstruct** or in the constructor macro, if you care about the initial value of the slot.

Notes: The order of evaluation of the initialization forms is *not* necessarily the same as the order in which they appear in the constructor call, nor the order in which they appear in the **defstruct**; you should make sure your code does not depend on the order of evaluation. The forms are reevaluated on every constructor-macro call, so that if, for example, the form **(gensym)** were used as an initialization form, either in a call to a constructor macro or as a default initialization in the **defstruct**, then every call to the constructor macro would create a new symbol.

There are two symbols that are specially recognized by the constructor. They are **:make-array**, which should only be used for **:array** and **:array-leader** type structures (or the named versions of those types), and **:times**, which should only be used for **:grouped-array** type structures. If one of these symbols appears instead of a slot name, then it is interpreted just as the **:make-array** option or the **:times** option, and it overrides what was requested in that option. For example:

```
(make-ship ship-x-position 10.0
           ship-y-position 12.0
           :make-array (:leader-length 5 :area disaster-area))
```

4.2 By-position Constructor Macros

If the **:constructor** option is given as **(:constructor** *name arglist***)**, then instead of making a keyword-driven constructor, **defstruct** defines a "function style" constructor, taking arguments whose meaning is determined by the argument's position rather than by a keyword. The *arglist* is used to describe what the arguments to the constructor will be. In the simplest case something like **(:constructor make-foo (a b c))** defines **make-foo** to be a three-argument constructor macro whose arguments are used to initialize the slots named **a**, **b**, and **c**.

In addition, the keywords **&optional, &rest**, and **&aux** are recognized in the argument list. They work in the way you might expect, but there are a few fine points worthy of explanation:

```
(:constructor make-foo
        (a &optional b (c 'sea) &rest d &aux e (f 'eff)))
```

This defines **make-foo** to be a constructor of one or more arguments. The first argument is used to initialize the **a** slot. The second argument is used to initialize the **b** slot. If there is no second argument, then the default value given in the body of the **defstruct** (if given) is used instead. The third argument is used to initialize the **c** slot. If there is no third argument, then the symbol **sea** is used instead. Any arguments following the third argument are collected into a list and used to initialize the **d** slot. If there are three or fewer arguments, then **nil** is placed in the **d** slot. The **e** slot *is not initialized*; its initial value is undefined. Finally, the **f** slot is initialized to contain the symbol **eff**.

The actions taken in the **b** and **e** cases were carefully chosen to allow the user to specify all possible behaviors. Note that the **&aux** "variables" can be used to completely override the default initializations given in the body.

Since there is so much freedom in defining constructors this way, it would be cruel to only allow the **:constructor** option to be given once. So, by special dispensation, you are allowed to give the **:constructor** option more than once, so that you can define several different constructors, each with a different syntax.

Note that even these "function-style" constructors do not guarantee that their arguments will be evaluated in the order that you wrote them. Also note that you cannot specify the **:make-array** nor **:times** information in this form of constructor macro.

4.3 Alterant Macros

A call to the alterant macro, in general, has the form:

(*name-of-alterant-macro instance-form*
  slot-name-1 form-1
  slot-name-2 form-2
  ...)

instance-form is evaluated, and should return an instance of the structure. Each *form* is evaluated, and the corresponding slot is changed to have the result as its new value. The slots are altered after all the *forms* are evaluated, so you can exchange the values of two slots, as follows:

```
(alter-ship enterprise
    ship-x-position (ship-y-position enterprise)
    ship-y-position (ship-x-position enterprise))
```

As with the constructor macro, the order of evaluation of the *forms* is undefined. Using the alterant macro can produce more efficient Lisp than using consecutive **setf**s when you are altering two byte fields of the same object, or when you are using the **:but-first** option.

You can use alterant macros on structures whose accessors require additional arguments. Put the additional arguments before the list of slots and values, in the same order as required by the accessors.

5. Byte Fields

The byte field feature of **defstruct** allows you to specify that several slots of your structure are bytes in an integer stored in one element of the structure. See the section "Byte Manipulation Functions". For example, suppose we had the following structure:

```
(defstruct (phone-book-entry (:type :list))
  name
  address
  (area-code 617.)
  exchange
  line-number)
```

This will work correctly. However, it wastes space. Area codes and exchange numbers are always less than **1000.**, and so both can fit into **10.** bit fields when expressed as binary numbers. Since Lisp Machine fixnums have (more than) **20.** bits, both of these values can be packed into a single fixnum. To tell **defstruct** to do so, you can change the structure definition to the following:

```
(defstruct (phone-book-entry (:type :list))
  name
  address
  ((area-code #o1212 617.)
   (exchange #o0012))
  line-number)
```

The magic octal numbers **#o1212** and **#o0012** are byte specifiers to be used with the functions **ldb** and **dpb**. The accessors, constructor, and alterant will now operate as follows:

```
(area-code pbe)       ==> (ldb #o1212 (caddr pbe))
(exchange pbe)  ==> (ldb #o0012 (caddr pbe))

(make-phone-book-entry
   name "Fred Derf"
   address "259 Octal St."
   exchange ex
   line-number 7788.)

==> (list "Fred Derf" "259 Octal St." (dpb ex 12 2322000) 17154)
```

```
(alter-phone-book-entry pbe
   area-code ac
   exchange ex)

==> ((lambda (g0530)
       (setf (nth 2 g0530)
             (dpb ac 1212 (dpb ex 12 (nth 2 g0530)))))
     pbe)
```

Note that the alterant macro is optimized to only read and write the second element of the list once, even though you are altering two different byte fields within it. This is more efficient than using two **setf**s. Additional optimization by the alterant macro occurs if the byte specifiers in the **defstruct** slot descriptions are constants. However, you need not worry about the details of how the alterant macro does its work.

If the byte specifier is **nil**, then the accessor will be defined to be the usual kind that accesses the entire Lisp object, thus returning all the byte field components as a fixnum. These slots may have default initialization forms.

The byte specifier need not be a constant; a variable or, indeed, any Lisp form, is legal as a byte specifier. It will be evaluated each time the slot is accessed. Of course, unless you are doing something very strange you will not want the byte specifier to change between accesses.

Constructor macros initialize words divided into byte fields as if they were deposited in the following order:

1. Initializations for the entire word given in the **defstruct** form.

2. Initializations for the byte fields given in the **defstruct** form.

3. Initializations for the entire word given in the constructor macro form.

4. Initializations for the byte fields given in the constructor macro form.

Alterant macros work similarly: the modification for the entire Lisp object is done first, followed by modifications to specific byte fields. If any byte fields being initialized or altered overlap each other, the action of the constructor and alterant will be unpredictable.

6. Grouped Arrays

The grouped array feature allows you to store several instances of a structure side-by-side within an array. This feature is somewhat limited; it does not support the **:include** and **:named** options.

The accessor functions are defined to take an extra argument, which should be an integer, and is the index into the array of where this instance of the structure starts. This index should normally be a multiple of the size of the structure, for things to make sense. Note that the index is the *first* argument to the accessor function and the structure is the *second* argument, the opposite of what you might expect. This is because the structure is **&optional** if the **:default-pointer** option is used.

Note that the "size" of the structure (for purposes of the **:size-symbol** and **:size-macro** options) is the number of elements in *one* instance of the structure; the actual length of the array is the product of the size of the structure and the number of instances. The number of instances to be created by the constructor macro is given as the argument to the **:times** option to **defstruct**, or the **:times** keyword of the constructor macro.

7. Named Structures

The *named structure* feature provides a very simple form of user-defined data type. Any array may be made a named structure, although usually the **:named** option of **defstruct** is used to create named structures. The principal advantages of a named structure are that it has a more informative printed representation than a normal array and that the **describe** function knows how to give a detailed description of it. (You do not have to use **describe-defstruct**, because **describe** can figure out what the names of the slots of the structure are by looking at the named structure's name.) Because of these improved user-interface features it is recommended that "system" data structures be implemented with named structures.

Another kind of user-defined data type, more advanced but less efficient when just used as a record structure, is provided by the *flavor* feature. See the document *Objects, Message Passing, and Flavors.*

A named structure has an associated symbol, called its "named structure symbol", that represents what user-defined type it is an instance of; the **typep** function, applied to the named structure, will return this symbol. If the array has a leader, then the symbol is found in element 1 of the leader; otherwise it is found in element 0 of the array. (Note: if a numeric-type array is to be a named structure, it must have a leader, since a symbol cannot be stored in any element of a numeric array.)

If you call **typep** with two arguments, the first being an instance of a named structure and the second being its named structure symbol, **typep** will return **t**. **t** will also be returned if the second argument is the named structure symbol of a **:named defstruct** included (using the **:include** option), directly or indirectly, by the **defstruct** for this structure. For example, if the structure **astronaut** includes the structure **person**, and **person** is a named structure, then giving **typep** an instance of an **astronaut** as the first argument, and the symbol **person** as the second argument, will return **t**. This reflects the fact that an astronaut is, in fact, a person, as well as being an astronaut.

You may associate with a named structure a function that will handle various operations that can be done on the named structure. Currently, you can control how the named structure is printed, and what **describe** will do with it.

To provide such a handler function, make the function be the **named-structure-invoke** property of the named structure symbol. The functions that know about named structures will apply this handler function to several arguments. The first is a "keyword" symbol to identify the calling function, and the second is the named structure itself. The rest of the arguments passed depend on the caller; any named structure function should have a "**&rest**" parameter to absorb any extra arguments that might be passed. Just what the function is expected to do depends on the keyword it is passed as its first argument. The following are the keywords defined at present:

:which-operations
Should return a list of the names of the operations the function handles.

:print-self
The arguments are **:print-self,** the named structure, the stream to output to, the current depth in list-structure, and **t** if slashification is enabled (**prin1** versus **princ**). The printed representation of the named structure should be output to the stream. If the named structure symbol is not defined as a function, or **:print-self** is not in its **:which-operations** list, the printer will default to a reasonable printed representation, namely:

#<named-structure-symbol octal-address>

:describe
The arguments are **:describe** and the named structure. It should output a description of itself to **standard-output**. If the named structure symbol is not defined as a function, or **:describe** is not in its **:which-operations** list, the describe system will check whether the named structure was created by using the **:named** option of **defstruct**; if so, the names and values of the structure's fields will be enumerated.

Here is an example of a simple named-structure handler function:

```
(defun (person named-structure-invoke) (op self &rest args)
  (selectq op
    (:which-operations '(:print-self))
    (:print-self
      (format (first args)
              (if (third args) "#<person ~A>" "~A")
              (person-name self)))
    (otherwise (ferror nil "Illegal operation ~S" op))))
```

For this definition to have any effect, the person **defstruct** used as an example earlier must be modified to include the **:named** attribute.

This handler causes a person structure to include its name in its printed representation; it also causes **princ** of a person to print just the name, with no "#<" syntax. Even though the astronaut structure of our examples **:includes** the person structure, this named-structure handler will not be invoked when an astronaut is printed, and an astronaut will not include his name in his printed representation. This is because named structures are not as general as flavors. See the document *Objects, Message Passing, and Flavors.*

The following functions operate on named structures.

named-structure-p *x* *Function*
This semi-predicate returns **nil** if *x* is not a named structure; otherwise it returns *x*'s named structure symbol.

named-structure-symbol *x* *Function*

x should be a named structure. This returns *x*'s named structure symbol: if *x* has an array leader, element 1 of the leader is returned, otherwise element 0 of the array is returned.

make-array-into-named-structure *array* *Function*

array is made to be a named structure, and is returned.

named-structure-invoke *operation structure* &rest *args* *Function*

operation should be a keyword symbol, and *structure* should be a named structure. The handler function of the named structure symbol, found as the value of the **named-structure-invoke** property of the symbol, is called with appropriate arguments. (This function used to take its first two arguments in the opposite order, and that argument order will continue to work indefinitely, but it should not be used in new programs.)

See also the **:named-structure-symbol** keyword to **make-array**.

8. The si:defstruct-description Structure

This section discusses the internal structures used by **defstruct** that might be useful to programs that want to interface to **defstruct** nicely. For example, if you want to write a program that examines structures and displays them the way **describe** and the Inspector do, your program will work by examining these structures. The information in this section is also necessary if you are thinking of defining your own structure types.

Whenever the user defines a new structure using **defstruct**, **defstruct** creates an instance of the **si:defstruct-description** structure. This structure can be found as the **si:defstruct-description** property of the name of the structure; it contains such useful information as the name of the structure, the number of slots in the structure, and so on.

The **si:defstruct-description** structure is defined as follows, in the **system-internals** package (also called the **si** package): (This is a simplified version of the real definition. There are other slots in the structure that we are not telling you about.)

```
(defstruct (defstruct-description
                (:default-pointer description)
                (:conc-name defstruct-description-))
           name
           size
           property-alist
           slot-alist)
```

The **name** slot contains the symbol supplied by the user to be the name of the structure, such as **spaceship** or **phone-book-entry**.

The **size** slot contains the total number of locations in an instance of this kind of structure. This is *not* the same number as that obtained from the **:size-symbol** or **:size-macro** options to **defstruct**. A named structure, for example, usually uses up an extra location to store the name of the structure, so the **:size-macro** option will get a number one larger than that stored in the **defstruct** description.

The **property-alist** slot contains an alist with pairs of the form (*property-name* . *property*) containing properties placed there by the **:property** option to **defstruct** or by property names used as options to **defstruct**. (See the option **:property**.)

The **slot-alist** slot contains an alist of pairs of the form (*slot-name* . *slot-description*). A *slot-description* is an instance of the **defstruct-slot-description** structure. The **defstruct-slot-description** structure is defined something like this, also in the **si** package: (This is a simplified version of the real definition. There are other slots in the structure which we aren't telling you about.)

```
(defstruct (defstruct-slot-description
                (:default-pointer slot-description)
                (:conc-name defstruct-slot-description-))
           number
           ppss
           init-code
           ref-macro-name)
```

The **number** slot contains the number of the location of this slot in an instance of the structure. Locations are numbered starting with **0**, and continuing up to one less than the size of the structure. The actual location of the slot is determined by the reference-consing function associated with the type of the structure. See the section "Options to **defstruct-define-type**".

The **ppss** slot contains the byte specifier code for this slot if this slot is a byte field of its location. If this slot is the entire location, then the **ppss** slot contains **nil**.

The **init-code** slot contains the initialization code supplied for this slot by the user in the **defstruct** form. If there is no initialization code for this slot, then the init-code slot contains the symbol **si:%%defstruct-empty%%**.

The **ref-macro-name** slot contains the symbol that is defined as a macro or a subst that expands into a reference to this slot (that is, the name of the accessor function).

9. Extensions to defstruct

The macro **defstruct-define-type** can be used to teach **defstruct** about new types that it can use to implement structures.

defstruct-define-type *Macro*

This macro is used for teaching **defstruct** about new types; it is described in the rest of this chapter.

9.1 An Example of defstruct-define-type

Let us start by examining a sample call to **defstruct-define-type**. This is how the **:list** type of structure might have been defined:

```
(defstruct-define-type :list
       (:cons (initialization-list description keyword-options)
             :list
             `(list . ,initialization-list))
       (:ref (slot-number description argument)
             `(nth ,slot-number ,argument)))
```

This is the simplest possible form of **defstruct-define-type**. It provides **defstruct** with two Lisp forms: one for creating forms to construct instances of the structure, and one for creating forms to become the bodies of accessors for slots of the structure.

The keyword **:cons** is followed by a list of three variables that will be bound while the constructor-creating form is evaluated. The first, **initialization-list**, will be bound to a list of the initialization forms for the slots of the structure. The second, **description**, will be bound to the **defstruct-description** structure for the structure. See the section "The **si:defstruct-description** Structure". For a description of the third variable, **keyword-options**, and the **:list** keyword: See the section "Options to **defstruct-define-type**".

The keyword **:ref** is followed by a list of three variables that will be bound while the accessor-creating form is evaluated. The first, **slot-number**, will bound to the number of the slot that the new accessor should reference. The second, **description**, will be bound to the **defstruct-description** structure for the structure. The third, **argument**, will be bound to the form that was provided as the argument to the accessor.

9.2 Syntax of defstruct-define-type

The syntax of **defstruct-define-type** is:

```
(defstruct-define-type type
        option-1
        option-2
        ...)
```

where each *option* is either the symbolic name of an option or a list of the form (*option-name* . *rest*). Different options interpret *rest* in different ways. The symbol *type* is given an **si:defstruct-type-description** property of a structure that describes the type completely.

9.3 Options to defstruct-define-type

This section is a catalog of all the options currently known about by **defstruct-define-type**.

:cons The **:cons** option to **defstruct-define-type** is how you supply **defstruct** with the code necessary to cons up a form that will construct an instance of a structure of this type.

The **:cons** option has the syntax:

```
(:cons (inits description keywords) kind
        body)
```

body is some code that should construct and return a piece of code that will construct, initialize, and return an instance of a structure of this type.

The symbol *inits* will be bound to the information that the constructor conser should use to initialize the slots of the structure. The exact form of this argument is determined by the symbol *kind*. There are currently two kinds of initialization. There is the **:list** kind, where *inits* is bound to a list of initializations, in the correct order, with **nils** in uninitialized slots. And there is the **:alist** kind, where *inits* is bound to an alist with pairs of the form (*slot-number* . *init-code*).

The symbol *description* will be bound to the instance of the **defstruct-description** structure that **defstruct** maintains for this particular structure. See the section "The **si:defstruct-description** Structure". This is so that the constructor conser can find out such things as the total size of the structure it is supposed to create.

The symbol *keywords* will be bound to an alist with pairs of the form (*keyword* . *value*), where each *keyword* was a keyword supplied to the constructor macro that was not the name of a slot, and *value* was the Lisp object that followed the keyword. This is how you can make your own special keywords, like the existing **:make-array** and **:times** keywords. See the section "Constructor Macros". You specify the list of acceptable keywords with the **:keywords** option.

It is an error not to supply the **:cons** option to **defstruct-define-type**.

:ref The **:ref** option to **defstruct-define-type** is how you supply **defstruct** with the necessary code that it needs to cons up a form that will reference an instance of a structure of this type.

The **:ref** option has the syntax:

```
(:ref (number description arg-1 ... arg-n)
      body)
```

body is some code that should construct and return a piece of code that will reference an instance of a structure of this type.

The symbol *number* will be bound to the location of the slot that is to be referenced. This is the same number that is found in the number slot of the **defstruct-slot-description** structure. See the section "The **si:defstruct-description** Structure".

The symbol *description* will be bound to the instance of the **defstruct-description** structure that **defstruct** maintains for this particular structure.

The symbols *arg-i* are bound to the forms supplied to the accessor as arguments. Normally there should be only one of these. The *last* argument is the one that will be defaulted by the **:default-pointer** option. See the section "Options to **defstruct**". **defstruct** will check that the user has supplied exactly *n* arguments to the accessor function before calling the reference consing code.

It is an error not to supply the **:ref** option to **defstruct-define-type**.

:overhead

The **:overhead** option to **defstruct-define-type** is how you declare to **defstruct** that the implementation of this particular type of structure "uses up" some number of locations in the object actually constructed. This option is used by various "named" types of structures that store the name of the structure in one location.

The syntax of **:overhead** is: **(:overhead** *n***)** where *n* is a fixnum that says how many locations of overhead this type needs.

This number is used only by the **:size-macro** and **:size-symbol** options to **defstruct**. See the section "Options to **defstruct**".

:named

The **:named** option to **defstruct-define-type** controls the use of the **:named** option to **defstruct**. With no argument, the **:named** option means that this type is an acceptable "named structure". With an argument, as in

(:named *type-name***)**, the symbol *type-name* should be the name of some other structure type that **defstruct** should use if someone asks for the named version of this type. (For example, in the definition of the **:list** type the **:named** option is used like this: **(:named :named-list)**.)

:keywords

The **:keywords** option to **defstruct-define-type** allows you to define additional constructor keywords for this type of structure. (The **:make**-array constructor keyword for structures of type **:array** is an example.) The syntax is: **(:keywords** *keyword-1* **...** *keyword-n***)** where each *keyword* is a symbol that the constructor conser expects to find in the *keywords* alist (explained above).

:defstruct

The **:defstruct** option to **defstruct-define-type** allows you to run some code and return some forms as part of the expansion of the **defstruct** macro.

The **:defstruct** option has the syntax:

```
(:defstruct (description)
            body)
```

body is a piece of code that will be run whenever **defstruct** is expanding a **defstruct** form that defines a structure of this type. The symbol *description* will be bound to the instance of the **defstruct-description** structure that **defstruct** maintains for this particular structure.

The value returned by the *body* should be a *list* of forms to be included with those that the **defstruct** expands into. Thus, if you only want to run some code at **defstruct**-expand time, and you do not want to actually output any additional code, then you should be careful to return **nil** from the code in this option.

:predicate

The **:predicate** option specifies how to construct a **:predicate** option for **defstruct**.

```
(:predicate (description name)
           `(defun ,name (x)
              (and (frobbozp x)
                   (eq (frobbozref x 0)
                       ',(defstruct-description-name)))))
```

The syntax for the option follows.

```
(:predicate (description name)
             body)
```

The variable *description* is bound to the **defstruct-description** structure maintained for the structure for which we are to generate a predicate. The variable *name* is bound to the symbol that is to be defined as a predicate. *body* is a piece of code that is evaluated to return the defining form for the predicate.

:copier

The **:copier** option specifies how to copy a particular type of structure for situations when it is necessary to provide a copying function other than the one that **defstruct** would generate.

```
(:copier (description name)
  `(fset-carefully ',name 'copy-frobboz))
```

The syntax for the option follows.

(:copier (*description name*)
          body)

description is bound to an instance of the defstruct-description structure, *name* is bound to the symbol to be defined, and *body* is some code to evaluate to get the defining form.

Index

symbolics™

FLAV Objects, Message Passing, and Flavors

Cambridge, Massachusetts

Objects, Message Passing, and Flavors

# 990052

February 1984

This document corresponds to Release 5.0.

This document was prepared by the Documentation Group of Symbolics, Inc.

Printing year and number: 87 86 85 84 9 8 7 6 5 4 3 2 1

Table of Contents

1. Introduction

The object oriented programming style used in the Smalltalk and Actor families of languages is available in Zetalisp, and used by the Lisp Machine software system. Its purpose is to perform *generic operations* on objects. Part of its implementation is simply a convention in procedure calling style; part is a powerful language feature, called Flavors, for defining abstract objects. This chapter attempts to explain what programming with objects and with message passing means, the various means of implementing these in Zetalisp, and when you should use them. It assumes no prior knowledge of any other languages.

2. Objects

When writing a program, it is often convenient to model what the program does in terms of *objects*: conceptual entities that can be likened to real-world things. Choosing what objects to provide in a program is very important to the proper organization of the program. In an object-oriented design, specifying what objects exist is the first task in designing the system. In a text editor, the objects might be "pieces of text", "pointers into text", and "display windows". In an electrical design system, the objects might be "resistors", "capacitors", "transistors", "wires", and "display windows". After specifying what objects there are, the next task of the design is to figure out what operations can be performed on each object. In the text editor example, operations on "pieces of text" might include inserting text and deleting text; operations on "pointers into text" might include moving forward and backward; and operations on "display windows" might include redisplaying the window and changing with which "piece of text" the window is associated.

In this model, we think of the program as being built around a set of objects, each of which has a set of operations that can be performed on it. More rigorously, the program defines several *types* of object (the editor above has three types), and it can create many *instances* of each type (that is, there can be many pieces of text, many pointers into text, and many windows). The program defines a set of types of object, and the operations that can be performed on any of the instances of each type.

This should not be wholly unfamiliar to the reader. Earlier in this manual, we saw a few examples of this kind of programming. A simple example is disembodied property lists, and the functions **get**, **putprop**, and **remprop**. The disembodied property list is a type of object; you can instantiate one with **(cons nil nil)** (that is, by evaluating this form you can create a new disembodied property list); there are three operations on the object, namely **get**, **putprop**, and **remprop**. Another example in the manual was the first example of the use of **defstruct**, which was called a **ship**. **defstruct** automatically defined some operations on this object: the operations to access its elements. We could define other functions that did useful things with **ships**, such as computing their speed, angle of travel, momentum, or velocity, stopping them, moving them elsewhere, and so on.

In both cases, we represent our conceptual object by one Lisp object. The Lisp object we use for the representation has *structure*, and refers to other Lisp objects. In the property list case, the Lisp object is a list with alternating indicators and values; in the **ship** case, the Lisp object is an array whose details are taken care of by **defstruct**. In both cases, we can say that the object keeps track of an *internal state*, which can be *examined* and *altered* by the operations available for that type of object. **get** examines the state of a property list, and **putprop** alters it; **ship-x-position** examines the state of a ship, and **(setf (ship-mass ship) 5.0)** alters it.

We have now seen the essence of object-oriented programming. A conceptual object is modelled by a single Lisp object, which bundles up some state information. For every type of object, there is a set of operations that can be performed to examine or alter the state of the object.

3. Modularity

An important benefit of the object-oriented style is that it lends itself to a particularly simple and lucid kind of modularity. If you have modular programming constructs and techniques available, it helps and encourages you to write programs that are easy to read and understand, and so are more reliable and maintainable. Object-oriented programming lets a programmer implement a useful facility that presents the caller with a set of external interfaces, without requiring the caller to understand how the internal details of the implementation work. In other words, a program that calls this facility can treat the facility as a black box; the program knows what the facility's external interfaces guarantee to do, and that is all it knows.

For example, a program that uses disembodied property lists never needs to know that the property list is being maintained as a list of alternating indicators and values; the program simply performs the operations, passing them inputs and getting back outputs. The program only depends on the external definition of these operations: it knows that if it **putprops** a property, and doesn't **remprop** it (or **putprop** over it), then it can do **get** and be sure of getting back the same thing it put in. The important thing about this hiding of the details of the implementation is that someone reading a program that uses disembodied property lists need not concern himself with how they are implemented; he need only understand what they undertake to do. This saves the programmer a lot of time, and lets him concentrate his energies on understanding the program he is working on. Another good thing about this hiding is that the representation of property lists could be changed, and the program would continue to work. For example, instead of a list of alternating elements, the property list could be implemented as an association list or a hash table. Nothing in the calling program would change at all.

The same is true of the **ship** example. The caller is presented with a collection of operations, such as **ship-x-position**, **ship-y-position**, **ship-speed**, and **ship-direction**; it simply calls these and looks at their answers, without caring how they did what they did. In our example above, **ship-x-position** and **ship-y-position** would be accessor functions, defined automatically by **defstruct**, while **ship-speed** and **ship-direction** would be functions defined by the implementor of the **ship** type. The code might look like this:

```
(defstruct (ship)
  ship-x-position
  ship-y-position
  ship-x-velocity
  ship-y-velocity
  ship-mass)

(defun ship-speed (ship)
  (sqrt (+ (^ (ship-x-velocity ship) 2)
           (^ (ship-y-velocity ship) 2))))

(defun ship-direction (ship)
  (atan (ship-y-velocity ship)
        (ship-x-velocity ship)))
```

The caller need not know that the first two functions were structure accessors and that the second two were written by hand and do arithmetic. Those facts would not be considered part of the black box characteristics of the implementation of the **ship** type. The **ship** type does not guarantee which functions will be implemented in which ways; such aspects are not part of the contract between **ship** and its callers. In fact, **ship** could have been written this way instead:

```
(defstruct (ship)
  ship-x-position
  ship-y-position
  ship-speed
  ship-direction
  ship-mass)

(defun ship-x-velocity (ship)
  (* (ship-speed ship) (cos (ship-direction ship))))

(defun ship-y-velocity (ship)
  (* (ship-speed ship) (sin (ship-direction ship))))
```

In this second implementation of the **ship** type, we have decided to store the velocity in polar coordinates instead of rectangular coordinates. This is purely an implementation decision; the caller has no idea which of the two ways the implementation works, because he just performs the operations on the object by calling the appropriate functions.

We have now created our own types of objects, whose implementations are hidden from the programs that use them. Such types are usually referred to as *abstract types*. The object-oriented style of programming can be used to create abstract types by hiding the implementation of the operations, and simply documenting what the operations are defined to do.

Some more terminology: the quantities being held by the elements of the **ship** structure are referred to as *instance variables*. Each instance of a type has the same operations defined on it; what distinguishes one instance from another (besides identity (**eq**ness)) is the values that reside in its instance variables. The example above illustrates that a caller of operations does not know what the instance variables are; our two ways of writing the **ship** operations have different instance variables, but from the outside they have exactly the same operations.

One might ask: "But what if the caller evaluates **(aref ship 2)** and notices that he gets back the x-velocity rather than the speed? Then he can tell which of the two implementations were used." This is true; if the caller were to do that, he could tell. However, when a facility is implemented in the object-oriented style, only certain functions are documented and advertised: the functions which are considered to be operations on the type of object. The contract from **ship** to its callers only speaks about what happens if the caller calls these functions. The contract makes no guarantees at all about what would happen if the caller were to start poking around on his own using **aref**. A caller who does so *is in error*; he is depending on something that is not specified in the contract. No guarantees were ever made about the results of such action, and so anything may happen; indeed, **ship** may get reimplemented overnight, and the code that does the **aref** will have a different effect entirely and probably stop working. This example shows why the concept of a contract between a callee and a caller is important: the contract is what specifies the interface between the two modules.

Unlike some other languages that provide abstract types, Zetalisp makes no attempt to have the language automatically forbid constructs that circumvent the contract. This is intentional. One reason for this is that the Lisp Machine is an interactive system, and so it is important to be able to examine and alter internal state interactively (usually from a debugger). Furthermore, there is no strong distinction between the "system" programs and the "user" programs on the Lisp Machine; users are allowed to get into any part of the language system and change what they want to change.

In summary: by defining a set of operations, and making only a specific set of external entrypoints available to the caller, the programmer can create his own abstract types. These types can be useful facilities for other programs and programmers. Since the implementation of the type is hidden from the callers, modularity is maintained, and the implementation can be changed easily.

We have hidden the implementation of an abstract type by making its operations into functions which the user may call. The important thing is not that they are functions--in Lisp everything is done with functions. The important thing is that we have defined a new conceptual operation and given it a name, rather than requiring anyone who wants to do the operation to write it out step-by-step. Thus we say **(ship-x-velocity s)** rather than **(aref s 2)**.

It is just as true of such abstract-operation functions as of ordinary functions that sometimes they are simple enough that we want the compiler to compile special code for them rather than really calling the function. (Compiling special code like this is often called *open-coding.*) The compiler is directed to do this through use of macros, defsubsts, or optimizers. **defstruct** arranges for this kind of special compilation for the functions that get the instance variables of a structure.

When we use this optimization, the implementation of the abstract type is only hidden in a certain sense. It does not appear in the Lisp code written by the user, but does appear in the compiled code. The reason is that there may be some compiled functions that use the macros (or whatever); even if you change the definition of the macro, the existing compiled code will continue to use the old definition. Thus, if the implementation of a module is changed programs that use it may need to be recompiled. This is something we sometimes accept for the sake of efficiency.

In the present implementation of flavors, which is discussed below, there is no such compiler incorporation of nonmodular knowledge into a program, except when the "outside-accessible instance variables" feature is used. See the section "**defflavor** Options". This problem is explained further in that section. If you don't use the "outside-accessible instance variables" feature, you don't have to worry about this.

4. Generic Operations

Suppose we think about the rest of the program that uses the **ship** abstraction. It may want to deal with other objects that are like **ships** in that they are movable objects with mass, but unlike **ships** in other ways. A more advanced model of a ship might include the concept of the ship's engine power, the number of passengers on board, and its name. An object representing a meteor probably would not have any of these, but might have another attribute such as how much iron is in it.

However, all kinds of movable objects have positions, velocities, and masses, and the system will contain some programs that deal with these quantities in a uniform way, regardless of what kind of object the attributes apply to. For example, a piece of the system that calculates every object's orbit in space need not worry about the other, more peripheral attributes of various types of objects; it works the same way for all objects. Unfortunately, a program that tries to calculate the orbit of a ship will need to know the ship's attributes, and will have to call **ship-x-position** and **ship-y-velocity** and so on. The problem is that these functions won't work for meteors. There would have to be a second program to calculate orbits for meteors that would be exactly the same, except that where the first one calls **ship-x-position**, the second one would call **meteor-x-position**, and so on. This would be very bad; a great deal of code would have to exist in multiple copies, all of it would have to be maintained in parallel, and it would take up space for no good reason.

What is needed is an operation that can be performed on objects of several different types. For each type, it should do the thing appropriate for that type. Such operations are called *generic* operations. The classic example of generic operations is the arithmetic functions in most programming languages, including Zetalisp. The + (or **plus**) function will accept either fixnums or flonums, and perform either fixnum addition or flonum addition, whichever is appropriate, based on the data types of the objects being manipulated. In our example, we need a generic **x-position** operation that can be performed on either **ships**, **meteors**, or any other kind of mobile object represented in the system. This way, we can write a single program to calculate orbits. When it wants to know the *x* position of the object it is dealing with, it simply invokes the generic **x-position** operation on the object, and whatever type of object it has, the correct operation is performed, and the *x* position is returned.

A terminology for the use of such generic operations has emerged from the Smalltalk and Actor languages: performing a generic operation is called *sending a message.* The objects in the program are thought of as little people, who get sent messages and respond with answers. In the example above, the objects are sent **x-position** messages, to which they respond with their *x* position. This *message passing* is how generic operations are performed.

Sending a message is a way of invoking a function. Along with the *name* of the message, in general, some arguments are passed; when the object is done with the message, some values are returned. The sender of the message is simply calling a function with some arguments, and getting some values back. The interesting thing is that the caller did not specify the name of a procedure to call. Instead, it specified a message name and an object; that is, it said what operation to perform, and what object to perform it on. The function to invoke was found from this information.

When a message is sent to an object, a function therefore must be found to handle the message. The two data used to figure out which function to call are the *type* of the object, and the *name* of the message. The same set of functions are used for all instances of a given type, so the type is the only attribute of the object used to figure out which function to call. The rest of the message besides the name are data which are passed as arguments to the function, so the name is the only part of the message used to find the function. Such a function is called a *method*. For example, if we send an **x-position** message to an object of type **ship**, then the function we find is "the **ship** type's **x-position** method". A method is a function that handles a specific kind of message to a specific kind of object; this method handles messages named **x-position** to objects of type **ship**.

In our new terminology: the orbit-calculating program finds the x position of the object it is working on by sending that object a message named **x-position** (with no arguments). The returned value of the message is the x position of the object. If the object was of type **ship**, then the **ship** type's **x-position** method was invoked; if it was of type **meteor**, then the **meteor** type's **x-position** method was invoked. The orbit-calculating program just sends the message, and the right function is invoked based on the type of the object. We now have true generic functions, in the form of message passing: the same operation can mean different things depending on the type of the object.

5. Generic Operations in Lisp

How do we implement message passing in Lisp? By convention, objects that receive messages are always *functional* objects (that is, you can apply them to arguments), and a message is sent to an object by calling that object as a function, passing the name of the message as the first argument, and the arguments of the message as the rest of the arguments. Message names are represented by symbols; normally these symbols are in the keyword package since messages are a protocol for communication between different programs, which may reside in different packages. So if we have a variable **my-ship** whose value is an object of type **ship**, and we want to know its *x* position, we send it a message as follows:

```
(funcall my-ship ':x-position)
```

This form returns the *x* position as its returned value. To set the ship's *x* position to **3.0**, we send it a message like this:

```
(funcall my-ship ':set-x-position 3.0)
```

It should be stressed that no new features are added to Lisp for message sending; we simply define a convention on the way objects take arguments. The convention says that an object accepts messages by always interpreting its first argument as a message name. The object must consider this message name, find the function which is the method for that message name, and invoke that function.

This raises the question of how message receiving works. The object must somehow find the right method for the message it is sent. Furthermore, the object now has to be callable as a function; objects can't just be **defstructs** any more, since those aren't functions. But the structure defined by **defstruct** was doing something useful: it was holding the instance variables (the internal state) of the object. We need a function with internal state; that is, we need a coroutine.

Of the Zetalisp features presented so far, the most appropriate is the closure. See the section "Closures". A message-receiving object could be implemented as a closure over a set of instance variables. The function inside the closure would have a big **selectq** form to dispatch on its first argument.

While using closures does work, it has several serious problems. The main problem is that in order to add a new operation to a system, it is necessary to modify a lot of code; you have to find all the types that understand that operation, and add a new clause to the **selectq**. The problem with this is that you cannot textually separate the implementation of your new operation from the rest of the system; the methods must be interleaved with the other operations for the type. Adding a new operation should only require *adding* Lisp code; it should not require *modifying* Lisp code.

The conventional way of making generic operations is to have a procedure for each operation, which has a big **selectq** for all the types; this means you have to modify code to add a type. The way described above is to have a procedure for each type, which has a big **selectq** for all the operations; this means you have to modify code to add an operation. Neither of these has the desired property that extending the system should only require adding code, rather than modifying code.

Closures are also somewhat clumsy and crude. A far more streamlined, convenient, and powerful system for creating message-receiving objects exists; it is called the *Flavor* mechanism. With flavors, you can add a new method simply by adding code, without modifying anything. Furthermore, many common and useful things to do are very easy to do with flavors. The rest of this chapter describes flavors.

6. Simple Use of Flavors

A *flavor*, in its simplest form, is a definition of an abstract type. New flavors are created with the **defflavor** special form, and methods of the flavor are created with the **defmethod** special form. New instances of a flavor are created with the **make-instance** function. This section explains simple uses of these forms.

For an example of a simple use of flavors, here is how the **ship** example above would be implemented.

```
(defflavor ship (x-position y-position
                 x-velocity y-velocity mass)
                ()
  :gettable-instance-variables)

(defmethod (ship :speed) ()
  (sqrt (+ (^ x-velocity 2)
           (^ y-velocity 2))))

(defmethod (ship :direction) ()
  (atan y-velocity x-velocity))
```

The code above creates a new flavor. The first subform of the **defflavor** is **ship**, which is the name of the new flavor. Next is the list of instance variables; they are the five that should be familiar by now. The next subform is something we will get to later. The rest of the subforms are the body of the **defflavor**, and each one specifies an option about this flavor. In our example, there is only one option, namely **:gettable-instance-variables**. This means that for each instance variable, a method should automatically be generated to return the value of that instance variable. The name of the message is a symbol with the same name as the instance variable, but interned on the keyword package. Thus, methods are created to handle the messages **:x-position**, **:y-position**, and so on.

Each of the two **defmethod** forms adds a method to the flavor. The first one adds a handler to the flavor **ship** for messages named **:speed**. The second subform is the lambda-list, and the rest is the body of the function that handles the **:speed** message. The body can refer to or set any instance variables of the flavor, the same as it can with local variables or special variables. When any instance of the **ship** flavor is invoked with a first argument of **:direction**, the body of the second **defmethod** will be evaluated in an environment in which the instance variables of **ship** refer to the instance variables of this instance (the one to which the message was sent). So when the arguments of **atan** are evaluated, the values of instance variables of the object to which the message was sent will be used as the arguments. **atan** will be invoked, and the result it returns will be returned by the instance itself.

Now we have seen how to create a new abstract type: a new flavor. Every instance of this flavor will have the five instance variables named in the **defflavor** form, and the seven methods we have seen (five that were automatically generated because of the **:gettable-instance-variables** option, and two that we wrote ourselves). The way to create an instance of our new flavor is with the **make-instance** function. Here is how it could be used:

```
(setq my-ship (make-instance 'ship))
```

This will return an object whose printed representation is:

```
#<SHIP 13731210>
```

(Of course, the value of the magic number will vary; it is not interesting anyway.) The argument to **make-instance** is, as you can see, the name of the flavor to be instantiated. Additional arguments, not used here, are *init options*, that is, commands to the flavor of which we are making an instance, selecting optional features. This will be discussed more in a moment.

Examination of the flavor we have defined shows that it is quite useless as it stands, since there is no way to set any of the parameters. We can fix this up easily, by putting the **:settable-instance-variables** option into the **defflavor** form. This option tells **defflavor** to generate methods for messages named **:set-x-position**, **:set-y-position**, and so on; each such method takes one argument, and sets the corresponding instance variable to the given value.

Another option we can add to the **defflavor** is **:initable-instance-variables**, to allow us to initialize the values of the instance variables when an instance is first created. **:initable-instance-variables** does not create any methods; instead, it makes *initialization keywords* named **:x-position**, **:y-position**, and so on, that can be used as init-option arguments to **make-instance** to initialize the corresponding instance variables. The set of init options are sometimes called the *init-plist* because they are like a property list.

Here is the improved **defflavor**:

```
(defflavor ship (x-position y-position
                 x-velocity y-velocity mass)
                ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

All we have to do is evaluate this new **defflavor**, and the existing flavor definition will be updated and now include the new methods and initialization options. In fact, the instance we generated a while ago will now be able to accept these new messages! We can set the mass of the ship we created by evaluating

```
(funcall my-ship ':set-mass 3.0)
```

and the **mass** instance variable of **my-ship** will properly get set to **3.0**.

In order to improve the clarity of heavily object-oriented programs, we do not use **funcall** to send messages. Instead, we use the **send** function, which has a shorter, more euphonious, and more specific name.

In fact, **send** and **funcall** are currently identical in their effects. Use of **send** is purely a matter of style. In a future release, it will be possible to send messages to objects of any data type. **send** and **funcall** will be identical when sending to an instance, but will be different when sending to objects of some other data types. Programs that use **send** to send messages and **funcall** to call functions will continue to work when this change is made.

send *object message-name* &rest *arguments* *Function*

Sends the message named *message-name* to the *object*. *arguments* are the arguments passed.

lexpr-send *object message-name* &rest *arguments* *Function*

Sends the message named *message-name* to the *object*. *arguments* are the arguments passed, except that the last element of *arguments* should be a list, and all the elements of that list are passed as arguments. Example:

```
(send some-window ':set-edges 10 10 40 40)
```

does the same thing as

```
(setq new-edges '(10 10 40 40))
(lexpr-send some-window ':set-edges new-edges)
```

If you want to experiment with flavors, it is useful to know that **describe** of an instance tells you the flavor of the instance and the values of its instance variables. If we were to evaluate **(describe my-ship)** at this point, the following would be printed:

```
#<SHIP 13731210>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:         unbound
        Y-POSITION:         unbound
        X-VELOCITY:         unbound
        Y-VELOCITY:         unbound
        MASS:               3.0
```

Now that the instance variables are "initable", we can create another ship and initialize some of the instance variables using the init-plist. Let's do that and **describe** the result:

```
(setq her-ship (make-instance 'ship ':x-position 0.0
                                     ':y-position 2.0
                                     ':mass 3.5))
              ==> #<SHIP 13756521>


(describe her-ship)
#<SHIP 13756521>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:        0.0
        Y-POSITION:        2.0
        X-VELOCITY:        unbound
        Y-VELOCITY:        unbound
        MASS:              3.5
```

A flavor can also establish default initial values for instance variables. These default values are used when a new instance is created if the values are not initialized any other way. The syntax for specifying a default initial value is to replace the name of the instance variable by a list, whose first element is the name and whose second is a form to evaluate to produce the default initial value. For example:

```
(defvar *default-x-velocity* 2.0)
(defvar *default-y-velocity* 3.0)

(defflavor ship ((x-position 0.0)
                 (y-position 0.0)
                 (x-velocity *default-x-velocity*)
                 (y-velocity *default-y-velocity*)
                 mass)
                ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)

(setq another-ship (make-instance 'ship ':x-position 3.4))

(describe another-ship)
#<SHIP 14563643>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:        3.4
        Y-POSITION:        0.0
        X-VELOCITY:        2.0
        Y-VELOCITY:        3.0
        MASS:              unbound
```

x-position was initialized explicitly, so the default was ignored. **y-position** was initialized from the default value, which was **0.0**. The two velocity instance variables were initialized from their default values, which came from two global variables. **mass** was not explicitly initialized and did not have a default initialization, so it was left unbound.

There are many other options that can be used in **defflavor,** and the init options can be used more flexibly than just to initialize instance variables; full details are given later in this chapter. But even with the small set of features we have seen so far, it is easy to write object-oriented programs.

7. Mixing Flavors

Now we have a system for defining message-receiving objects so that we can have generic operations. If we want to create a new type called **meteor** that would accept the same generic operations as **ship**, we could simply write another **defflavor** and two more **defmethods** that looked just like those of **ship**, and then meteors and ships would both accept the same operations. **ship** would have some more instance variables for holding attributes specific to ships, and some more methods for operations that are not generic, but are only defined for ships; the same would be true of **meteor**.

However, this would be a wasteful thing to do. The same code has to be repeated in several places, and several instance variables have to be repeated. The code now needs to be maintained in many places, which is always undesirable. The power of flavors (and the name "flavors") comes from the ability to mix several flavors and get a new flavor. Since the functionality of **ship** and **meteor** partially overlap, we can take the common functionality and move it into its own flavor, which might be called **moving-object**. We would define **moving-object** the same way as we defined **ship** in the previous section. Then, **ship** and **meteor** could be defined like this:

```
(defflavor ship (engine-power number-of-passengers name)
                (moving-object)
   :gettable-instance-variables)

(defflavor meteor (percent-iron) (moving-object)
   :initable-instance-variables)
```

These **defflavor** forms use the second subform, which we ignored previously. The second subform is a list of flavors to be combined to form the new flavor; such flavors are called *components*. Concentrating on **ship** for a moment (analogous things are true of **meteor**), we see that it has exactly one component flavor: **moving-object**. It also has a list of instance variables, which includes only the ship-specific instance variables and not the ones that it shares with **meteor**. By incorporating **moving-object**, the **ship** flavor acquires all of its instance variables, and so need not name them again. It also acquires all of **moving-object**'s methods, too. So with the new definition, **ship** instances will still accept the **:x-velocity** and **:speed** messages, and they will do the same thing. However, the **:engine-power** message will also be understood (and will return the value of the **engine-power** instance variable).

What we have done here is to take an abstract type, **moving-object**, and build two more specialized and powerful abstract types on top of it. Any ship or meteor can do anything a moving object can do, and each also has its own specific abilities. This kind of building can continue; we could define a flavor called

ship-with-passenger that was built on top of **ship**, and it would inherit all of **moving-object**'s instance variables and methods as well as **ship**'s instance variables and methods. Furthermore, the second subform of **defflavor** can be a list of several components, meaning that the new flavor should combine all the instance variables and methods of all the flavors in the list, as well as the ones *those* flavors are built on, and so on. All the components taken together form a big tree of flavors. A flavor is built from its components, its components' components, and so on. We sometimes use the term "components" to mean the immediate components (the ones listed in the **defflavor**), and sometimes to mean all the components (including the components of the immediate components and so on). (Actually, it is not strictly a tree, since some flavors might be components through more than one path. It is really a directed graph; it can even be cyclic.)

The order in which the components are combined to form a flavor is important. The tree of flavors is turned into an ordered list by performing a *top-down, depth-first* walk of the tree, including nonterminal nodes *before* the subtrees they head, and eliminating duplicates. For example, if **flavor-1**'s immediate components are **flavor-2** and **flavor-3**, and **flavor-2**'s components are **flavor-4** and **flavor-5**, and **flavor-3**'s component was **flavor-4**, then the complete list of components of **flavor-1** would be:

```
    flavor-1, flavor-2, flavor-4, flavor-5, flavor-3
```

The flavors earlier in this list are the more specific, less basic ones; in our example, **ship-with-passengers** would be first in the list, followed by **ship**, followed by **moving-object**. A flavor is always the first in the list of its own components. Notice that **flavor-4** does not appear twice in this list. Only the first occurrence of a flavor appears; duplicates are removed. (The elimination of duplicates is done during the walk; if there is a cycle in the directed graph, it will not cause a nonterminating computation.)

The set of instance variables for the new flavor is the union of all the sets of instance variables in all the component flavors. If both **flavor-2** and **flavor-3** have instance variables named **foo**, then **flavor-1** will have an instance variable named **foo**, and any methods that refer to **foo** will refer to this same instance variable. Thus different components of a flavor can communicate with one another using shared instance variables. (Typically, only one component ever sets the variable, and the others only look at it.) The default initial value for an instance variable comes from the first component flavor to specify one.

The way the methods of the components are combined is the heart of the flavor system. When a flavor is defined, a single function, called a *combined method*, is constructed for each message supported by the flavor. This function is constructed out of all the methods for that message from all the components of the flavor. There are many different ways that methods can be combined; these can be selected by the user when a flavor is defined. The user can also create new forms of combination.

There are several kinds of methods, but so far, the only kinds of methods we have seen are *primary* methods. The default way primary methods are combined is that all but the earliest one provided are ignored. In other words, the combined method is simply the primary method of the first flavor to provide a primary method. What this means is that if you are starting with a flavor **foo** and building a flavor **bar** on top of it, then you can override **foo**'s method for a message by providing your own method. Your method will be called, and **foo**'s will never be called.

Simple overriding is often useful; if you want to make a new flavor **bar** that is just like **foo** except that it reacts completely differently to a few messages, then this will work. However, often you don't want to completely override the base flavor's (**foo**'s) method; sometimes you want to add some extra things to be done. This is where combination of methods is used.

The usual way methods are combined is that one flavor provides a primary method, and other flavors provide *daemon methods*. The idea is that the primary method is "in charge" of the main business of handling the message, but other flavors just want to keep informed that the message was sent, or just want to do the part of the operation associated with their own area of responsibility.

When methods are combined, a single primary method is found; it comes from the first component flavor that has one. Any primary methods belonging to later component flavors are ignored. This is just what we saw above; **bar** could override **foo**'s primary method by providing its own primary method.

However, you can define other kinds of methods. In particular, you can define *daemon* methods. They come in two kinds, *before* and *after*. There is a special syntax in **defmethod** for defining such methods. Here is an example of the syntax. To give the **ship** flavor an after-daemon method for the **:speed** message, the following syntax would be used:

```
(defmethod (ship :after :speed) ()
   body)
```

Now, when a message is sent, it is handled by a new function called the *combined* method. The combined method first calls all of the before daemons, then the primary method, then all the after daemons. Each method is passed the same arguments that the combined method was given. The returned values from the combined method are the values returned by the primary method; any values returned from the daemons are ignored. Before-daemons are called in the order that flavors are combined, while after-daemons are called in the reverse order. In other words, if you build **bar** on top of **foo**, then **bar**'s before-daemons will run before any of those in **foo**, and **bar**'s after-daemons will run after any of those in **foo**.

The reason for this order is to keep the modularity order correct. If we create **flavor-1** built on **flavor-2**; then it should not matter what **flavor-2** is built out of. Our new before-daemons go before all methods of **flavor-2**, and our new after-daemons go after all methods of **flavor-2**. Note that if you have no daemons, this reduces to the form of combination described above. The most recently added component flavor is the highest level of abstraction; you build a higher-level object on top of a lower-level object by adding new components to the front. The syntax for defining daemon methods can be found in the description of **defmethod** below.

To clarify this, let's consider a simple example: the **:print-self** method. The Lisp printer (that is, the **print** function) prints instances of flavors by sending them **:print-self** messages. The first argument to the **:print-self** message is a stream (we can ignore the others for now), and the receiver of the message is supposed to print its printed representation on the stream. In the **ship** example above, the reason that instances of the **ship** flavor printed the way they did is because the **ship** flavor was actually built on top of a very basic flavor called **vanilla-flavor**; this component is provided automatically by **defflavor**. It was **vanilla-flavor**'s **:print-self** method that was doing the printing. Now, if we give **ship** its own primary method for the **:print-self** message, then that method will take over the job of printing completely; **vanilla-flavor**'s method will not be called at all. However, if we give **ship** a before-daemon method for the **:print-self** message, then it will get invoked before the **vanilla-flavor** message, and so whatever it prints will appear before what **vanilla-flavor** prints. So we can use before-daemons to add prefixes to a printed representation; similarly, after-daemons can add suffixes.

There are other ways to combine methods besides daemons, but this way is the most common. The more advanced ways of combining methods are explained in a later section. See the section "Method Combination". The **vanilla-flavor** and what it does for you are also explained later. See the section "Vanilla Flavor".

8. Flavor Functions

defflavor *Macro*

A flavor is defined by a form

```
(defflavor flavor-name (var1 var2...) (flav1 flav2...)
        opt1 opt2...)
```

flavor-name is a symbol which serves to name this flavor. It will get an **si:flavor** property of the internal data-structure containing the details of the flavor.

(typep *obj***)**, where *obj* is an instance of the flavor named *flavor-name*, will return the symbol *flavor-name*. **(typep** *obj flavor-name***)** is **t** if *obj* is an instance of a flavor, one of whose components (possibly itself) is *flavor-name*.

var1, *var2*, and so on, are the names of the instance-variables containing the local state for this flavor. A list of the name of an instance-variable and a default initialization form is also acceptable; the initialization form will be evaluated when an instance of the flavor is created if no other initial value for the variable is obtained. If no initialization is specified, the variable will remain unbound.

flav1, *flav2*, and so on, are the names of the component flavors out of which this flavor is built. The features of those flavors are inherited as described previously.

opt1, *opt2*, and so on, are options; each option may be either a keyword symbol or a list of a keyword symbol and arguments. The options to **defflavor** are described in another section. See the section "**defflavor** Options".

all-flavor-names *Variable*

This is a list of the names of all the flavors that have ever been **defflavor**'ed.

defmethod *Macro*

A method, that is, a function to handle a particular message sent to an instance of a particular flavor, is defined by a form such as

```
(defmethod (flavor-name method-type message) lambda-list
  form1 form2...)
```

flavor-name is a symbol which is the name of the flavor which is to receive the method. *method-type* is a keyword symbol for the type of method; it is omitted when you are defining a primary method, which is the usual case. *message* is a keyword symbol which names the message to be handled.

The meaning of the *method-type* depends on what kind of method combination is declared for this message. For instance, for daemons **:before** and **:after** are allowed. See the section "Method Combination". That section contains a complete description of method types and the way methods are combined.

lambda-list describes the arguments and "aux variables" of the function; the first argument to the method, which is the message keyword, is automatically handled, and so it is not included in the *lambda-list*. Note that methods may not have **&quote** arguments; that is they must be functions, not special forms. *form1*, *form2*, and so on, are the function body; the value of the last form is returned.

The variant form

```
(defmethod (flavor-name message) function)
```

where *function* is a symbol, says that *flavor-name*'s method for *message* is *function*, a symbol which names a function. That function must take appropriate arguments on the LM-2; the first argument is the message keyword. On the 3600, the first three arguments are the object receiving the message, the mapping table (which can safely be ignored), and the message keyword.

If you redefine a method that is already defined, the old definition is replaced by the new one. Given a flavor, a message name, and a method type, there can only be one function, so if you define a **:before** daemon method for the **foo** flavor to handle the **:bar** message, then you replace the previous before-daemon; however, you do not affect the primary method or methods of any other type, message name, or flavor.

The function spec for a method looks like:

```
(:method flavor-name message) or
(:method flavor-name method-type message)
```

This is useful to know if you want to trace or advise a method, or if you want to poke around at the method function itself, for example, disassemble it.

make-instance *flavor-name init-option1 value1 init-option2 value2...* *Function*

Creates and returns an instance of the specified flavor. Arguments after the first are alternating init-option keywords and arguments to those keywords. These options are used to initialize instance variables and to select arbitrary options, as described above. If the flavor supports the **:init** message, it is sent to the newly created object with one argument, the init-plist. This is a disembodied property-list containing the init-options specified and those defaulted from the flavor's **:default-init-plist**. **make-instance** is an easy-to-call interface to **instantiate-flavor**; for full details refer to that function.

instantiate-flavor *flavor-name init-plist* &optional *send-init-message-p return-unhandled-keywords area* *Function*

This is an extended version of **make-instance**, giving you more features. Note that it takes the init-plist as an argument, rather than taking an **&rest** argument of init-options and values.

The *init-plist* argument must be a disembodied property list; **locf** of an **&rest** argument will do. Beware! This property list can be modified; the properties from the default-init-plist that do not simply initialize instance variables are **putprop**'ed on if not already present, and some **:init** methods do explicit **putprops** onto the init-plist.

In the event that **:init** methods do **remprop** of properties already on the init-plist (as opposed to simply doing **get** and **putprop**), then the init-plist will get **rplacd**'ed. This means that the actual list of options will be modified. It also means that **locf** of an **&rest** argument will not work; the caller of **instantiate-flavor** must copy its rest argument (for example, with **copylist**); this is because **rplacd** is not allowed on **&rest** arguments.

First, if the flavor's method-table and other internal information have not been computed or are not up to date, they are computed. This may take a substantial amount of time and invoke the compiler, but will only happen once for a particular flavor no matter how many instances you make, unless you change something.

Next, the instance variables are initialized. There are several ways this initialization can happen. If an instance variable is declared initable, and a keyword with the same spelling as its name appears in *init-plist*, it is set to the value specified after that keyword. If an instance variable does not get initialized this way, and an initialization form was specified for it in a **defflavor**, that form is evaluated and the variable is set to the result. The initialization form may not depend on any instance variables nor on **self**; it will not be evaluated in the "inside" environment in which methods are called. If an instance variable does not get initialized either of these ways it will be left unbound; presumably an **:init** method should initialize it.

Note that a simple empty disembodied property list is **(nil)**, which is what you should give if you want an empty init-plist. If you use **nil**, the property list of **nil** will be used, which is probably not what you want.

If any keyword appears in the *init-plist* but is not used to initialize an instance variable and is not declared in an **:init-keywords** option it is presumed to be a misspelling. See the section "**defflavor** Options". So any keywords that you handle in an **:init** handler should also be mentioned in the **:init-keywords** option of the definition of the flavor. If the *return-unhandled-keywords* argument is not supplied, such keywords are complained about by signalling an error. But if *return-unhandled-keywords* is supplied non-**nil**, a list of such keywords is returned as the second value of **instantiate-flavor**.

Note that default values in the *init-plist* can come from the **:default-init-plist** option to **defflavor**. See the section "**defflavor** Options".

If the *send-init-message-p* argument is supplied and non-**nil**, an **:init** message is sent to the newly created instance, with one argument, the *init-plist*. **get** can be used to extract options from this property-list. Each flavor that needs initialization can contribute an **:init** method, by defining a daemon.

If the *area* argument is specified, it is the number of an area in which to cons the instance; otherwise it is consed in the default area.

defwrapper *Macro*

This is hairy and if you don't understand it you should skip it.

Sometimes the way the flavor system combines the methods of different flavors (the daemon system) is not powerful enough. In that case **defwrapper** can be used to define a macro which expands into code which is wrapped around the invocation of the methods. This is best explained by an example; suppose you needed a lock locked during the processing of the **:foo** message to the **bar** flavor, which takes two arguments, and you have a **lock-frobboz** special-form which knows how to lock the lock (presumably it generates an **unwind-protect**). **lock-frobboz** needs to see the first argument to the message; perhaps that tells it what sort of operation is going to be performed (read or write).

```
(defwrapper (bar :foo) ((arg1 arg2) . body)
  `(lock-frobboz (self arg1)
     ,body))
```

The use of the **body** macro-argument prevents the **defwrapper**'ed macro from knowing the exact implementation and allows several **defwrappers** from different flavors to be combined properly.

Note well that the argument variables, **arg1** and **arg2**, are not referenced with commas before them. These may look like **defmacro** "argument" variables, but they are not. Those variables are not bound at the time the **defwrapper**-defined macro is expanded and the back-quoting is done; rather the result of that macro-expansion and back-quoting is code which, when a message is sent, will bind those variables to the arguments in the message as local variables of the combined method.

Consider another example. Suppose you thought you wanted a **:before** daemon, but found that if the argument was **nil** you needed to return from processing the message immediately, without executing the primary method. You could write a wrapper such as

```
(defwrapper (bar :foo) ((arg1) . body)
  `(cond ((null arg1))          ;Do nothing if arg1 is nil
         (t before-code
            . ,body)))
```

Suppose you need a variable for communication among the daemons for a particular message; perhaps the **:after** daemons need to know what the primary method did, and it is something that cannot be easily deduced from just the arguments. You might use an instance variable for this, or you might create a special variable which is bound during the processing of the message and used free by the methods.

```
(defvar *communication*)
(defwrapper (bar :foo) (ignore . body)
  `(let ((*communication* nil))
     . ,body))
```

Similarly you might want a wrapper which puts a ***catch** around the processing of a message so that any one of the methods could throw out in the event of an unexpected condition.

Redefining a wrapper automatically performs the necessary recompilation of the combined method of the flavor. If a wrapper is given a new definition, the combined method is recompiled so that it gets the new definition. If a wrapper is redefined with the same old definition, the existing combined methods will keep being used, since they are still correct.

Like daemon methods, wrappers work in outside-in order; when you add a **defwrapper** to a flavor built on other flavors, the new wrapper is placed outside any wrappers of the component flavors. However, *all* wrappers happen before *any* daemons happen. When the combined method is built, the calls to the before-daemon methods, primary methods, and after-daemon methods are all placed together, and then the wrappers are wrapped around them. Thus, if a component flavor defines a wrapper, methods added by new flavors will execute within that wrapper's context.

8.1 Facility for Handling Messages to Flavor Objects

Whoppers are related to wrappers. A wrapper is a kind of macro that can be used to handle a message to an object of some flavor. Whoppers can do most of the things that wrappers can do, but have several advantages.

Both wrappers and whoppers are used in certain cases in which **:before** and **:after** daemons are not powerful enough. **:before** and **:after** daemons let you put some code before or after the execution of a method; wrappers and whoppers let you put some code *around* the execution of the method. For example, you might want to bind a special variable to some value around the execution of a method. You might also want to establish a condition handler or set up a ***catch**. Wrappers and whoppers can also decide whether or not the method should be executed.

The main difference between wrappers and whoppers is that a wrapper is like a macro, whereas a whopper is like a function. If you modify a wrapper, all of the combined methods that use that wrapper have to be recompiled; the system does this automatically, but it still takes time. If you modify a whopper, only the whopper has to be recompiled; the combined methods need not be changed. Another disadvantage of wrappers is that a wrapper's body is expanded in all of the combined methods in which it is involved, and if that body is very large and complex, all of that code is duplicated in many different compiled-code objects instead of being shared. Using whoppers is also somewhat easier than using wrappers. Whoppers are slightly slower than wrappers since they require two extra function calls each time a message is sent.

Whoppers are defined with the following special form:

defwhopper *(flavor-name message-name) arglist body...* *Special Form*

Defines a whopper for the specified message to the specified flavor. *arglist* is the list of arguments, which should be the same as the argument list for any method handling the specified message.

When a message is sent to an object of some flavor, and a whopper is defined for that message, the whopper runs before any of the methods (primary or daemon). The arguments are passed, and the body of the whopper is executed. If the whopper does not do anything special, the methods themselves are never run; the result of the whopper is returned as the result of sending the message. However, most whoppers usually run the methods for the message. To make this happen, the body of the whopper calls one of the following two functions:

continue-whopper &rest *arguments* *Function*

Calls the methods for the message that was intercepted by the whopper. *arguments* is the list of arguments passed to those methods. This function must be called from inside the body of a whopper. Normally the whopper passes down the same arguments that it was given. However, some whoppers might want to change the values of the arguments and pass new values; this is valid.

lexpr-continue-whopper &rest *arguments* *Function*

This is like **continue-whopper**, but the last element of *arguments* is a list of arguments to be passed. It is useful when the arguments to the intercepted message include an **&rest** argument.

The following whopper binds the value of the special variable **base** to 3 around the execution of the **:print-integer** message to flavor **foo** (this message takes one argument):

```
(defwhopper (foo :print-integer) (n)
  (let ((base 3))
    (continue-whopper n)))
```

The following whopper sets up a ***catch** around the execution of the **:compute-height** message to flavor **giant**, no matter what arguments this message uses:

```
(defwhopper (giant :compute-height) (&rest args)
  (*catch 'too-high
    (lexpr-continue-whopper args)))
```

Like daemon methods, whoppers work in outward-in order; when you add a **defwhopper** to a flavor built on other flavors, the new whopper is placed outside any whoppers of the component flavors. However, *all* whoppers happen before *any* daemons happen. Thus, if a component defines a whopper, methods added by new flavors are considered part of the continuation of that whopper and are called only when the whopper calls its continuation.

Whoppers and wrappers are considered equal for purposes of combination. If two flavors are combined, one having a wrapper and the other having a whopper for some method, then the wrapper or whopper of the flavor that is further out is on the outside. If, for some reason, the very same flavor has both a wrapper and a whopper for the same message, the wrapper goes outside the whopper.

Symbol \\
defun-method *function-spec flavor argument-list body...* *Special Form*

Sometimes you write a function which is not itself a method, but which is to be called by methods and wants to be able to access the instance variables of the object **self.** **defun-method** is like **defun**, but the function is able to access the instance variables of *flavor*. It is valid to call the function only while executing inside a method or a **defun-method** for an object of the specified flavor, or of some flavor built upon it.

function-spec must be a symbol.

defun-method works by defining two functions: one is a function named *function-spec*, and the other is a function named **(:defun-method** *function-spec***)**. An optimizer is also added to *function-spec* (since optimizers currently can be added only to symbols, *function-spec* is constrained to be a symbol for now). The function named *function-spec* can be called from anywhere, as long as **self** is bound to an appropriate instance. The environment is correctly set up, and the internal **:defun-method** is called. This requires calling into the Flavor system and has some performance penalty over sending a message. However, if *function-spec* is called from a context where the compiler can know the current flavor (in other words, some constraints on what **self** can be), the optimizer on *function-spec* turns into a call to the **:defun-method** internal function, generating inline code to pass the correct environment.

Also, because of the optimizer, **defun-method** acts like a subst in that better code is generated if the **defun-method** is defined in a file earlier than where it is used. However, **defun-methods** are not much faster than message-passing, even when the optimized version of the call is being used.

Note that it is faster to send a computed message than it is to call a computed function that is a **defun-method**! It is slower to use **funcall** to call the function being defined with a **defun-method** than it is to send a message in which you have to have a form that computes the name of the message at run time.

defselect-method *function-spec flavor body...* *Special Form*

This special form is like **defselect**, but the forms of the *body* are able to access the instance variables of *flavor*. See **defun-method**.

undefflavor *flavor-name* *Function*

Removes the flavor named by *flavor-name*.

undefmethod *(flavor [type] message)* *Macro*

```
(undefmethod (flavor :before :message))
```

removes the method created by

```
(defmethod (flavor :before :message) (args) ...)
```

To remove a wrapper, use **undefmethod** with **:wrapper** as the method type.

undefmethod is simply an interface to **fundefine**. **undefmethod** accepts the same syntax as **defmethod**.

undefun-method *function-spec* *Special Form*

undefun-method undoes the effect of **defun-method** in the same way that **undefmethod** undoes the effect of **defmethod**. This is a special form, not a function, so *function-spec* is not evaluated.

When you redefine a **defun-method** to no longer be a **defun-method**, you must use **undefun-method** for the **:defun-method** function generated internally by it. Otherwise the compiler will think that the function is still a **defun-method** and hence will generate the wrong code.

self *Variable*

When a message is sent to an object, the variable **self** is automatically bound to that object, for the benefit of methods which want to manipulate the object itself (as opposed to its instance variables).

recompile-flavor *flavor-name* &optional *single-message (use-old-combined-methods* **t***) (do-dependents* **t***)* *Function*

Updates the internal data of the flavor and any flavors that depend on it. If *single-message* is supplied non-**nil**, only the methods for that message are changed. The system does this when you define a new method that did not previously exist. If *use-old-combined-methods* is **t**, then the existing combined method functions will be used if possible. New ones will only be generated if the set of methods to be called has changed. This is the default. If *use-old-combined-methods* is **nil**, automatically generated functions to call multiple methods or to contain code generated by wrappers will be regenerated unconditionally. If *do-dependents* is **nil**, only the specific flavor you specified will be recompiled. Normally it and all flavors that depend on it will be recompiled.

recompile-flavor only affects flavors that have already been compiled. Typically this means it affects flavors that have been instantiated, but does not bother with mixins. See the section "Flavor Families".

compile-flavor-methods *flavor...* *Macro*

The form **(compile-flavor-methods** *flavor-name-1 flavor-name-2...***)**, placed in a file to be compiled, will cause the compiler to include the automatically generated combined methods for the named flavors in the resulting **bin** file, provided all of the necessary flavor definitions have been made. Furthermore, when the **bin** file is loaded, internal data structures (such as the list of all methods of a flavor) will get generated.

This means that the combined methods get compiled at compile time, and the data structures get generated at load time, rather than both things happening at run time. This is a very good thing to use, since the need to invoke the compiler at run-time makes programs that use flavors slow the first time they are run. (The compiler will still be called if incompatible changes have been made, such as addition or deletion of methods that must be called by a combined method.)

You should only use **compile-flavor-methods** for flavors that are going to be instantiated. For a flavor that will never be instantiated (that is, a flavor that only serves to be a component of other flavors that actually do get instantiated), it is a complete waste of time, except in the unusual case where those other flavors can all inherit the combined methods of this flavor instead of each one having its own copy of a combined method which happens to be identical to the others.

The **compile-flavor-methods** forms should be compiled after all of the information needed to create the combined methods is available. You should put these forms after all of the definitions of all relevant flavors, wrappers, and methods of all components of the flavors mentioned.

When a **compile-flavor-methods** form is seen by the interpreter, the combined methods are compiled and the internal data structures are generated.

get-handler-for *object message* — *Function*

Given an object and a message, will return that object's method for that message, or **nil** if it has none. When *object* is an instance of a flavor, this function can be useful to find which of that flavor's components supplies the method. If you get back a combined method, you can use the List Combined Methods Zmacs command to find out what it does. See the section "Useful Zmacs Commands".

This is related to the **:handler** function spec. See the section "Function Specs".

This function can be used with other things than flavors, and has an optional argument which is not relevant here and not documented.

flavor-allows-init-keyword-p *flavor-name keyword* — *Function*

Returns non-**nil** if the flavor named *flavor-name* allows *keyword* in the init options when it is instantiated, or **nil** if it does not. The non-**nil** value is the name of the component flavor which contributes the support of that keyword.

si:flavor-allowed-init-keywords *flavor-name* — *Function*

Returns a list of all symbols that are valid init-options for the flavor, sorted alphabetically. *flavor-name* should be the name of a flavor (a symbol). This function is primarily useful for people, rather than programs, to call to get information. You can use this to help remember the name of an init-option or to help write documentation about a particular flavor.

symeval-in-instance *instance symbol* &optional *no-error-p* — *Function*

This function is used to find the value of an instance variable inside a particular instance. *Instance* is the instance to be examined, and *symbol* is the instance variable whose value should be returned. If there is no such instance variable, an error is signalled, unless *no-error-p* is non-**nil** in which case **nil** is returned.

set-in-instance *instance symbol value* — *Function*

This function is used to alter the value of an instance variable inside a particular instance. *Instance* is the instance to be altered, *symbol* is the instance variable whose value should be set, and *value* is the new value. If there is no such instance variable, an error is signalled.

locate-in-instance *instance symbol* — *Function*

Returns a locative pointer to the cell inside *instance* which holds the value of the instance variable named *symbol*.

describe-flavor *flavor-name* — *Function*

This function prints out descriptive information about a flavor; it is self-explanatory. An important thing it tells you that can be hard to figure out yourself is the combined list of component flavors; this list is what is printed after the phrase "and directly or indirectly depends on".

si:*flavor-compilations* *Variable*

This variable contains a history of when the flavor mechanism invoked the compiler. It is a list; elements toward the front of the list represent more recent compilations. Elements are typically of the form

```
(:method flavor-name type message-name)
```

and *type* is typically **:combined**.

You may **setq** this variable to **nil** at any time; for instance before loading some files that you suspect may have missing or obsolete **compile-flavor-methods** in them.

si:*flavor-compile-trace* *Variable*

A string containing a textual description of each invocation of the compiler by the flavor system. New elements are appended to the end of the string (it has a fill pointer).

si:flavor-default-init-putprop *flavor value property* *Function*

si:flavor-default-init-putprop is just like **putprop** except that its first argument is either a flavor structure or the name of a flavor. It puts the property on the default init plist of the specified flavor.

si:flavor-default-init-get *flavor property* *Function*

si:flavor-default-init-get is just like **get** except that its first argument is either a flavor structure or the name of a flavor. It retrieves the property from the default init plist of the specified flavor. You can use **setf**:

```
(setf (si:flavor-default-init-get f p) x)
```

si:flavor-default-init-remprop *flavor property* *Function*

si:flavor-default-init-remprop is just like **remprop** except that its first argument is either a flavor structure or the name of a flavor. It removes the property from the default init plist of the specified flavor.

funcall-self *message arguments...* *Function*

When **self** is an instance or an entity, **(funcall-self** *args...***)** has the same effect as **(funcall self** *args...***)** except that it is a little faster since it doesn't have to re-establish the context in which the instance variables evaluate correctly. If **self** is not an instance (nor an "entity"), **funcall-self** and **funcall self** do the same thing.

When **self** is an instance, **funcall-self** will only work correctly if it is used in a method or a function, wrapped in a **declare-flavor-instance-variables**, that was called (not necessarily directly) from a method. Otherwise the instance-variables will not be already set up.

(funcall-self ...) is no longer used and is no longer any faster; use **(send self ...)** instead. See the function **send**.

lexpr-funcall-self *message arguments... list-of-arguments* *Function*

This function is a cross between **lexpr-funcall** and **funcall-self**. When **self** is an instance or an entity, **(lexpr-funcall-self** *args...***)** has the same effect as **(lexpr-funcall self** *args...***)** except that it is a little faster since it doesn't have to re-establish the context in which the instance variables evaluate correctly. If **self** is not an instance (nor an "entity"), **lexpr-funcall-self** and **lexpr-funcall** do the same thing.

(lexpr-funcall-self ...) is no longer used; use **(lexpr-send self ...)** instead. See the function **lexpr-send**.

declare-flavor-instance-variables *(flavor) body...* *Macro*

Sometimes you will write a function which is not itself a method, but which is to be called by methods and wants to be able to access the instance variables of the object **self**. The form

```
(declare-flavor-instance-variables (flavor-name)
  function-definition)
```

surrounds the *function-definition* with a declaration of the instance variables for the specified flavor, which will make them accessible by name. Currently this works by declaring them as special variables, but this implementation may be changed in the future. Note that it is only legal to call a function defined this way while executing inside a method for an object of the specified flavor, or of some flavor built upon it.
declare-flavor-instance-variables is obsolete. It exists only for compatibility with the old flavor system (pre-System 210).

9. defflavor Options

There are quite a few options to **defflavor**. They are all described here, although some are for very specialized purposes and not of interest to most users. Each option can be written in two forms; either the keyword by itself, or a list of the keyword and "arguments" to that keyword.

Several of these options declare things about instance variables. These options can be given with arguments which are instance variables, or without any arguments in which case they refer to all of the instance variables listed at the top of the **defflavor**. This is *not* necessarily all the instance variables of the component flavors; just the ones mentioned in this flavor's **defflavor**. When arguments are given, they must be instance variables that were listed at the top of the **defflavor**; otherwise they are assumed to be misspelled and an error is signalled. It is legal to declare things about instance variables inherited from a component flavor, but to do so you must list these instance variables explicitly in the instance variable list at the top of the **defflavor**.

:gettable-instance-variables
Enables automatic generation of methods for getting the values of instance variables. The message name is the name of the variable, in the keyword package (that is, put a colon in front of it.)

Note that there is nothing special about these methods; you could easily define them yourself. This option generates them automatically to save you the trouble of writing out a lot of very simple method definitions. (The same is true of methods defined by the **:settable-instance-variables** option.) If you define a method for the same message name as one of the automatically generated methods, the new definition will override the old one, just as if you had manually defined two methods for the same message name.

:settable-instance-variables
Enables automatic generation of methods for setting the values of instance variables. The message name is "**:set-**" followed by the name of the variable. All settable instance variables are also automatically made gettable and initable. (See the note in the description of the **:gettable-instance-variables** option.)

:initable-instance-variables
The instance variables listed as arguments, or all instance variables listed in this **defflavor** if the keyword is given alone, are made *initable*. This means that they can be initialized through use of a keyword (a colon followed by the name of the variable) as an init-option argument to **make-instance**.

:init-keywords
The arguments are declared to be keywords in the initialization property-list

which are processed by this flavor's **:init** methods. The system uses this for error-checking: before the system sends the **:init** message, it makes sure that all the keywords in the init-plist are either initable-instance-variables, or elements of this list. If the caller misspells a keyword or otherwise uses a keyword that no component flavor handles, **make-instance** signals an error. When you write a **:init** handler that accepts some keywords, they should be listed in the **:init-keywords** option of the flavor.

:default-init-plist
The arguments are alternating keywords and value forms, like a property-list. When the flavor is instantiated, these properties and values are put into the init-plist unless already present. This allows one component flavor to default an option to another component flavor. The value forms are only evaluated when and if they are used. For example,

```
(:default-init-plist :frob-array
                     (make-array 100))
```

would provide a default "frob array" for any instance for which the user did not provide one explicitly. **:default-init-plist** entries that initialize instance variables are not added to the init-plist seen by the **:init** methods.

:required-instance-variables
Declares that any flavor incorporating this one which is instantiated into an object must contain the specified instance variables. An error occurs if there is an attempt to instantiate a flavor that incorporates this one if it does not have these in its set of instance variables. Note that this option is not one of those which checks the spelling of its arguments in the way described at the start of this section (if it did, it would be useless).

Required instance variables may be freely accessed by methods just like normal instance variables. The difference between listing instance variables here and listing them at the front of the **defflavor** is that the latter declares that this flavor "owns" those variables and will take care of initializing them, while the former declares that this flavor depends on those variables but that some other flavor must be provided to manage them and whatever features they imply.

:required-init-keywords
The arguments are keywords. It is an error to try to make an instance of this flavor or any incorporating it without specifying these keywords as arguments to **make-instance** (or **instantiate-flavor**) or a **:default-init-plist** option in a component flavor. This error can often be detected at compile time.

:required-methods
The arguments are names of messages which any flavor incorporating this one must handle. An error occurs if there is an attempt to instantiate such a flavor and it is lacking a method for one of these messages. Typically this option appears in the **defflavor** for a base flavor. See the section "Flavor

Families". Usually this is used when a base flavor does a **(send self ...)** to send itself a message that is not handled by the base flavor itself; the idea is that the base flavor will not be instantiated alone, but only with other components (mixins) that do handle the message. This keyword allows the error of having no handler for the message be detected when the flavor is defined (which usually means at compile time) rather than at run time.

:required-flavors

The arguments are names of flavors which any flavor incorporating this one must include as components, directly or indirectly. The difference between declaring flavors as required and listing them directly as components at the top of the **defflavor** is that declaring flavors to be required does not make any commitments about where those flavors will appear in the ordered list of components; that is left up to whoever does specify them as components. The main thing that declaring a flavor as required accomplishes is to allow instance variables declared by that flavor to be accessed. It also provides error checking: an attempt to instantiate a flavor which does not include the required flavors as components will signal an error. Compare this with **:required-methods** and **:required-instance-variables**.

For an example of the use of required flavors, consider the **ship** example given earlier, and suppose we want to define a **relativity-mixin** which increases the mass dependent on the speed. We might write,

```
(defflavor relativity-mixin () (moving-object))
(defmethod (relativity-mixin :mass) ()
  (// mass (sqrt (- 1 (^ (// (send self ':speed)
                             *speed-of-light*)
                         2)))))
```

but this would lose because any flavor that had **relativity-mixin** as a component would get **moving-object** right after it in its component list. As a base flavor, **moving-object** should be last in the list of components so that other components mixed in can replace its methods and so that daemon methods combine in the right order. **relativity-mixin** has no business changing the order in which flavors are combined, which should be under the control of its caller, for example:

```
(defflavor starship ()
           (relativity-mixin long-distance-mixin ship))
```

which puts **moving-object** last (inheriting it from **ship**).

So instead of the definition above we write,

```
(defflavor relativity-mixin () ()
        (:required-flavors moving-object))
```

which allows **relativity-mixin**'s methods to access **moving-object** instance variables such as **mass** (the rest mass), but does not specify any place for **moving-object** in the list of components.

It is very common to specify the *base flavor* of a mixin with the **:required-flavors** option in this way.

:included-flavors

The arguments are names of flavors to be included in this flavor. The difference between declaring flavors here and declaring them at the top of the **defflavor** is that when component flavors are combined, if an included flavor is not specified as a normal component, it is inserted into the list of components immediately after the last component to include it. Thus included flavors act like defaults. The important thing is that if an included flavor *is* specified as a component, its position in the list of components is completely controlled by that specification, independently of where the flavor that includes it appears in the list.

:included-flavors and **:required-flavors** are used in similar ways; it would have been reasonable to use **:included-flavors** in the **relativity-mixin** example above. The difference is that when a flavor is required but not given as a normal component, an error is signalled, but when a flavor is included but not given as a normal component, it is automatically inserted into the list of components at a "reasonable" place.

:no-vanilla-flavor

Normally when a flavor is defined, the special flavor **si:vanilla-flavor** is included automatically at the end of its list of components. The vanilla flavor provides some default methods for the standard messages which all objects are supposed to understand. These include **:print-self**, **:describe**, **:which-operations**, and several other messages. See the section "Vanilla Flavor".

If any component of a flavor specifies the **:no-vanilla-flavor** option, then **si:vanilla-flavor** will not be included in that flavor. This option should not be used casually.

:mixture

Defines a family of related flavors. When **make-instance** (or **instantiate-flavor**) is called, it uses keywords in the init-plist to decide which flavor of the family to instantiate. Thus, init options can be used to select the flavor as well as instance-variable values.

The *ancestral* flavor is the one that includes the **:mixture** option in its **defflavor**. The flavors in the family are automatically constructed by mixing various mixins with the ancestral flavor. The names for the family members are chosen automatically. The name of such an automatically constructed flavor is a concatenation of the names of its components, separated by hyphens; however, obvious redundancies are removed heuristically.

defflavor of the ancestral flavor also defines the automatically constructed flavors. **compile-flavor-methods** of the ancestral flavor also compiles combined methods of the automatically constructed flavors.

The **:mixture** option has the following form:

(**:mixture** *spec spec* ...)

Each *spec* is processed independently, and all the resulting mixins are mixed together. A *spec* may be any of the following:

(*keyword mixin*)
: Add mixin if the value of *keyword* is **t**; add nothing if **nil**.

(*keyword* (*value mixin*) (*value mixin*) ...)
: Look up the value of *keyword* in this alist and add the specified mixin.

(*keyword mixin subspec subspec* ...)

(*keyword* (*value mixin subspec subspec* ...) ...)
: Subspecs take on the same forms as specs. Subspecs are processed only when the specified keyword has the specified value. Use them when there are interdependencies among keywords.

A *mixin* is one of the following:

symbol	The name of a flavor to be mixed in
nil	No flavor needs to be mixed in if the keyword takes on this value
string	This value is illegal: Signal an error with the string as the message

make-instance and **instantiate-flavor** check that the keywords are given with legal values.

Example:

```
(defflavor cereal-stream (...) (stream)
  ...
  (:init-keywords :characters :direction :ascii :hang-up-when-close)
  (:mixture (:characters
              (t nil (:direction (:in buffered-line-input-stream)
                                 (:out buffered-output-character-stream))
                     (:ascii ascii-translating-character-stream))
              (nil nil (:direction (:in buffered-input-stream)
                                   (:out buffered-output-stream))
                       (:ascii "Ascii translation is not meaningful
                                  for binary streams")))
            (:hang-up-when-close hang-up-when-close-mixin)))
```

Note the need for an **:init-keywords** declaration for any keywords that are used only in the **:mixture** declaration.

In this declaration, any kind of stream may have a **:hang-up-when-close** option. The **:characters** option does not itself add any mixins (hence the **nil**), but the processing of the **:direction** option depends on whether it is used with a character stream or a binary stream. The **:ascii** option is allowed only for character streams, and we specify an error message if it is used with a binary stream. If **:ascii** had not been mentioned in the **:characters nil** case, the keyword would have been ignored by **make-instance** on the assumption that an **:init** method was going to do something with it.

:default-handler

The argument is the name of a function which is to be called when a message is received for which there is no method. It will be called with whatever arguments the instance was called with, including the message name; whatever values it returns will be returned. If this option is not specified on any component flavor, it defaults to a function which will signal an error.

On the 3600, the function used with the **:default-handler** option to **defflavor** receives two additional arguments that it does not receive on the LM-2. The first argument is **self** and the second is always **nil**.

This is equivalent to using the **:unclaimed-message** message. See the section "Vanilla Flavor".

Example:

```
(defflavor lisp-stream (forward) ()
  (:default-handler lisp-stream-forward))

(defun lisp-stream-forward (#+3600 self #+3600 ignore message &rest arguments)
  (lexpr-funcall (send self ':forward) message arguments))
```

Note the use of #+3600 to indicate that the extra arguments apply only to the 3600.

:ordered-instance-variables

This option is mostly for esoteric internal system uses. The arguments are names of instance variables which must appear first (and in this order) in all instances of this flavor, or any flavor depending on this flavor. This is used for instance variables which are specially known about by microcode, and in connection with the **:outside-accessible-instance-variables** option. If the keyword is given alone, the arguments default to the list of instance variables given at the top of this **defflavor**.

:outside-accessible-instance-variables

The arguments are instance variables which are to be accessible from "outside" of this object, that is from functions other than methods. A macro (actually a **defsubst**) is defined which takes an object of this flavor as an argument and returns the value of the instance variable; **setf** may be used to set the value of the instance variable. The name of the macro is the name of the flavor concatenated with a hyphen and the name of the instance variable. These macros are similar to the accessor macros created by **defstruct**. See the document *Defstruct*.

This feature works in two different ways, depending on whether the instance variable has been declared to have a fixed slot in all instances, via the **:ordered-instance-variables** option.

If the variable is not ordered, the position of its value cell in the instance will have to be computed at run time. This takes noticeable time, more than actually sending a message would take. An error will be signalled if the argument to the accessor macro is not an instance or is an instance which does not have an instance variable with the appropriate name. However, there is no error check that the flavor of the instance is the flavor the accessor macro was defined for, or a flavor built upon that flavor. This error check would be too expensive.

If the variable is ordered, the compiler will compile a call to the accessor macro into a subprimitive which simply accesses that variable's assigned slot by number. This subprimitive is only 3 or 4 times slower than **car**. The only error-checking performed is to make sure that the argument is really an instance and is really big enough to contain that slot. There is no check that the accessed slot really belongs to an instance variable of the appropriate name. Any functions that use these accessor macros will have to be recompiled if the number or order of instance variables in the flavor is changed. The system will not know automatically to do this recompilation. If you aren't very careful, you may forget to recompile something, and have a very hard-to-find bug. Because of this problem, and because using these macros is less elegant than sending messages, the use of this option is discouraged. In any case the use of these accessor macros should be confined to the module which owns the flavor, and the "general public" should send messages.

:accessor-prefix

Normally the accessor macro created by the **:outside-accessible-instance-variables** option to access the flavor *f*'s instance variable *v* is named *f-v*. Specifying **(:accessor-prefix get$)** would cause it to be named **get$***v* instead.

:special-instance-variables

Use the **:special-instance-variables** option if you need instance variables to be bound as special variables when an instance is entered. Its format is like that of **:gettable-instance-variables**; that is, the option can be **:special-instance-variables** to declare all of the instance variables to be special variables, or it can be of the format **(:special-instance-variables a b c)** to declare only the instance variables **a**, **b**, and **c** to be special variables. When any method is called, these special variables are bound to the values in the instance, and references to these variables from methods are compiled as special variable references. This detracts from performance and should be avoided.

:method-order
The old name - **:select-method-order** - is still accepted, but it might go away in a future release. This is purely an efficiency hack. The arguments are names of messages that are frequently used or for which speed is important. Their combined methods are inserted into the handler hash table first, so that they are found by the first hash probe.

:method-combination
Declares the way that methods from different flavors will be combined. Each "argument" to this option is a list (*type order message1 message2...*). *Message1*, *message2*, and so on, are names of messages whose methods are to be combined in the declared fashion. *type* is a keyword which is a defined type of combination. See the section "Method Combination". *Order* is a keyword whose interpretation is up to *type*; typically it is either **:base-flavor-first** or **:base-flavor-last**.

Any component of a flavor may specify the type of method combination to be used for a particular message. If no component specifies a type of method combination, then the default type is used, namely **:daemon**. If more than one component of a flavor specifies it, then they must agree on the specification, or else an error is signalled.

:documentation
The list of arguments to this option is remembered on the flavor's property list as the **:documentation** property. The (loose) standard for what can be in this list is as follows; this may be extended in the future. A string is documentation on what the flavor is for; this may consist of a brief overview in the first line, then several paragraphs of detailed documentation. A symbol is one of the following keywords:

:mixin A flavor that you may want to mix with others to provide a useful feature.

:essential-mixin
A flavor that must be mixed in to all flavors of its class, or inappropriate behavior will ensue.

:lowlevel-mixin
A mixin used only to build other mixins.

:combination
A combination of flavors for a specific purpose.

:special-purpose
A flavor used for some internal or kludgey purpose by a particular program, which is not intended for general use.

This documentation can be viewed with the **describe-flavor** function or the Describe Flavor (m-X) Zmacs command.

:abstract-flavor

Declares that the flavor exists only to define a protocol; it is not intended to be instantiated by itself. Instead, it is intended to have more specialized flavors mixed in before being instantiated.

Trying to instantiate an abstract flavor signals an error.

:abstract-flavor is an advanced feature that affects paging. It decreases paging and usage of virtual memory by allowing abstract flavors to have combined methods. Normally, only instantiated flavors get combined methods, which are little Lisp functions that are automatically built and compiled by the flavor system to call all of the methods that are being combined to make the effective method. Sometimes many different instantiated flavors use the same combination of methods as each other. If this is the case, and the abstract flavor's combined methods are the same ones that are needed by the instantiated flavors, then all instantiated flavors can simply share the combined methods of the abstract flavor instead of having to each make their own. This sharing improves performance because it reduces the working set.

compile-flavor-methods is permitted on an abstract flavor. It is useful for combined methods that most specializations of that flavor would be able to share. This was one reason for this change. Without the **:abstract-flavor** declaration, **compile-flavor-methods** warned you about the flavor not being one that could be instantiated.

10. Flavor Families

The following organization conventions are recommended for all programs that use flavors.

A *base flavor* is a flavor that defines a whole family of related flavors, all of which will have that base flavor as one of their components. Typically the base flavor includes things relevant to the whole family, such as instance variables, **:required-methods** and **:required-instance-variables** declarations, default methods for certain messages, **:method-combination** declarations, and documentation on the general protocols and conventions of the family. Some base flavors are complete and can be instantiated, but most are not instantiatable and merely serve as a base upon which to build other flavors. The base flavor for the *foo* family is often named **basic-***foo*.

A *mixin flavor* is a flavor that defines one particular feature of an object. A mixin cannot be instantiated, because it is not a complete description. Each module or feature of a program is defined as a separate mixin; a usable flavor can be constructed by choosing the mixins for the desired characteristics and combining them, along with the appropriate base flavor. By organizing your flavors this way, you keep separate features in separate flavors, and you can pick and choose among them. Sometimes the order of combining mixins does not matter, but often it does, because the order of flavor combination controls the order in which daemons are invoked and wrappers are wrapped. Such order dependencies would be documented as part of the conventions of the appropriate family of flavors. A mixin flavor that provides the *mumble* feature is often named *mumble***-mixin**.

If you are writing a program that uses someone else's facility to do something, using that facility's flavors and methods, your program might still define its own flavors, in a simple way. The facility might provide a base flavor and a set of mixins, and the caller can combine these in various combinations depending on exactly what it wants, since the facility probably would not provide all possible useful combinations. Even if your private flavor has exactly the same components as a pre-existing flavor, it can still be useful since you can use its **:default-init-plist** to select options of its component flavors and you can define one or two methods to customize it "just a little".

11. Vanilla Flavor

The messages described in this section are a standard protocol which all message-receiving objects are assumed to understand. The standard methods that implement this protocol are automatically supplied by the flavor system unless the user specifically tells it not to do so. These methods are associated with the flavor **si:vanilla-flavor**:

si:vanilla-flavor *Flavor*

Unless you specify otherwise (with the **:no-vanilla-flavor** option to **defflavor**), every flavor includes the "vanilla" flavor, which has no instance variables but provides some basic useful methods.

:print-self *stream prindepth slashify-p* *Message*

The object should output its printed-representation to a stream. The printer sends this message when it encounters an instance. The arguments are the stream, the current depth in list-structure (for comparison with **prinlevel**), and whether slashification is enabled (**prin1** vs **princ**). See the section "What the Printer Produces". Vanilla-flavor ignores the last two arguments, and prints something like #<*flavor-name octal-address*>. The *flavor-name* tells you what type of object it is, and the *octal-address* allows you to tell different objects apart (provided the garbage collector doesn't move them behind your back).

:describe *Message*

The object should describe itself, printing a description onto the **standard-output** stream. The **describe** function sends this message when it encounters an instance or an entity. Vanilla-flavor outputs the object, the name of its flavor, and the names and values of its instance-variables, in a reasonable format.

:which-operations *Message*

The object should return a list of the messages it can handle. Vanilla-flavor generates the list once per flavor and remembers it, minimizing consing and compute-time. If a new method is added, the list is regenerated the next time someone asks for it.

:operation-handled-p *operation* *Message*

operation is a message name. The object should return **t** if it has a handler for the specified message, or **nil** if it does not.

:get-handler-for *operation* *Message*

operation is a message name. The object should return the method it uses to handle *operation*. If it has no handler for that message, it should return

nil. This is like the **get-handler-for** function, but you can only use it on objects known to accept messages.

:send-if-handles *operation* &rest *arguments* *Message*

operation is a message name and *arguments* is a list of arguments for that message. The object should send itself that message with those arguments, if it handles the message. If it doesn't handle the message it should just return **nil**.

:eval-inside-yourself *form* *Message*

The argument is a form which is evaluated in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. It works to **setq** one of these special variables; the instance variable will be modified. This is mainly intended to be used for debugging. An especially useful value of *form* is **(break t)**; this gets you a Lisp top level loop inside the environment of the methods of the flavor, allowing you to examine and alter instance variables, and run functions that use the instance variables.

:funcall-inside-yourself *function* &rest *args* *Message*

function is applied to *args* in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. It works to **setq** one of these special variables; the instance variable will be modified. This is mainly intended to be used for debugging.

:unclaimed-message *message* &rest *arguments* *Message*

For each message, the flavor system checks to be sure that a method exists for this message. When no method is found, it checks for a handler for **:unclaimed-message**. When such a handler exists, it is invoked with arguments *message* (the unclaimed message) and all of the arguments that were sent to the unclaimed message.

This is equivalent to using the **:default-handler** option to **defflavor**. See the section "**defflavor** Options".

12. Method Combination

As was mentioned earlier, there are many ways to combine methods. The way we have seen is called the **:daemon** type of combination. To use one of the others, you use the **:method-combination** option to **defflavor** to say that all the methods for a certain message to this flavor, or a flavor built on it, should be combined in a certain way.

The following types of method combination are supplied by the system. It is possible to define your own types of method combination; for information on this, see the code. Note that for most types of method combination other than **:daemon** you must define the order in which the methods are combined, either **:base-flavor-first** or **:base-flavor-last**, in the **:method-combination** option. In this context, base-flavor means the last element of the flavor's fully expanded list of components.

Which method type keywords are allowed depends on the type of method combination selected. There are also certain method types used for internal purposes.

:daemon	This is the default type of method combination. All the **:before** methods are called, then the primary (untyped) method for the outermost flavor that has one is called, then all the **:after** methods are called. The value returned is the value of the primary method.
:progn	All the methods are called, inside a **progn** special form. Methods can have a **:progn** type for documentation. This means that all of the methods are called, and the result of the combined method is whatever the last of the methods returns.
:or	All the methods are called, inside an **or** special form. Methods can have an **:or** type for documentation. This means that each of the methods is called in turn. If a method returns a non-**nil** value, that value is returned and none of the rest of the methods are called; otherwise, the next method is called. In other words, each method is given a chance to handle the message; if it doesn't want to handle the message, it should return **nil**, and the next method will get a chance to try.
:and	All the methods are called, inside an **and** special form. Methods can have an **:and** type for documentation. The basic idea is much like **:or**.
:list	Calls all the methods and returns a list of their returned values. Methods can have a **:list** type for documentation.
:inverse-list	Calls each method with one argument; these arguments are

successive elements of the list which is the sole argument to the message. Methods can have an **:inverse-list** type for documentation. Returns no particular value. If the result of a **:list**-combined message is sent back with an **:inverse-list**-combined message, with the same ordering and with corresponding method definitions, each component flavor receives the value which came from that flavor.

:pass-on Calls each method on the values returned by the preceding one. The values returned by the combined method are those of the outermost call. Methods can have a **:pass-on** type for documentation. The format of the declaration in the **defflavor** is:

```
(:method-combination (:pass-on (ordering . arglist)) . operation-names)
```

Where *ordering* is **:base-flavor-first** or **:base-flavor-last**. *arglist* may include the **&aux** and **&optional** keywords.

:append All the component methods are called as arguments to **append**. It expects each of the methods to return a list; the final result is the result of appending all these lists. Methods can have an **:append** type for documentation.

:nconc All the component methods are called as arguments to **nconc**. It expects each of the methods to return a list; the final result is the result of concatenating these lists. Methods can have an **:append** type for documentation.

:daemon-with-or This is like the **:daemon** method combination type, except the primary method is wrapped in an **or** special form with all **:or** methods. Multiple values will be returned from the primary method, but not the **:or** methods. This will produce combined methods like this (simplified to ignore multiple values):

```
(progn (foo-before-method)
       (or (foo-or-method)
           (foo-primary-method))
       (foo-after-method))
```

This is primarily useful for flavors in which a mixin introduces an alternative to the primary method. Each **:or** message gets a chance to run before the primary method and to decide whether the primary method should be run or not; if any **:or** method returns a non-**nil** value, the primary method is not run (nor are the rest of the **:or** methods). Note that the ordering of the combination of the **:or** methods is controlled by the *order* keyword in the **:method-combination** option to **defflavor**.

:daemon-with-and
This is like **:daemon-with-or** except combining **:and** methods in

an **and** special form. The primary method will only be run if all of the **:and** methods return non-**nil** values.

:daemon-with-override
This is like the **:daemon** method combination type, except an **or** special form is wrapped around the entire combined method with all **:override** typed methods before the combined method. This differs from **:daemon-with-or** in that the **:before** and **:after** daemons are not run unless *none* of the **:override** methods returns non-**nil**. The combined method looks something like this:

```
(or (foo-override-method)
    (progn (foo-before-method)
           (foo-primary-method)
           (foo-after-method)))
```

:case Takes a subsidiary message name. It dispatches on this message name just as the original message name caused a primary dispatch. This facility is used in the condition handling system. (See the document *Conditions*.)

```
(defmethod (sys:subscript-out-of-bounds :case :proceed :new-subscript)
           (&optional (sub (prompt-and-read ':number
                                "Subscript to use instead: ")))
  "Supply a different subscript"
  (values ':new-subscript sub))

(send obj ':proceed ':new-subscript new-sub)
```

Here is a table of all the method types used in the standard system (a user can add more, by defining new forms of method combination).

(no type) If no type is given to **defmethod**, a primary method is created. This is the most common type of method.

:before

:after These are used for the before-daemon and after-daemon methods used by **:daemon** method combination.

:override This allows some of the features of **:or** method combination to be used with daemons. An **:override** method can choose at run-time whether to act like a primary method or to act as if it was not there. In typical usages of this feature, the **:override** method usually returns **nil** and does nothing, but in exceptional circumstances it takes over the handling of the message. **:override** is used only with the **:daemon-with-override** method combination.

:default If there are no untyped methods among any of the flavors being combined, then the **:default** methods (if any) are treated as if they were untyped. If there are any untyped methods, the **:default** methods are ignored.

Typically a base-flavor defines some default methods for certain of the messages understood by its family. See the section "Flavor Families".

:or	
:and	These are used for **:daemon-with-or** and **:daemon-with-and** method combination.
:wrapper	This type is used internally by **defwrapper**.
:whopper	This type is used internally by **defwhopper**.
:combined	Used internally for automatically generated *combined* methods.

The most common form of combination is **:daemon**. One thing may not be clear: when do you use a **:before** daemon and when do you use an **:after** daemon? In some cases the primary method performs a clearly defined action and the choice is obvious: **:before :launch-rocket** puts in the fuel, and **:after :launch-rocket** turns on the radar tracking.

In other cases the choice can be less obvious. Consider the **:init** message, which is sent to a newly created object. To decide what kind of daemon to use, we observe the order in which daemon methods are called. First the **:before** daemon of the highest level of abstraction is called, then **:before** daemons of successively lower levels of abstraction are called, and finally the **:before** daemon (if any) of the base flavor is called. Then the primary method is called. After that, the **:after** daemon for the lowest level of abstraction is called, followed by the **:after** daemons at successively higher levels of abstraction.

Now, if there is no interaction among all these methods, if their actions are completely orthogonal, then it doesn't matter whether you use a **:before** daemon or an **:after** daemon. It makes a difference if there is some interaction. The interaction we are talking about is usually done through instance variables; in general, instance variables are how the methods of different component flavors communicate with each other. In the case of the **:init** message, the *init-plist* can be used as well. The important thing to remember is that no method knows beforehand which other flavors have been mixed in to form this flavor; a method cannot make any assumptions about how this flavor has been combined, and in what order the various components are mixed.

This means that when a **:before** daemon has run, it must assume that none of the methods for this message have run yet. But the **:after** daemon knows that the **:before** daemon for each of the other flavors has run. So if one flavor wants to convey information to the other, the first one should "transmit" the information in a **:before** daemon, and the second one should "receive" it in an **:after** daemon. So while the **:before** daemons are run, information is "transmitted"; that is, instance variables get set up. Then, when the **:after** daemons are run, they can look at the instance variables and act on their values.

In the case of the **:init** method, the **:before** daemons typically set up instance variables of the object based on the init-plist, while the **:after** daemons actually do things, relying on the fact that all of the instance variables have been initialized by the time they are called.

Of course, since flavors are not hierarchically organized, the notion of levels of abstraction is not strictly applicable. However, it remains a useful way of thinking about systems.

Combination Method Types

Methods used with **:progn**, **:append**, **:nconc**, **:and**, **:or**, **:list**, **:inverse-list**, and **:pass-on** combination types can use the combination type as the method type. This is useful in documenting how the method is used.

In the following example, **(:method foo :or :find-frabjous-frob)** could have been defined as **(:method foo :find-frabjous-frob)**. The only difference is one of style: Using **:or** as the method type makes it clear that the methods are combined using **:or** combination.

```
(defflavor foo (frobl) (bar)
  (:method-combination (:or :base-flavor-last :find-frabjous-frob)))

(defmethod (foo :or :find-frabjous-frob) (type)
  (dolist (frob frobl)
    (when (send frob ':frabjous-p type)
      (return frob))))
```

13. Copying Instances

Many people have asked "How do I copy an instance?" and have expressed surprise when told that the flavor system does not include any built-in way to copy instances. Why isn't there just a function **copy-instance** that creates a new instance of the same flavor with all its instance variables having the same values as in the original instance? This would work for the simplest use of flavors, but it isn't good enough for most advanced uses of flavors. A number of issues are raised by copying:

- Do you or do you not send an **:init** message to the new instance? If you do, what init-plist options do you supply?

- If the instance has a property list, you should copy the property list (e.g. with **copylist**) so that sending a **:putprop** or **:remprop** message to one of the instances does not affect the properties of the other instance.

- The instance may be contained in data structure maintained by the program of which it is a part. For example, a graphics system might have a list of all the objects that are currently visible on the screen. Copying such an instance requires making the appropriate entries in the data structure.

- If the instance is a pathname, the concept of copying is not even meaningful. Pathnames are *interned*, which means that there can only be one pathname object with any given set of instance-variable values.

- If the instance is a stream connected to a network, some of the instance variables represent an agent in another host elsewhere in the network. Copying the instance requires that a copy of that agent somehow be constructed.

- If the instance is a stream connected to a file, should copying the stream make a copy of the file or should it make another stream open to the same file? Should the choice depend on whether the file is open for input or for output?

In general, you can see that in order to copy an instance one must understand a lot about the instance. One must know what the instance variables mean so that the values of the instance variables can be copied if necessary. One must understand what relations to the external environment the instance has so that new relations can be established for the new instance. One must even understand what the general concept "copy" means in the context of this particular instance, and whether it means anything at all.

Copying is a generic operation, whose implementation for a particular instance depends on detailed knowledge relating to that instance. Modularity dictates that this knowledge be contained in the instance's flavor, not in a "general copying function". Thus the way to copy an instance is to send it a message.

The flavor system chooses not to provide any default method for copying an instance, and does not even suggest a standard name for the copying message, because copying involves so many semantic issues.

One way that people have organized copying of instances is to define a message, **:copy**, whose methods are combined with **:append** method combination. Each method supplies some init-plist options. Thus each component flavor controls the copying of its own aspect of the instance's behavior. The resulting appended list of init-plist options is used to create the new instance. Each component flavor has an **:init** method that extracts the init-plist options that are relevant to it and initializes the appropriate aspect of the new instance. A wrapper can be used to clean up the interface to the **:copy** message seen from the outside. A simple example follows:

```
(defflavor basic-copyable-object () ()
  (:method-combination (:append :base-flavor-last :copy)))

(defwrapper (basic-copyable-object :copy) (() . body)
  `(lexpr-funcall #'make-instance (typep self) (progn ,@body)))

(defflavor copyable-property-list-mixin () (si:property-list-mixin))

(defmethod (copyable-property-list-mixin :copy) ()
  `(:property-list ,(copylist (send self :property-list))))

(defflavor example () (copyable-property-list-mixin basic-copyable-object))

(setq a (make-instance 'example))

(send a :putprop 1 'value)

(setq b (send a :copy))

(send b ':get 'value)  =>  1

(send b :putprop 1.5 'value)

(send b :get 'value)  =>  1.5

(send a :get 'value)  =>  1
```

A related feature is the **:fasd-form** message which provides a way for an instance to tell the compiler how to copy it from one Lisp world into another, via a **bin** file. This is different from making a second copy of the instance in the same Lisp world. **:fasd-form** is a way to get an *equivalent* instance when the **bin** file is loaded.

14. Implementation of Flavors

An object which is an instance of a flavor is implemented using the data type **dtp-instance**. The representation is a structure whose first word, tagged with a header data type, points to a structure (known to the microcode as an "instance descriptor") containing the internal data for the flavor. The remaining words of the structure are value cells containing the values of the instance variables. The instance descriptor is a **defstruct** which appears on the **si:flavor** property of the flavor name. It contains, among other things, the name of the flavor, the size of an instance, the table of methods for handling messages, and information for accessing the instance variables.

defflavor creates such a data structure for each flavor, and links them together according to the dependency relationships between flavors.

A message is sent to an instance simply by calling it as a function, with the first argument being the message keyword. The instance descriptor contains a hash table that associates the message keyword to the actual function to be called. If there is only one method, this is that method, otherwise it is an automatically generated function, called the combined method, which calls the appropriate methods in the right order. See the section "Mixing Flavors". If there are wrappers, they are incorporated into this combined method. The function that handles the message is called with three special arguments preceding the arguments of the message. These are **self** (the object to which the message was sent), **self-mapping-table** (an internal data structure used in the accessing of instance variables), and the message keyword. (The LM-2 binds **self** and **self-mapping-table** as special variables rather than passing them as arguments.)

The function-specifier syntax
(:method *flavor-name optional-method-type message-name***)** is understood by **fdefine** and related functions.

14.1 Order of Definition

There is a certain amount of freedom to the order in which you do **defflavor**'s, **defmethod**'s, and **defwrapper**'s. This freedom is designed to make it easy to load programs containing complex flavor structures without having to do things in a certain order. It is considered important that not all the methods for a flavor need be defined in the same file. Thus the partitioning of a program into files can be along modular lines.

The rules for the order of definition are as follows.

Before a method can be defined (with **defmethod** or **defwrapper**) its flavor must have been defined (with **defflavor**). This makes sense because the system has to have a place to remember the method, and because it has to know the instance-variables of the flavor if the method is to be compiled.

When a flavor is defined (with **defflavor**) it is not necessary that all of its component flavors be defined already. This is to allow **defflavor**'s to be spread between files according to the modularity of a program, and to provide for mutually dependent flavors. Methods can be defined for a flavor some of whose component flavors are not yet defined; however, in certain cases compiling those methods will produce a warning that an instance variable was declared special (because the system did not realize it was an instance variable). Such a warning indicates that the compiled code will not work.

The methods automatically generated by the **:gettable-instance-variables** and **:settable-instance-variables** options to **defflavor** are generated at the time the **defflavor** is done.

The first time a flavor is instantiated, the system looks through all of the component flavors and gathers various information. At this point an error will be signalled if not all of the components have been **defflavor**'ed. This is also the time at which certain other errors are detected, for instance lack of a required instance-variable (see the **:required-instance-variables** option to **defflavor**). The combined methods are generated at this time also, unless they already exist. They will already exist if **compile-flavor-methods** was used, but if those methods are obsolete because of changes made to component flavors since the compilation, new combined methods will be made.

After a flavor has been instantiated, it is possible to make changes to it. These changes will affect all existing instances if possible. This is described more fully immediately below.

14.2 Changing a Flavor

You can change anything about a flavor at any time. You can change the flavor's general attributes by doing another **defflavor** with the same name. You can add or modify methods by doing **defmethod**'s. If you do a **defmethod** with the same flavor-name, message-name, and (optional) method-type as an existing method, that method is replaced with the new definition. You can remove a flavor with **undefflavor** and a method with **undefmethod**.

These changes will always propagate to all flavors that depend upon the changed flavor. Normally the system will propagate the changes to all existing instances of the changed flavor and all flavors that depend on it. However, this is not possible when the flavor has been changed so drastically that the old instances would not work properly with the new flavor. This happens if you change the number of instance variables, which changes the size of an instance. It also happens if you change the order of the instance variables (and hence the storage layout of an instance), or if you change the component flavors (which can change several subtle aspects of an instance). The system does not keep a list of all the instances of each flavor, so it cannot find the instances and modify them to conform to the new flavor definition. Instead it gives you a warning message, on the **error-output** stream, to the effect that the flavor was changed incompatibly and the old instances will not get the new version. The system leaves the old flavor data-structure intact (the old instances will continue to point at it) and makes a new one to contain the new version of the flavor. If a less drastic change is made, the system modifies the original flavor data-structure, thus affecting the old instances that point at it. However, if you redefine methods in such a way that they only work for the new version of the flavor, then trying to use those methods with the old instances won't work.

15. Entities

This section applies to the LM-2 only. An *entity* is a Lisp object; the entity is one of the primitive data types provided by the Lisp Machine system (the **data-type** function returns **dtp-entity** if it is given an entity). Entities are just like closures: they have all the same attributes and functionality. The only difference between the two primitive types is their data type: entities are clearly distinguished from closures because they have a different data type. The reason there is an important difference between them is that various parts of the (not so primitive) Lisp system treat them differently. See the section "Entities: Closures". The Lisp functions that deal with entities are discussed in that section.

A closure is simply a kind of function, but an entity is assumed to be a message-receiving object. Thus, when the Lisp printer is given a closure, it prints a simple textual representation, but when it is handed an entity, it sends the entity a **:print-self** message, which the entity is expected to handle. The **describe** function also sends entities messages when it is handed them. So when you want to make a message-receiving object out of a closure, you should use an entity instead. See the section "Generic Operations in Lisp".

Usually there is no point in using entities instead of flavors. Entities were introduced into Zetalisp before flavors were, and perhaps they would not have been had flavors already existed. Flavors have had considerably more attention paid to efficiency and to good tools for using them.

Entities are created with the **entity** function. See the section "Entities: Closures". The function part of an entity should usually be a function created by **defselect**.

16. Useful Zmacs Commands

This section briefly documents some Zmacs commands that are useful in conjunction with flavors.

m-. The m-. (Edit Definition) command can find the definition of a flavor in the same way that it can find the definition of a function.

Edit Definition can find the definition of a method if you give

`(:method` *flavor type message*`)`

as the function name. The keyword **:method** may be omitted. Completion will occur on the flavor name and message name as usual with Edit Definition.

Describe Flavor (m-X)
: Asks for a flavor name in the minibuffer and describes its characteristics. When typing the flavor name you have completion over the names of all defined flavors (thus this command can be used to aid in guessing the name of a flavor). The display produced is mouse sensitive where there are names of flavors and of methods; as usual the right-hand mouse button gives you a menu of operations and the left-hand mouse button does the most common operation, typically positioning the editor to the source code for the thing you are pointing at.

List Methods (m-X)

Edit Methods (m-X)
: Asks you for a message in the minibuffer and lists all the flavors which have a method for that message. You may type in the message name, point to it with the mouse, or let it default to the message which is being sent by the Lisp form the cursor is inside of. List Methods produces a mouse-sensitive display allowing you to edit selected methods or just see which flavors have methods, while Edit Methods skips the display and proceeds directly to editing the methods. As usual with this type of command, the Zmacs command **control-.** is redefined to advance the cursor to the next method in the list, reading in its source file if necessary. Pressing c-. while the display is on the screen edits the first method.

List Combined Methods (m-X)

Edit Combined Methods (m-X)
: Asks you for a message and a flavor in two minibuffers and lists all the methods which would be called if that message were sent to an instance of that flavor. You may point to the message and flavor with the mouse, and there is completion for the flavor name. As in List/Edit Methods, the display is mouse sensitive and the Edit version of the command skips the display and proceeds directly to the editing phase.

List Combined Methods can be very useful for telling what a flavor will do in response to a message. It shows you the primary method, the daemons, and the wrappers and lets you see the code for all of them; type **control-.** to get to successive ones.

17. Property List Messages

It is often useful to associate a property list with an abstract object, for the same reasons that it is useful to have a property list associated with a symbol. This section describes a mixin flavor that can be used as a component of any new flavor in order to provide that new flavor with a property list. See the section "Property Lists". That section contains a general discussion of property lists, plus more details and examples. [Currently, the functions **get, putprop**, and so on, do not accept flavor instances as arguments and send the corresponding message; this will be fixed.]

si:property-list-mixin *Flavor*
This mixin flavor provides the basic operations on property lists.

:get *indicator* of **si:property-list-mixin** *Method*
The **:get** message looks up the object's *indicator* property. If it finds such a property, it returns the value; otherwise it returns **nil.**

:getl *indicator-list* of **si:property-list-mixin** *Method*
The **:getl** message is like the **:get** message, except that the argument is a list of indicators. The **:getl** message searches down the property list for any of the indicators in *indicator-list,* until it finds a property whose indicator is one of those elements. It returns the portion of the property list beginning with the first such property that it found. If it doesn't find any, it returns **nil.**

:putprop *property indicator* of **si:property-list-mixin** *Method*
This gives the object an *indicator*-property of *property.*

:remprop *indicator* of **si:property-list-mixin** *Method*
This removes the object's *indicator* property, by splicing it out of the property list. It returns that portion of the list inside the object of which the former *indicator*-property was the **car.**

:push-property *value indicator* of **si:property-list-mixin** *Method*
The *indicator*-property of the object should be a list (note that **nil** is a list and an absent property is **nil**). This message sets the *indicator*-property of the object to a list whose **car** is *value* and whose **cdr** is the former *indicator*-property of the list. This is analogous to doing

```
(push value (get object indicator))
```

See the **push** special form.

:property-list of **si:property-list-mixin** *Method*

This returns the list of alternating indicators and values that implements the property list.

:set-property-list *list* of **si:property-list-mixin** *Method*

This sets the list of alternating indicators and values that implements the property list to *list*.

:property-list *list* (for **si:property-list-mixin)** *Init Option*

This initializes the list of alternating indicators and values that implements the property list to *list*.

Index

symbolics™

COND Conditions

Cambridge, Massachusetts

Signalling and Handling Conditions

# 990097

March 1984

This document corresponds to Release 5.0.

This document was prepared by the Documentation Group of Symbolics, Inc.

Printing year and number: 87 86 85 84 9 8 7 6 5 4 3 2 1

Table of Contents

List of Figures

1. Introduction

This document describes the condition signalling and handling system for the Lisp Machine. This document is for applications programmers.

In conjunction with using this document, you might want to review the discussion of flavors. See the document *Objects, Message Passing, and Flavors.*

1.1 Overview and Definitions

An *event* is "something that happens" during execution of a program. That is, it is some circumstance that the system can detect, like the effect of dividing by zero. Some events are errors — which means something happened that was not part of the contract of a given function — and some are not. In either case, a program can report that the event has occurred, and it can find and execute user-supplied code as a result.

The reporting process is called *signalling,* and subsequent processing is called *handling.* A *handler* is a piece of user-supplied code that assumes control when it is invoked as a result of signalling. Lisp Machine software includes default mechanisms to handle a standard set of events automatically.

The mechanism for reporting the occurrence of an event relies on flavors. Each standard class of events has a corresponding flavor called a *condition.* For example, occurrences of the event "dividing by zero" correspond to the condition **sys:divide-by-zero**.

The mechanism for reporting the occurrence of an event is called *signalling a condition.* The signalling mechanism creates a *condition object* of the flavor appropriate for the event. The condition object is an instance of that flavor. The instance contains information about the event, such as a textual message to report, and various parameters of the condition. For example, when a program divides a number by zero, the signalling mechanism creates an instance of the flavor **sys:divide-by-zero**.

Handlers are pieces of user or system code that are bound for a particular condition or set of conditions. When an event occurs, the signalling mechanism searches all of the currently bound handlers to find the one that corresponds to the condition. The handler can then access the instance variables of the condition object to learn more about the condition and hence about the event.

Handlers have dynamic scope, so that the handler that is invoked for a condition is the one that was bound most recently.

The condition system provides flexible mechanisms for determining what to do after a handler runs. The handler can try to *proceed*, which means that the program might be able to continue execution past the point at which the condition was signalled, possibly after correcting the error. Any program can designate *restart* points. This facility allows a user to retry an operation from some earlier point in a program.

Some conditions are very specific to a particular set of error circumstances and others are more general. For example, **fs:delete-failure** is a specialization of **fs:file-operation-failure** which is in turn a specialization of **fs:file-error**. You choose the level of condition that is appropriate to handle according to the needs of the particular application. Thus, a handler can correspond to a single condition or to a predefined class of conditions. This capability is provided by the flavor inheritance mechanism.

1.2 Overview of This Document

This document is for applications programmers. It contains descriptions of all conditions that are signalled by Lisp Machine software. With this information, you can write your own handlers for events detected by the system or define and handle classes of events appropriate for your own application.

This manual describes the following major topics.

- Mechanisms for handling conditions that have been signalled by system or application code.
- Mechanisms for defining new conditions.
- Mechanisms that are appropriate for application programs to use to signal conditions.
- All of the conditions that are defined by and used in the system software.

2. Conditions

This section provides an overview of how applications programs treat conditions.

- A program signals a condition when it wants to report an occurrence of an event.
- A program binds a handler when it wants to gain control when an event occurs.

When the system or a user function detects an error, it signals an appropriate condition and some handler bound for that condition then deals with it.

Conditions are flavors. Each condition is named by a symbol that is the name of a flavor, for example, **sys:unbound-variable**, **sys:divide-by-zero**, **fs:file-not-found**. As part of signalling a condition, the program creates a condition object of the appropriate flavor. The condition object contains information about the event, such as a textual message to report and various parameters. For example, a condition object of flavor **fs:file-not-found** contains the pathname that the file system failed to find.

Handlers are bound with dynamic scope, so the most recently bound handler for the condition is invoked. When an event occurs, the signalling mechanism searches all of the current handlers, starting with the innermost handler, for one that can handle the condition that has been signalled. When an appropriate handler is found, it can access the condition object to learn more about the error.

2.1 Example of a Handler

condition-case is a simple form for binding a handler. For example:

```
(condition-case ()
    (// a b)
  (sys:divide-by-zero *infinity*))
```

This form does two things.

- Evaluates **(// a b)** and returns the result.
- Binds a handler for the **sys:divide-by-zero** condition which applies during the evaluation of **(// a b)**.

In this example, it is a simple handler that just returns a value. If division by zero happened in the course of evaluating **(// a b)**, the form would return the value of ***infinity*** instead. If any other error occurred, it would be handled by the system's default handler for that condition or by some other user handler of higher scope.

You can also bind a handler for a predefined class of conditions. For example, the symbol **fs:file-operation-failure** refers to the set of all error conditions in file system operations, such as "file not found" or "directory not found" or "link to nonexistent file", but not to such errors as "network connection closed" or "invalid arguments to **open**", which are members of different classes.

2.2 Signalling

You can signal a condition by calling either **signal** or **error**. **signal** is the most general signalling function; it can signal any condition. It allows either a handler or the user to proceed from the error. **error** is a more restrictive version that accepts only error conditions and does not allow proceeding. **error** is guaranteed never to return to its caller.

Both **signal** and **error** have the same calling sequence. The first argument is a symbol that names a condition; the rest are keyword arguments that let you provide extra information about the error. See the section "Signalling Conditions". Full details on using the signalling mechanism are in that section.

Applications programs rarely need to signal system conditions although they can. Usually when you have a signalling application, you need to define a new condition flavor to signal it. Two simpler signalling functions, called **ferror** and **fsignal**, are applicable when you want to signal without defining a new condition.

It is very important to understand that signalling a condition is not just the same thing as throwing to a tag. ***throw** is a simple control-structure mechanism allowing control to escape from an inner form to an outer form. Signalling is a convention for finding and executing a piece of user-supplied code when one of a class of events occurs. A condition handler might in fact do a ***throw**, but it is under no obligation to do so. User programs can continue to use ***throw**; it is simply a different capability with a different application.

2.3 Condition Flavors

Symbols for conditions are the names of flavors; sets of conditions are defined by the flavor inheritance mechanism. For example, the flavor **lmfs:lmfs-file-not-found** is built on the flavor **fs:file-not-found**, which is built on **fs:file-operation-failure**, which is in turn built on the flavor **error**.

The flavor inheritance mechanism controls which handler is invoked. For example, when a Lisp Machine file system operation fails to find a file, it could signal **lmfs:lmfs-file-not-found**. The signalling mechanism invokes the first appropriate handler that it finds, in this case, a handler for **fs:file-not-found**, one for **fs:file-operation-failure**, or one for **error**. In general, if a handler is bound for flavor **a**, and a condition object **c** of flavor **b** is signalled, then the handler is invoked if **(typep c 'a)** is true; that is, if **a** is one of the flavors that **b** is built on.

The symbol **condition** refers to all conditions, including simple, error, and debugger conditions. The symbol **error** refers to the set of all error conditions. Figure 1 shows an overview of the flavor hierarchy.

error is a base flavor for many conditions, but not all. *Simple conditions* are those built on **condition**; *debugger conditions* are those built on **dbg:debugger-condition**. *Error conditions* or *errors* are those built on **error**. For your own condition definitions, whether you decide to treat something as an error or as a simple condition is up to the semantics of the application.

From a more technical viewpoint, the distinction between simple conditions and debugger conditions hinges on what action occurs when the program does not provide its own handler. For a debugger condition, the system invokes the Debugger; for a simple condition, **signal** simply returns **nil** to the caller.

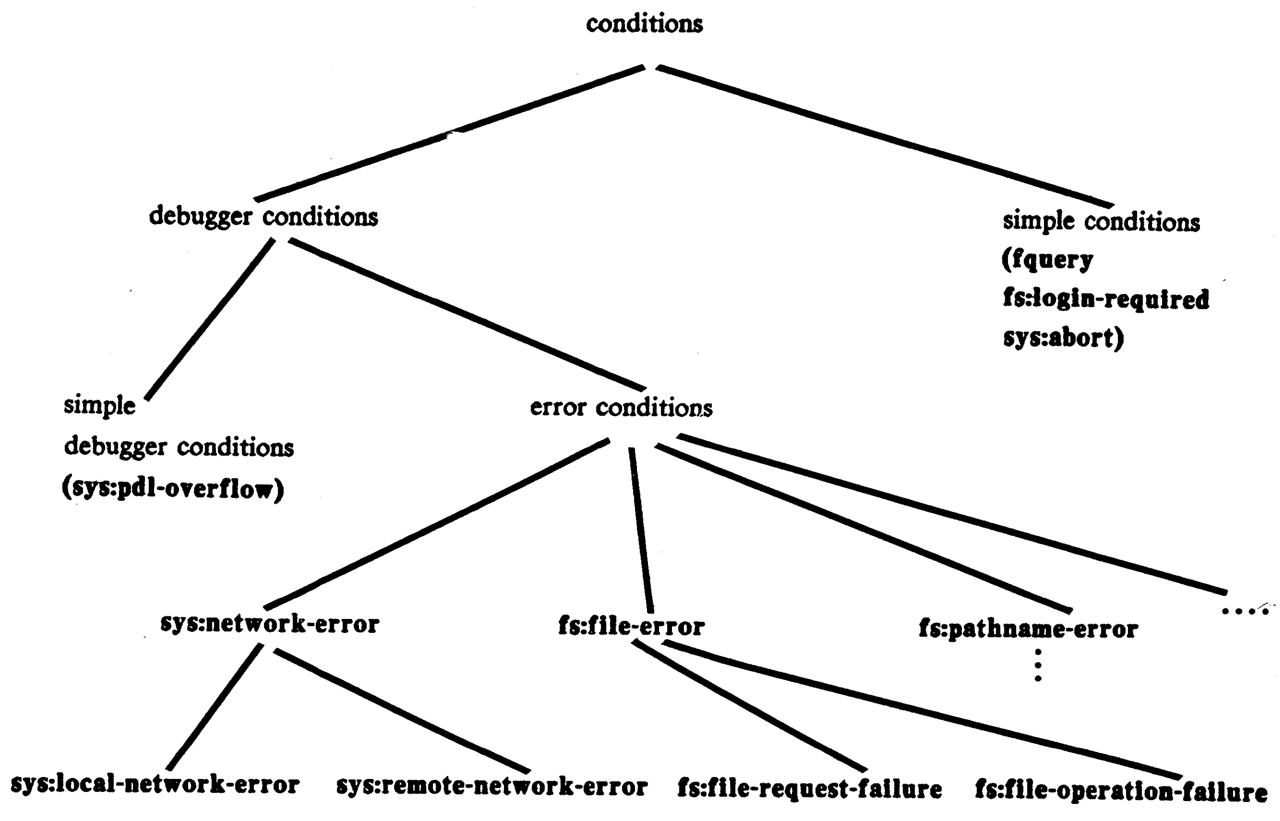

Figure 1. Condition flavor hierarchy

3. Creating New Conditions

An application might need to detect and signal events that are specific to the application. To support this, you need to define new conditions.

Defining a new condition is straightforward. For simple cases, you need only two forms: one defines the flavor, and the other defines a **:report** message. Build the flavor definition on either **error** or **condition**, depending on whether or not the condition you are defining represents an error. The following example defines an error condition.

```
(defflavor block-wrong-color () (error))

(defmethod (block-wrong-color :report) (stream)
   (format stream "The block was of the wrong color."))
```

Your program can now signal the error as follows:

```
(error 'block-wrong-color)
```

:report requires one argument, which is a stream for it to use in printing an error message. Its message should be a sentence, ending with a period and with no leading or trailing newlines.

The **:report** method must not depend on the dynamic environment in which it is invoked. That is, it should not do any free references to special variables. It should use only its own instance variables. This is because the condition object might receive a **:report** message in a dynamic environment that is different from the one in which it was created. This situation is common with **condition-case**.

The above example is adequate but does not take advantage of the power of the condition system. For example, the error message tells you only the class of event detected, not anything about this specific event. You can use instance variables to make condition objects unique to a particular event. For example, add instance variables **:block** and **:color** to the flavor so that **error** can use them to build the condition object:

```
(defflavor block-wrong-color (block color) (error)
  :initable-instance-variables
  :gettable-instance-variables)

(defmethod (block-wrong-color :report) (stream)
  (format stream "The block ~S was ~S, which is the wrong color."
          block color))
```

The **:initable-instance-variables** option defines the two instance variables **:block** and **:color**; the **:gettable-instance-variables** option defines methods for the **:block** and **:color** messages, which handlers can send to find out details of the condition.

Your program would now signal the error as follows:

```
(error 'block-wrong-color ':block the-bad-block
                          ':color the-bad-color)
```

The only other interesting thing to do when creating a condition is to define proceed types. See the section "Proceeding".

It is a good idea to use **compile-flavor-methods** for any condition whose instantiation is considered likely, to avoid the need for run-time combination and compilation of the flavor. See the macro **compile-flavor-methods**. Otherwise, the flavor must be combined and compiled the first time the event occurs, which causes perceptible delay.

3.1 Creating a Set of Condition Flavors

You can define your own sets of conditions and condition hierarchies. Just create a new flavor and build the flavors on each other accordingly. The base flavor for the set does not need a **:report** method if it is never going to be signalled itself. For example:

```
(defflavor block-world-error () (error))

(defflavor block-wrong-color (block color) (block-world-error)
  :initable-instance-variables)

(defflavor block-too-big (block container) (block-world-error)
  :initable-instance-variables)

(defmethod (block-too-big :report) (stream)
  (format stream "The block ~S is too big to fit in the ~S."
                 block container))

(defmethod (block-wrong-color :report) (stream)
  (format stream "The block ~S was ~S, which is the wrong color."
          block color))
```

4. Establishing Handlers

4.1 What is a Handler?

A handler consists of user-supplied code that is invoked when an appropriate condition signal occurs. Lisp Machine software includes default handlers for all standard conditions. Application programs need not handle all conditions explicitly but can provide handlers for just the conditions most relevant to the needs of the application.

4.2 Classes of Handlers

The simplest form of handler handles every error condition, each in the same way. The form for binding this handler is **ignore-errors**. In addition, four basic forms are available to bind handlers for particular conditions. Each of these has a standard version and a conditional variant:

- **condition-bind** and **condition-bind-if**
  condition-bind is the most general form. It allows the handler to run in the dynamic environment in which the error was signalled and to try to proceed from the error.

- **condition-bind-default** and **condition-bind-default-if**
  condition-bind-default is a variant of **condition-bind**. It binds a handler on the default condition list instead of the bound condition list. The distinction is described in these two sections. See the section "Signalling Conditions". See the section "Default Handlers and Complex Modularity".

- **condition-case** and **condition-case-if**
  condition-case is the simplest form to use. It returns to the dynamic environment in which the handler was bound and so does not allow proceeding.

- **condition-call** and **condition-call-if**
  condition-call is a more general version of **condition-case**. It uses user-specified predicates to select the clause to be executed.

In the conditional variants, the handlers are bound only if some expression is true.

4.3 Reference material

condition-bind *bindings body...* *Special Form*

condition-bind binds handlers for conditions and then evaluates its body with those handlers bound. One of the handlers might be invoked if a condition signal occurs while the body is being evaluated. The handlers bound have dynamic scope.

The following simple example sets up application-specific handlers for two standard error conditions, **fs:file-not-found** and **fs:delete-failure**.

```
(condition-bind ((fs:file-not-found 'my-fnf-handler)
                 (fs:delete-failure 'my-delete-handler))
  (deletef pathname))
```

The format for **condition-bind** is:

```
(condition-bind ((condition-flavor-1 handler-1)
                 (condition-flavor-2 handler-2)
                 ...
                 (condition-flavor-m handler-m))
  form-1
  form-2
  ...
  form-n)
```

condition-flavor-j	The name of a condition flavor or a list of names of condition flavors. The *condition-flavor-j* need not be unique or mutually exclusive. (See the section "Finding a Handler". Search order is explained in that section.)
handler-j	A form that is evaluated to produce a handler function. One handler is bound for each condition flavor clause in the list. The forms for binding handlers are evaluated in order from *handler-1* to *handler-m*. All the *handler-j* forms are evaluated and then all handlers are bound. When *handler* is a lambda-expression, it is compiled. (In a future release, it will be a proper lexical closure, capable of referring to the lexical variables of the containing block, but for now it is a separate top-level function.)
form-i	A body, constituting an implicit **progn**. The forms are evaluated sequentially. The **condition-bind** form returns whatever values *form-n* returns (**nil** when the body contains no forms). The handlers that are bound disappear when the **condition-bind** form is exited.

If a condition signal occurs for one of the *condition-flavor-j* during evaluation of the body, the signalling mechanism examines the bound handlers in the order in which they appear in the **condition-bind** form, invoking the first appropriate handler. You can think of the mechanism as being analogous to **typecase** or **selectq**. It invokes the handler function with one argument, the condition object. The handler runs in the dynamic environment in which the error occurred; no ***throw** is performed.

Any handler function can take one of three actions:

- It can return **nil** to indicate that it does not want to handle the condition after all. The handler is free to decide not to handle the condition, even though the *condition-flavor-j* matched. (In this case the signalling mechanism continues to search for a condition handler.)
- It can throw to some outer catch-form, using ***throw**.
- If the condition has any proceed types, it can proceed from the condition by sending a **:proceed** message to the condition object and returning the resulting values. In this case, **signal** returns all of the values returned by the handler function. (Proceed types are not available for conditions signalled with **error**. See the section "Proceeding".)

condition-bind-if *cond-form bindings body...* *Special Form*

condition-bind-if binds its handlers conditionally. In all other respects, it is just like **condition-bind**. It has extra subform called *cond-form*, for the conditional. Its format is:

```
(condition-bind-if cond-form
                   ((condition-flavor-1 handler-1)
                    (condition-flavor-2 handler-2)
                    ...
                    (condition-flavor-m handler-m))
  form-1
  form-2
  ...
  form-n)
```

condition-bind-if first evaluates *cond-form*. If the result is **nil**, it evaluates the handler forms but does not bind any handlers. It then executes the body as if it were a **progn**. If the result is not **nil**, it continues just like **condition-bind** binding the handlers and executing the body.

condition-bind-default *bindings body...* *Special Form*

condition-bind-default-if *cond-form bindings body...* *Special Form*

These forms bind their handlers on the default handler list instead of the bound handler list. (See the section "Finding a Handler".) In other respects **condition-bind-default** is just like **condition-bind**, and **condition-bind-default-if** is just like **condition-bind-if**. Such default handlers are examined by the signalling mechanism only after all of the bound handlers have been examined. Thus, a **condition-bind-default** can be overridden by a **condition-bind** outside of it. This advanced feature is described in more detail in another section. See the section "Default Handlers and Complex Modularity".

condition-case *(vars...) form clause...* *Special Form*

condition-case binds handlers for conditions, expressing the handlers as clauses of a case-like construct instead of as functions. The handlers bound have dynamic scope.

Examples:

```
(condition-case ()
    (time:parse string)
  (time:parse-error *default-time*))

(condition-case (e)
    (time:parse string)
  (time:parse-error
   (format error-output "~A, using default time instead." e)
   *default-time*))

(do () (nil)
  (condition-case (e)
      (return (time:parse string))
    (time:parse-error
      (setq string
            (prompt-and-read
              ':string
              "~A~%Use what time instead? " e)))))
```

The format is:

```
(condition-case (var1 var2 ...)
    form
  (condition-flavor-1 form-1-1 form-1-2 ... form-1-n)
  (condition-flavor-2 form-2-1 form-2-2 ... form-2-n)
  ...
  (condition-flavor-m form-m-1 form-m-2 ... form-m-n))
```

Each *condition-flavor-j* is either a condition flavor, a list of condition flavors, or **:no-error**. If **:no-error** is used, it must be the last of the handler clauses. The remainder of each clause is a body, a list of forms constituting an implicit **progn**.

condition-case binds one handler for each clause. The handlers are bound simultaneously.

If a condition is signalled during the evaluation of *form*, the signalling mechanism examines the bound handlers in the order in which they appear in the definition, invoking the first appropriate handler.

condition-case normally returns the values returned by *form*. If a condition is signalled during the evaluation of *form*, the signalling mechanism determines whether the condition is one of the *condition-flavor-j*. If so, the following actions occur:

1. It automatically performs a ***throw** to unwind the dynamic environment back to the point of the **condition-case**. This discards the handlers bound by the **condition-case**.
2. It executes the body of the corresponding clause.
3. It makes **condition-case** return the values produced by the last form in the handler clause.

While the clause is executing, *var1* is bound to the condition object that was signalled and the rest of the variables (*var2*, ...) are bound to **nil**. If none of the clauses needs to examine the condition object, you can omit *var1*.

```
(condition-case () ...)
```

As a special case, *condition-flavor-m* (the last one) can be the special symbol **:no-error**. If *form* is evaluated and no error is signalled during the evaluation, **condition-case** executes the **:no-error** clause instead of returning the values returned by *form*. The variables *var1*, *var2*, and so on are bound to the values produced by *form*, in the style of **multiple-value-bind**, so that they can be accessed by the body of the **:no-error** case. Any extra variables are bound to **nil**.

When an event occurs that none of the cases handles, the signalling mechanism continues to search the dynamic environment for a handler. You can provide a case that handles any **error** condition by using **error** as one *condition-flavor-j*.

condition-case-if *cond-form (vars...) form clause...* *Special Form*

condition-case-if binds its handlers conditionally. In all other respects, it is just like **condition-case**. Its syntax includes *cond-form*, a subform that controls binding handlers:

```
(condition-case-if cond-form (var)
   form
  (condition-flavor-1 form-1-1 form-1-2 ... form-1-n)
  (condition-flavor-2 form-2-1 form-2-2 ... form-2-n)
  ...
  (condition-flavor-m form-m-1 form-m-2 ... form-m-n))
```

condition-case-if first evaluates *cond-form*. If the result is **nil**, it does not set up any handlers; it just evaluates the form. If the result is not **nil**, it continues just like **condition-case**, binding the handlers and evaluating the form.

The **:no-error** clause applies whether or not *cond-form* is **nil**.

condition-call *(vars...) form clause...* *Special Form*

condition-call binds handlers for conditions, expressing the handlers as clauses of a case-like construct instead of as functions. These handlers have dynamic scope.

condition-call and **condition-case** have similar applications. The major distinction is that **condition-call** provides the mechanism for using a complex conditional criterion to determine whether or not to use a handler. **condition-call** clauses do not have the ability to decline to handle a condition because the clause is selected on the basis of the predicate, rather than on the basis of the type of a condition.

The format is:

```
(condition-call (var)
   form
  (predicate-1 form-1-1 form-1-2 ... form-1-n)
  (predicate-2 form-2-1 form-2-2 ... form-2-n)
  ...
  (predicate-m form-m-1 form-m-2 ... form-m-n))
```

Each *predicate-j* must be a function of one argument. The predicates are called, rather than evaluated. The *form-j-i* are a body, a list of forms constituting an implicit **progn**. The handler clauses are bound simultaneously.

When a condition is signalled, each predicate in turn (in the order in which they appear in the definition) is applied to the condition object. The corresponding handler clause is executed for the first predicate that returns a value other than **nil**. The predicates are called in the dynamic environment of the signaller.

condition-call takes the following actions when it finds the right predicate:

1. It automatically performs a ***throw** to unwind the dynamic environment back to the point of the **condition-call**. This discards the handlers bound by the **condition-call**.

2. It executes the body of the corresponding clause.

3. It makes **condition-call** return the values produced by the last form in the clause.

During the execution of the clause, the variable *var* is bound to the condition object that was signalled. If none of the clauses needs to examine the condition object, you can omit *var:*

```
(condition-call () ...)
```

condition-call and :no-error

As a special case, *predicate-m* (the last one) can be the special symbol **:no-error**. If *form* is evaluated and no error is signalled during the evaluation, **condition-case** executes the **:no-error** clause instead of returning the values returned by *form*. The variables *vars* are bound to the values produced by *form*, in the style of **multiple-value-bind**, so that they can be accessed by the body of the **:no-error** case. Any extra variables are bound to **nil**.

Some limitations on predicates:

- Predicates must not have side effects. The number of times that the signalling mechanism chooses to invoke the predicates and the order in which it invokes them are not defined. For side effects in the dynamic environment of the signal, use **condition-bind**.
- The predicates are not lexical closures and therefore cannot access variables of the lexically containing form, unless those variables are declared **special**.
- Lambda-expression predicates are not compiled.

condition-call-if *cond-form (vars...) form clause...* *Special Form*

condition-call-if binds its handlers conditionally. In all other respects, it is just like **condition-call**. Its format includes *cond-form*, the subform that controls binding handlers:

```
(condition-call-if cond-form (var)
  form
  (predicate-1 form-1-1 form-1-2 ... form-1-n)
  (predicate-2 form-2-1 form-2-2 ... form-2-n)
  ...
  (predicate-m form-m-1 form-m-2 ... form-m-n))
```

condition-call-if first evaluates *cond-form*. If the result is **nil**, it does not set up any handlers; it just evaluates the form. If the result is not **nil**, it continues just like **condition-call**, binding the handlers and evaluating the form.

The **:no-error** clause applies whether or not *cond-form* is **nil**.

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

ignore-errors sets up a very simple handler on the bound handlers list that handles all error conditions. Normally, it executes *body* and returns the first value of the last form in *body* as its first value and **nil** as its second value. If an error signal occurs while *body* is executing, **ignore-errors** immediately returns with **nil** as its first value and something not **nil** as its second value.

ignore-errors replaces **errset** and **catch-error**.

4.4 Application: Handlers Examining the Stack

condition-bind handlers are invoked in the dynamic environment in which the error is signalled. Thus the Lisp stack still holds the frames that existed when the error was signalled. A handler can examine the stack using the functions described in this section.

One important application of this facility is for writing error logging code. For example, network servers might need to keep providing service even though no user is available to run the Debugger. By using these functions, the server can record some information about the state of the stack into permanent storage, so that a maintainer can look at it later and determine what went wrong.

These functions return information about stack frames. Each stack frame is identified by a *frame pointer*, represented as a Lisp locative pointer. In order to use any of these functions, you need to have appropriate environment bindings set up. The macro **dbg:with-erring-frame** both sets up the environment properly and provides a frame pointer to the stack frame that got the error. Within the body of that macro, use the appropriate functions to move up and down stack frames; these functions take a frame pointer and return a new frame pointer by following links in the stack.

These frame-manipulating functions are actually *subprimitives*, even though they do not have a % sign in their name. If given an argument that is not a frame pointer, they stand a good chance of crashing the machine. Use them with care.

The functions that return new frame pointers work by going to the *next frame* or the *previous frame* of some category. "Next" means the frame of a procedure that was invoked more recently (the frame called by this one; toward the top of the stack). "Previous" means the frame of a procedure that was invoked less recently (the caller of this frame; towards the base of the stack).

These functions assume three categories of frames: *interesting active* frames, *active* frames, and *open* frames.

- An active frame simply means a procedure that is currently running (or active) on the stack.
- Interesting active frames include all of the active frames except those that are parts of the internals of the Lisp interpreter, such as frames for **eval**, **apply**, **funcall**, **let**, and other basic Lisp special forms. The list of such functions is the value of the system constant **dbg:*uninteresting-functions***.
- Open frames include all the active frames as well as frames that are still under construction, for functions whose arguments are still being computed. See the section "How to Read Assembly Language (LM-2)". On the 3600, open frames and active frames are synonymous.

4.4.1 Reference material

The functions in this section all take a frame pointer as an argument. For functions that indicate a direction on the stack, using **nil** as the argument indicates the frame at relevant end of the stack. For example, when you are using a function that looks up the stack, **nil** as the argument indicates the top-most stack frame.

Remember to use the functions in this section only within the context of the **dbg:with-erring-frame** macro.

dbg:with-erring-frame *(var object)* *Macro*

dbg:with-erring-frame sets up an environment with appropriate bindings for using the rest of the functions that examine the stack. It binds *var* with the frame pointer to the stack frame that signalled the error. *var* is always a pointer to an interesting stack frame. *object* is the condition object for the error, which was the first argument given to the **condition-bind** handler.

```
(defun my-handler (condition-object)
  (dbg:with-erring-frame (frame-ptr condition-object)
    body...))
```

Inside *body*, the variable **frame-ptr** is bound to the frame pointer of the frame that got the error.

dbg:get-frame-function-and-args *frame-pointer* *Function*

dbg:get-frame-function-and-args returns a list containing the name of the function for *frame-pointer* and the values of the arguments.

dbg:frame-next-active-frame *frame-pointer* *Function*

dbg:frame-next-active-frame returns a frame pointer to the next active frame following *frame-pointer*. If *frame-pointer* is the last active frame on the stack, it returns **nil**.

dbg:frame-next-interesting-active-frame *frame-pointer* *Function*

dbg:frame-next-interesting-active-frame returns a frame pointer to the next interesting active frame following *frame-pointer*. If *frame-pointer* is the last interesting active frame on the stack, it returns **nil**.

dbg:frame-next-open-frame *frame-pointer* *Function*

dbg:frame-next-open-frame returns a frame pointer to the next open frame following *frame-pointer*. If *frame-pointer* is the last open frame on the stack, it returns **nil**.

dbg:frame-previous-active-frame *frame-pointer* *Function*

dbg:frame-previous-active-frame returns a frame pointer to the previous active frame before *frame-pointer*. If *frame-pointer* is the first active frame on the stack, it returns **nil**.

dbg:frame-previous-interesting-active-frame *frame-pointer* *Function*

dbg:frame-previous-interesting-active-frame returns a frame pointer to the previous interesting active frame before *frame-pointer.* If *frame-pointer* is the first interesting active frame on the stack, it returns **nil.**

dbg:frame-previous-open-frame *frame-pointer* *Function*

dbg:frame-previous-open-frame returns a frame pointer to the previous open frame before *frame-pointer.* If *frame-pointer* is the first open frame on the stack, it returns **nil.**

dbg:frame-next-nth-active-frame *frame-pointer* &optional *(count 1)* *Function*

dbg:frame-next-nth-active-frame goes up the stack by *count* active frames from *frame-pointer* and returns a frame pointer to that frame. It returns a second value that is not **nil.** When count is positive, this is like calling **dbg:frame-next-active-frame** *count* times; *count* can also be negative or zero. If either end of the stack is reached, it returns a frame pointer to the first or last active frame and **nil.**

dbg:frame-next-nth-interesting-active-frame *frame-pointer* &optional *(count 1)* *Function*

dbg:frame-next-nth-interesting-active-frame goes up the stack by *count* interesting active frames from *frame-pointer* and returns a frame pointer to that frame. It returns a second value that is not **nil.** When count is positive, this is like calling **dbg:frame-next-interesting-active-frame** *count* times; *count* can also be negative or zero. If either end of the stack is reached, it returns a frame pointer to the first or last active frame and **nil.**

dbg:frame-next-nth-open-frame *frame-pointer* &optional *(count 1)* *Function*

dbg:frame-next-nth-open-frame goes up the stack by *count* open frames from *frame-pointer* and returns a frame pointer to that frame. It returns a second value that is not **nil.** When count is positive, this is like calling **dbg:frame-next-open-frame** *count* times; *count* can also be negative or zero. If either end of the stack is reached, it returns a frame pointer to the first or last active frame and **nil.**

dbg:frame-out-to-interesting-active-frame *frame-pointer* *Function*

dbg:frame-out-to-interesting-active-frame returns either *frame-pointer* (if it points to an interesting active frame) or the previous interesting active frame before *frame-pointer.* (This is what the `c-m-U` command in the debugger does.)

dbg:frame-active-p *frame-pointer* *Function*

dbg:frame-active-p indicates whether *frame-pointer* is an active frame.

Value	*Meaning*
nil	Frame is not active
not **nil**	Frame is active

dbg:frame-real-function *frame-pointer* *Function*

dbg:frame-real-function returns either the function object associated with *frame-pointer* or **self** when the frame was the result of sending a message to an instance.

dbg:frame-total-number-of-args *frame-pointer* *Function*

dbg:frame-total-number-of-args returns the number of arguments that were passed in *frame-pointer*. For functions that take an **&rest** parameter, each argument is counted separately. On the 3600, sending a message to an instance results in two implicit arguments being passed internally along with the other arguments. These implicit arguments are included in the count.

dbg:frame-number-of-spread-args *frame-pointer* &optional *(type* **':supplied***)* *Function*

dbg:frame-number-of-supplied-args returns the number of "spread" arguments that were passed in *frame-pointer*. (These are the arguments that are not part of a **&rest** parameter.) On the 3600, sending a message to an instance results in two implicit arguments being passed internally along with the other arguments. These implicit arguments are included in the count.

type requests more specific definition of the number:

Value	*Meaning*
:supplied	Returns the number of arguments that were actually passed by the caller, except for arguments that were bound to a **&rest** parameter. This is the default.
:expected	Returns the number of arguments that were expected by the function being called.
:allocated	Returns the number of arguments for which stack locations have been allocated. In the absence of a **&rest** parameter, this is the same as **:expected** for compiled functions, and the same as **:supplied** for interpreted and LM-2 microcoded functions. If stack locations were allocated for arguments that were bound to a **&rest** parameter, they are included in the returned count.

These values would all be the same except in cases where a wrong-number-of-arguments error occurred, or where there are optional arguments (expected but not supplied).

dbg:frame-arg-value *frame-pointer n* &optional *callee-context no-error-p* *Function*

dbg:frame-arg-value returns the value of the *n*th argument to *frame-pointer*. It returns a second value, which is a locative pointer to the word in the stack that holds the argument. If *n* is out of range, then it takes action based on *no-error-p*: if *no-error-p* is **nil**, it signals an error, otherwise it returns **nil**. *n* can also be the name of the argument (a symbol, but it need not be in the right package). Each argument passed for a &rest parameter counts as a separate argument when *n* is a number. On the LM-2, this function ignores its third argument. On the 3600, it controls whether you get the caller or callee copy of the argument (original or possibly modified.)

dbg:frame-number-of-locals *frame-pointer* *Function*

dbg:frame-number-of-locals returns the number of local variables allocated for *frame-pointer*.

dbg:frame-local-value *frame-pointer n* &optional *no-error-p* *Function*

dbg:frame-local-value returns the value of the *n*th local variable in *frame-pointer*. *n* can also be the name of the local variable (a symbol, but it need not be in the right package). It returns a second value, which is a locative pointer to the word in the stack that holds the local variable. If *n* is out of range, then the action is based on *no-error-p*: if *no-error-p* is **nil**, it signals an error, otherwise it returns **nil**.

dbg:frame-self-value *frame-pointer* &optional *instance-frame-only* *Function*

dbg:frame-self-value returns the value of **self** in *frame-pointer*, or **nil** if **self** does not have a value. If *instance-frame-only* is not **nil** then it returns **nil** unless this frame is actually a message-sending frame created by **send**.

dbg:frame-real-value-disposition *frame-pointer* *Function*

dbg:frame-real-value-disposition returns a symbol indicating how the calling function is going to handle the values to be returned by this frame. If the calling function just returns the values to its caller, then the symbol indicates how the final recipient of the values is going to handle them.

Value	*Meaning*
:ignore	The values would be ignored; the function was called for effect.
:single	The first value would be received and the rest would not; the function was called for value.
:multiple	All the values would be received; the function was called for multiple values. It returns a second value indicating the number of values expected. **nil** indicates an indeterminate number. The 3600 currently always returns **nil**.

dbg:print-function-and-args *frame-pointer* &optional *show-pc-p* *Function*

dbg:print-function-and-args prints the name of the function executing in *frame-pointer* and the names and values of its arguments, in the same format as the Debugger uses. If *show-pc-p* is true, the program counter value of the frame, relative to the beginning of the function, is printed in octal. **dbg:print-function-and-args** returns the number of local slots occupied by arguments.

dbg:print-frame-locals *frame-pointer local-start* &optional *(indent 0)* *Function*

dbg:print-frame-locals prints the names and values of the local variables of *frame-pointer*. *local-start* is the first local slot number to print; the value returned by **dbg:print-function-and-args** is often suitable for this. *indent* is the number of spaces to indent each line; the default is no indentation.

5. Signalling Conditions

5.1 Signalling Mechanism

The following functions and macros invoke the signalling mechanism, which finds and invokes a handler for the condition.

error
signal
ferror
fsignal
signal-proceed-case

5.1.1 Finding a Handler

The signalling mechanism finds a handler by inspecting four lists of handlers, in this order:

1. It first looks down the list of *bound* handlers, which are handlers set up by **condition-bind**, **condition-case**, and **condition-call** forms.
2. Next, it looks down the list of *default* handlers, which are set up by **condition-bind-default**.
3. Next, it looks down the list of *interactive* handlers. This list normally contains only one handler, which enters the Debugger if the condition is based on **dbg:debugger-condition** and declines to handle it otherwise.
4. Finally, it looks down the list of *restart* handlers, which are set up by **error-restart**, **error-restart-loop**, and **catch-error-restart**. See the section "Default Handlers and Complex Modularity". See the section "Restart Handlers".
5. If it gets to the end of the last list without finding a willing handler, one of two things happens.
   - **signal** returns **nil** when both of the following are true:
     - The condition was signalled with **signal**, **fsignal**, or **signal-proceed-case**.
     - The condition object is not an instance of a condition based on **error**.
   - The Debugger assumes control.

The signalling mechanism checks each handler to see if it is willing to handle the condition. Some handlers have the ability to decline to handle the condition, in which case the signalling mechanism keeps searching. It calls the first willing handler it finds.

As we have seen, the signalling mechanism searches for handlers in a specific order. It looks at all the bound handlers before any of the default handlers and all of the default handlers before any of the restart handlers. Thus, it tries any **condition-bind** handler before any handler bound by **condition-bind-default**, even though the **condition-bind-default** is within the dynamic scope of the **condition-bind**. Similarly, it considers a **condition-bind** handler before an **error-restart** handler, even when the **error-restart** handler was bound more recently. See the section "Default Handlers and Complex Modularity".

While a bound or default handler is executing, that handler and all handlers inside it are removed from the list of bound or default handlers. This is to prevent infinite recursion when a handler signals the same condition that it is handling, as in the following simplistic example:

```
(condition-bind ((error '(lambda (x) (ferror "foo"))))
  (ferror "foo"))
```

If you want recursion, the handler should bind its own condition.

5.1.2 Signalling Simple Conditions

If a simple condition or a debugger condition not based on **error** is signalled, the signalling mechanism searches for a handler on the bound handler and default handler lists. When it finds one, it invokes it. Otherwise, the first restart handler for that condition handles it. If no restart handler for the condition is found, **signal** returns **nil**; **error** enters the Debugger.

5.1.3 Signalling Errors

In practice, if the **signal** function is applied to an error condition object, **signal** is very unlikely to return **nil**, because most processes contain a restart handler that handles all error conditions. The function at the base of the stack of most processes contains a **catch-error-restart** form that handles **error** and **sys:abort**. Thus, if you are in the Debugger as a result of an error, you can always use `ABORT`. The restart handler at the base of the stack always handles **sys:abort** and either terminates or restarts the process.

5.1.4 Restriction Due to Scope

A condition must be signalled only in the environment in which the event that it represents took place, to insure that handlers run in the proper dynamic environment. Therefore, you cannot signal a condition object that has already been signalled once. In particular, when you are writing a handler, you cannot have that handler signal its condition argument. Similarly, if a condition object is returned by some program (such as the **open** function given **nil** for the **:error** keyword), you cannot signal that object.

It is not correct to pass on the condition by signalling the handler's condition argument. This is incorrect:

```
(defun condition-handler (condition)
   (if something (*throw ...) (signal condition)))
```

Instead you should do this:

```
(defun condition-handler (condition)
   (if something (*throw ...) nil))
```

or this:

```
(defun condition-handler (condition)
   (if something (*throw ...) (signal 'some-other-condition)))
```

5.2 Reference material

signal *flavor-name* &rest *init-options* *Function*

signal is the primitive function for signalling a condition. The argument *flavor-name* is a condition flavor symbol. The *init-options* are the init options when the **condition-object** is created; they are passed in the **:init** message to the instance. (See the function **make-instance**.) **signal** creates a new condition object of the specified flavor, and signals it. If no handler handles the condition and the object is not an error object, **signal** returns **nil**. If no handler handles the condition and the object is an error object, the Debugger assumes control.

In a more advanced form of **signal**, *flavor-name* can be a condition object that has been created with **make-condition** but not yet signalled. In this case, *init-options* is ignored.

error *flavor-name* &rest *init-options* *Function*

error is the function for signalling a condition that is not proceedable. The argument *flavor-name* is a condition flavor symbol or an error object, created by **make-condition**. The *init-options* are the init options specified when the error object is created; they are passed in the **:init** message. **error** is similar to **signal** but restricted in the following ways:

- **error** sets the proceed types of the error object to **nil** so that it cannot be proceeded.
- If no handler exists, the Debugger assumes control, whether or not the object is an error object.

- **error** never returns to its caller.

In a more advanced form of **error**, *flavor-name* can be a condition object that has been created with **make-condition** but not yet signalled. In this case, *init-options* is ignored.

For compatibility with the old Maclisp **error** function, **error** tries to determine that it has been called with Maclisp-style arguments and turns into an **fsignal** or **ferror** as appropriate. If *flavor-name* is a string or a symbol that is not the name of a flavor, and **error** has no more than three arguments, **error** assumes it was called with Maclisp-style arguments.

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

fsignal is a simple function for signalling when you do not care to use a particular condition. **fsignal** signals **dbg:proceedable-ferror**. (See the flavor **dbg:proceedable-ferror**.) The arguments are passed as the **:format-string** and **:format-args** init keywords to the error object.

ferror *format-string* &rest *format-args* *Function*

ferror is a simple function for signalling when you do not care what the condition is. **ferror** signals **ferror**. (See the flavor **ferror**.) The arguments are passed as the **:format-string** and **:format-args** init keywords to the error object.

The old **(ferror nil ...)** syntax continues to be accepted for compatibility reasons indefinitely; the **nil** is ignored. An error is signalled if the first argument is a symbol other than **nil**; the first argument must be **nil** or a string.

sys:parse-ferror *format-string* &rest *format-args* *Function*

Signals an error of flavor **sys:parse-ferror**. *format-string* and *format-args* are passed as the **:format-string** and **:format-args** init options to the error object.

See the flavor **sys:parse-ferror**.

errorp *object* *Function*

errorp returns **t** if *object* is an error object, and **nil** otherwise. That is:

```
(errorp x) <=> (typep x 'error)
```

make-condition *flavor-name* &rest *init options* *Function*

make-condition creates a condition object of the specified flavor with the specified init-options. This object can then be signalled by passing it to **signal** or **error**. Note that you are not supposed to design functions that indicate errors by *returning* error objects; functions should always indicate errors by *signalling* error objects. This function makes it possible to build complex systems that use subroutines to generate condition objects so that their callers can signal them.

check-arg *var-name predicate description* *Macro*

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

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

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

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

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

var-name is the name of the variable whose value is of the wrong type. If the error is proceeded this variable will be **setq**'ed to a replacement value. *predicate* is a test for whether the variable is of the correct type. It can be either a symbol whose function definition takes one argument and returns non-**nil** if the type is correct, or it can be a nonatomic form which is evaluated to check the type, and presumably contains a reference to the variable *var-name*. *description* is a string which expresses *predicate* in English, to be used in error messages.

The *predicate* is usually a symbol such as **fixp**, **stringp**, **listp**, or **closurep**, but when there isn't any convenient predefined predicate, or when the condition is complex, it can be a form. For example:

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

If this error got to the Debugger, the message

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

would be printed.

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

check-arg uses *predicate* to determine whether the value of the variable is of the correct type. If it is not, **check-arg** signals the **sys:wrong-type-argument** condition. See the flavor **sys:wrong-type-argument**.

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

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

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

foo is the name of an argument whose value should be a number. **:number**

is a value which is passed as a second argument to **typep**; that is, it is a symbol that specifies a data type. The English form of the type name, which gets put into the error message, is found automatically.

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

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

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

argument-typecase *arg-name* &body *clauses* *Special Form*

argument-typecase is a hybrid of **typecase** and **check-arg-type**. Its clauses look like clauses to **typecase**. **argument-typecase** automatically generates an **otherwise** clause which signals an error. The proceed types to this error are similar to those from **check-arg**; that is, you can supply a new value that replaces the argument that caused the error.

For example, this:

```
(defun foo (x)
 (argument-typecase x
   (:symbol (print 'symbol))
   (:number (print 'number))))
```

is the same as this:

```
(defun foo (x)
  (check-arg x
    (typecase x
      (:symbol (print 'symbol) t)
      (:number (print 'number) t)
      (otherwise nil))
    "a symbol or a number"))
```

6. Default Handlers and Complex Modularity

When more than one handler exists for a condition, which one should be invoked? The signalling mechanism has an elaborate rule, but in practice, it usually invokes the innermost handler. See the section "Finding a Handler". "Innermost" is defined dynamically and thus means "the most recently bound handler".

This decision is made on the basic principle of modularity and referential transparency: a function should behave the same way, regardless of what calls it. Therefore, whether a handler bound by a function gets invoked should not depend on what is going on with that function's callers.

For example, suppose function **a** sets up a handler to deal with the **fs:file-not-found** condition, and then calls procedure **b** to perform some service for it. Now, unbeknownst to **a**, **b** sometimes opens a file, and **b** has a condition handler for **fs:file-not-found**. If **b**'s file is not found, **b**'s handler handles the error rather than **a**'s. This is as it should be, because it should not be visible to **a** that **b** uses a file (this is a hidden implementation detail of **b**). **a**'s unrelated condition handler should not meddle with **b**'s internal functioning. Therefore, the signalling mechanism follows a basic inside-to-outside searching rule.

Sometimes a function needs to signal a condition but still handle the condition itself if none of its callers handles it. On first encounter, this need seems to require an outside-to-inside searching rule instead of the inside-to-outside searching rule mandated by modularity considerations. How can you circumvent the rules to allow a function to handle something only if no outer function handles it?

Several strategies are available for dealing with this. Because of our lack of experience with the condition signalling system, we are not yet sure which of these are better than others. We are providing several mechanisms in order to allow experimentation and flexibility.

- The simplest solution is to provide a proceed type for proceeding from the Debugger. That is, your program signals an error to allow callers to handle the condition. If none of them handles it, the Debugger assumes control. Provided that the user decides to use the proceed type, your program then gets to handle the condition. If what your program wanted to do was to prompt the user anyway, this might be the right thing. This is most likely true if you think that a program error is probably happening and the user might want to be able to trace and manipulate the stack using the Debugger.
- Another simple solution is to signal a condition that is not an error. **signal** returns **nil** when no handler is found, and your program can take appropriate action.
- Use **condition-bind-default** to create a handler on the default handler list.

The signalling mechanism searches this list only after searching through all regular bound handlers. One drawback of this scheme is that it works only to one level. If you have three nested functions, you cannot get outside-to-inside modularity for all three, because only two lists exist, the bound list and the default list. This facility is probably good enough for some applications however.

- Use **dbg:condition-handled-p** to determine whether a handler has been bound for a particular flavor. This has the advantage that it works for any number of levels of nested handler, instead of only two. One disadvantage is that it can return **:maybe**, which is ambiguous.

The simple solutions work only if your program is doing the signalling. If some other program is signalling a condition, you cannot control whether the condition is an error condition or whether it has any proceed types; you can only write handlers.

6.1 Reference Material

dbg:condition-handled-p *condition-flavor* *Function*

dbg:condition-handled-p searches the bound handler list and the default handler list to see whether a handler exists for *condition-flavor*. This function should be called only from a **condition-bind** handler function. It starts looking from the point in the lists from which the current handler was invoked and proceeds to look outwards through the bound handler list and the default handler list. It returns a value to indicate what it found:

Value	*Meaning*
:maybe	**condition-bind** handlers for the flavor exist. These handlers are permitted to decline to handle the condition. You cannot determine what would happen without actually running the handler.
nil	No handler exists.
t	A handler exists.

7. Interactive Handlers

The interactive handler list contains one element: a handler that invokes the Debugger if the condition is built on **dbg:debugger-condition** and declines to handle the condition if it is not. No standard procedure exists for changing the contents of this list.

One of the original design goals of the condition signalling mechanism was to support building complex applications that could take over the function of the Debugger and provide their own. The exact definition of the problem is not completely clear however. We are not sure whether the current system provides this functionality.

If you are writing an application that needs to take over error handling completely, you might be able to create a **condition-bind** handler that handles **error**, to prevent invocation of the Debugger. This strategy might have problems that we have not anticipated. If you really need to get the Debugger out of the way, you might try changing the interactive handler list. We have not defined a way to do this; read the code for complete details. We cannot guarantee that whatever you do will work in future releases. However, we encourage your experimentation. Please contact us so that we can help you if possible.

Briefly, the variable holding the list is named **dbg:*interactive-handlers***, which holds an interactive handler object. The list is reset to hold the standard Debugger when you warm boot the machine.

An interactive handler object must handle the following messages:

:handle-condition-p *cond* *Message*

> **:handle-condition-p** examines *cond* which is a condition object. It returns **nil** it if declines to handle the condition and something other than **nil** when it is prepared to handle the condition.

:handle-condition *cond ignore* *Message*

> *cond* is a condition object. You should handle this condition, ignoring the second argument. **:handle-condition** can return values or throw in the same way that **condition-bind** handlers can.

8. Restart Handlers

One way to handle an error is to restart at some earlier point the program that got the error. A program can specify points where it is safe or convenient for it to be restarted should a condition signal occur during processing a function. The basic special form for doing this is called **error-restart**. The following example is taken from the system code:

```
(defun connect (address contact-name
                &optional (window-size default-window-size)
                (timeout (* 10. 60.))
                &aux conn real-address (try 0))
  (error-restart (connection-error
                   "Retry connection to ~A at ~S with longer timeout"
                   address contact-name)
    forms...))
```

This code fragment evaluates *forms* and returns the final value(s) if successful. If the Debugger assumes control as a result of a **chaos:connection-error** condition, the user is given the opportunity of restarting the program. The Debugger's prompt message would be something like this:

What does a contact-name look like:

```
s-A: "Retry connection to SCRC at FILE 1 with longer timeout"
```

If the user were to press s-A at this point, the forms implementing the connection would be evaluated again. That is, the body of the **error-restart** would be started again from the beginning.

Two variations on this basic paradigm are provided. **error-restart-loop** is an infinite loop version of **error-restart**. It always starts over regardless of whether a condition has been signalled. **catch-error-restart** never restarts, even when a condition is signalled. Instead it always returns, returning either the values from the body (if successful) or **nil** if a condition signal occurred.

catch-error-restart is the most primitive version of this control structure. The other two are built from it. It too has a conditional variant, **catch-error-restart-if,** for binding a restart handler conditionally.

A common paradigm is to use one of these forms in the command loop of an interactive program, with *condition-flavor* being **(error sys:abort)**. This way, if an unhandled error occurs, the user is offered the option of returning to the command loop, and the ABORT key returns to the command loop. Which form you use depends on the nature of your command loop.

8.1 Reference material

The use of "error-" in the names of these functions has no real significance. They could have been called **condition-restart, condition-restart-loop**, and so on, because they apply to all conditions.

error-restart *(condition-flavor format-string format-arg...) body...* *Special Form*

This form establishes a restart handler for *condition-flavor* and then evaluates the body. If the handler is not invoked, **error-restart** returns the values produced by the last form in the body and the restart handler disappears. When the restart handler is invoked, control is thrown back to the dynamic environment inside the **error-restart** form and execution of the body starts all over again. The format is:

```
(error-restart (condition-flavor format-string . format-args)
  form-1
  form-2
  ...)
```

condition-flavor is either a condition or a list of conditions that can be handled. *format-string* and *format-args* are a control string and a list of arguments (respectively) to be passed to **format** to construct a meaningful description of what would happen if the user were to invoke the handler. *format-args* are evaluated when the handler is bound. The Debugger uses these values to create a message explaining the intent of the restart handler.

Note: this is not compatible with the definition of **error-restart** in previous releases, (System 210 and earlier).

error-restart-loop *(condition-flavor format-string format-args...) body...* *Special Form*

error-restart-loop establishes a restart handler for *condition-flavor* and then evaluates the body. If the handler is not invoked, **error-restart-loop** evaluates the body again and again, in an infinite loop. Use the **return** function to leave the loop. This mechanism is useful for interactive top levels.

If a condition is signalled during the execution of the body and the restart handler is invoked, control is thrown back to the dynamic environment inside the **error-restart-loop** form and execution of the body is started all over again. The format is:

```
(error-restart-loop (condition-flavor format-string . format-args)
  form-1
  form-2
  ...)
```

condition-flavor is either a condition or a list of conditions that can be handled. *format-string* and *format-args* are a control string and a list of arguments (respectively) to be passed to **format** to construct a meaningful description of what would happen if the user were to invoke the handler. The Debugger uses these values to create a message explaining the intent of the restart handler.

catch-error-restart *(condition-flavor format-string . format-args) body...* *Special Form*

catch-error-restart establishes a restart handler for *condition-flavor* and then evaluates the body. If the handler is not invoked, **catch-error-restart** returns the values produced by the last form in the body, and the restart handler disappears. If a condition is signalled during the execution of the body and the restart handler is invoked, control is thrown back to the dynamic environment of the **catch-error-restart** form. In this case, **catch-error-restart** also returns **nil** as its first value and something other than **nil** as its second value. Its format is:

```
(catch-error-restart (condition-flavor format-string . format-args)
  form-1
  form-2
  ...)
```

condition-flavor is either a condition or a list of conditions that can be handled. *format-string* and *format-args* are a control string and a list of arguments (respectively) to be passed to **format** to construct a meaningful description of what would happen if the user were to invoke the handler. The Debugger uses these values to create a message explaining the intent of the restart handler.

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

catch-error-restart-if establishes its restart handler conditionally. In all other respects, it is the same as **catch-error-restart**. Its format is:

```
    (catch-error-restart-if cond-form
       (condition-flavor format-string . format-args)
      form-1
      form-2
      ...)
```

catch-error-restart-if first evaluates *cond-form*. If the result is **nil**, it evaluates the body as if it were a **progn** but does not establish any handlers. If the result is not **nil**, it continues just like **catch-error-restart**, establishing the handlers and executing the body.

8.2 Invoking Restart Handlers Manually

dbg:invoke-restart-handlers *object* &key *flavors* *Function*

> **dbg:invoke-restart-handlers** searches the list of restart handlers to find a restart handler for *object.* The *flavors* argument controls which restart handlers are examined. *flavors* is a list of condition names. When *flavors* is omitted, the function examines every restart handler. When *flavors* is provided, the function examines only those restart handlers that handle at least one of the conditions on the list.
>
> The first restart handler that it finds to handle the condition is invoked and given *object.* It returns **nil** if no appropriate restart handler is found.

dbg:invoke-restart-handlers can be called by handlers set up by **condition-bind** or **condition-bind-default**. The *object* argument should be the condition object passed to the handler. The handler calls this function to bypass the interactive handlers list, letting the innermost restart handler handle the condition. A program that wants to attempt to continue with a computation in the presence of errors might find this useful. For example, it could be used to support batch-mode compilation, with the user away from the console.

9. Proceeding

In some situations, execution can proceed past the point at which a condition was signalled. Events for which this is the case are called *proceedable conditions*. Some external agent makes the decision about whether it is reasonable to proceed after repairing the original problem. The agent is either a **condition-bind** handler or the user operating the Debugger.

In general, many different ways are available to proceed from a particular condition. Each way is identified by a *proceed type*, which is represented as a symbol.

9.1 Protocol for Proceeding

For proceeding to work, two conceptual agents must agree:

- The programmer who wrote the program that signals the condition;
- The programmer who wrote the **condition-bind** handler that decided to proceed from the condition, or else the user who told the Debugger to proceed.

The signaller signals the condition and provides the various proceed types. The handler chooses from among the proceed types to make execution proceed.

Each agent has certain responsibilities to the other; each must follow the protocol described below to make sure that any handler interacts correctly with any signaller. The following description should be considered a two-part protocol that each agent must follow in order to communicate correctly with the other.

In very simple cases, the signaller can use **fsignal**, which does not require any new flavor definitions.

In all other cases, the signaller signals the condition using **signal** or **signal-proceed-case**. The signaller also defines a condition flavor with at least one method to handle a proceed type. The way to define a method that creates a new proceed type is somewhat unusual in that it uses a style of method combination called **:case** combination. Here's an example from the system:

```
(defmethod (sys:subscript-out-of-bounds :case :proceed :new-subscript)
           (&optional (sub (prompt-and-read ':number
                                            "Subscript to use instead: ")))
  "Supply a different subscript."
  (values ':new-subscript sub))
```

This code fragment creates a proceed type called **:new-subscript** for the condition flavor **sys:subscript-out-of-bounds**. New proceed types are always defined by adding a **:case :proceed** method handler to the condition flavor. The method must always return values rather than throwing.

In **:case** method combination, the first argument to the **:proceed** message is like a subsidiary message name, causing a further dispatch just as the original message name caused a primary dispatch. The method from the example is invoked whenever an object of this flavor receives a **:proceed** message like this:

```
(send obj ':proceed ':new-subscript new-sub)
```

The variables in the lambda list for the method come from the rest of the arguments of the **send**.

All of the arguments to a **:proceed** method must be optional arguments. The **:proceed** method should provide default values for all its arguments. One useful way of doing this is to prompt a user for the arguments using the **query-io** stream. The example uses **prompt-and-read**. If all the optional arguments were supplied, the **:proceed** method must not do any input or output using **query-io**.

This facility has been defined assuming that **condition-bind** handlers would supply all the arguments for the method themselves. The Debugger runs this method and does not supply arguments, relying on the method to prompt the user for the arguments.

As in the example, the method should have a documentation string as the first form in its body. The **:document-proceed-type** message to a proceedable condition object displays the string. This string is used by the Debugger as a prompt to describe the proceed type. For example, the subscript example might result in the following Debugger prompt:

```
s-A: Supply a different subscript
```

The string should be phrased as a one-line description of the effects of proceeding from the condition. It should not have any leading or trailing newlines. (You can use the messages that the Debugger prints out to describe the effects of the s- commands as models if you are interested in stylistic consistency.)

Sometimes a simple fixed string is not adequate. You can provide a piece of Lisp code to compute the documentation text at run time by providing your own method for **:case :document-proceed-type**. This method definition takes the following form:

```
(defmethod (condition-flavor :case :document-proceed-type proceed-type)
           (stream)
  body...)
```

The body of the method should print documentation for *proceed-type* of *condition-flavor* onto *stream*.

The body of the **:proceed** method can do anything it wants. In general, it tries to repair the state of things so that execution can proceed past the point at which the condition was signalled. It can have side-effects on the state of the environment, it can return values so that the function that called **signal** can try to fix things up, or it can do both. Its operation is invisible to the handler; the signaller is free to divide the work between the function that calls **signal** and the **:proceed** method as it sees fit. When the **:proceed** method returns, **signal** returns all of those values to its caller. That caller can examine them and take action accordingly.

The meaning of these returned values is strictly a matter of convention between the **:proceed** method and the function calling **signal**. It is completely internal to the signaller and invisible to the handler. By convention, the first value is often the name of a proceed type. See the section "Signallers".

:proceed can return nil

A **:proceed** method can return a first value of **nil** if it declines to proceed from the condition. If a **nil** returned by a **:proceed** method becomes the return value for a **condition-bind** handler, this signifies that the handler has declined to handle the condition, and the condition continues to be signalled. When the **:proceed** message was sent by the Debugger, the Debugger prints a message saying that the condition was not proceeded, and it returns to its command level. This might be used by an interactive **:proceed** method that gives the user the opportunity either to proceed or to abort; if the user aborts, the method returns **nil**. Returning **nil** from a **:proceed** method should not be used as a substitute for detecting earlier (such as when the condition object is created) that the proceed type is inappropriate for that condition.

9.2 Proceed Type Messages

By default, condition objects have to handle all proceed types defined for the condition flavor. Condition objects can be created that handle only some of the proceed types for their condition flavor. When the signaller creates the condition object (with **signal** or **make-condition**), it can use the **:proceed-types** init option to specify which proceed types the object accepts. The value of the init option is a list of keyword symbols naming the proceed types.

```
    (signal 'my-condition ':proceed-types '(:abc))
```

The **:proceed-types** message to a condition object returns a list of keywords for the proceed types that the object is prepared to handle. (See the method **(:method condition :proceed-types)**.)

The **:proceed-type-p** message examines the list of valid proceed types to see whether it contains a particular proceed type. (See the method **(:method condition :proceed-type-p)**.)

A condition flavor might also have an **:init** daemon that could modify its **dbg:proceed-types** instance variable.

9.3 Proceeding with condition-bind handlers

Suppose the handler is a **condition-bind** handler function. Just to review, when the condition is signalled, the handler function is called with one argument, the condition object. The handler function can throw to some tag, return **nil** to say that it doesn't want to handle the condition, or try to proceed the condition.

The handler must not attempt to proceed using an invalid proceed type. It must determine which proceed types are valid for any particular condition object. It must do this at run-time because condition objects can be created that do not handle all of the proceed types for their condition flavor. (See the init option **(:method condition :proceed-types)**.) In addition, condition objects created with **error** instead of **signal** do not have any proceed types. The handler can use the **:proceed-types** and **:proceed-type-p** messages to determine which proceed types are available.

To proceed from a condition, a handler function sends the condition object a **:proceed** message with one or more arguments. The first argument is the proceed type (a keyword symbol) and the rest are the arguments for that proceed type. All of the standard proceed types are documented with their condition flavors. Thus the programmer writing the handler function can determine the meanings of the arguments. The handler function must always pass all of the arguments, even though they are optional.

Sending the **:proceed** message should be the last thing the handler does. It should then return immediately, propagating the values from the **:proceed** method back to its caller. Determining the meaning of the returned values is the business of the signaller only; the handler should not look at or do anything with these values.

9.4 Proceed Type Names

Any symbol can be used as the name of a proceed type, although using keyword symbols is conventional. The symbols **:which-operations** and **:case-documentation** are not valid names for proceed types because they are treated specially by the **:case** flavor combination. Do not use either of these symbols as the name of a proceed type when you create a new condition flavor.

9.5 Signallers

Signallers can use the **signal-proceed-case** special form to signal a proceedable condition. **signal-proceed-case** assumes that the first value returned by every proceed type is the keyword symbol for that proceed type. This convention is not currently enforced.

9.6 Reference material

signal-proceed-case *Special Form*

signal-proceed-case signals a proceedable condition. It has a clause to handle each proceed type of the condition. It has a slightly more complicated syntax than most special forms: you provide some variables, some argument forms, and some clauses:

```
(signal-proceed-case ((var1 var2 ...) arg1 arg2 ...)
   (proceed-type-1 body1...)
   (proceed-type-2 body2...)
   ...)
```

The first thing this form does is to call **signal**, evaluating each *arg* form to pass as an argument to **signal**. In addition to the arguments you supply, **signal-proceed-case** also specifies the **:proceed-types** init option, which it builds based on the *proceed-type-i* clauses.

When **signal** returns, **signal-proceed-case** treats the first returned value as the symbol for a proceed type. It then picks a *proceed-type-i* clause to run, based on that value. It works in the style of **selectq**: each clause starts with a proceed type (a keyword symbol), or a list of proceed types, and the rest of the clause is a list of forms to be evaluated. **signal-proceed-case** returns the values produced by the last form.

var1, *var2*, and so on, are bound to successive values returned from **signal** for use in the body of the *proceed-type-i* clause selected.

One *proceed-type-i* can be **nil**. If **signal** returns **nil**, meaning that the condition was not handled, **signal-proceed-case** runs the **nil** clause if one exists, or simply returns **nil** itself if no **nil** clause exists. Unlike **selectq**, no otherwise clause is available for **signal-proceed-case**.

The value passed as the **:proceed-types** option to **signal** lists the various proceed types in the same order as the clauses, so that the Debugger displays them in that order to the user and the RESUME command runs the first one.

10. Issues for Interactive Use

10.1 Tracing Conditions

trace-conditions *Variable*

The value of this variable is a condition or a list of conditions. It can also be **t**, meaning all conditions, or **nil**, meaning none.

If any condition is signalled that is built on the specified flavor (or flavors), the Debugger immediately assumes control, before any handlers are searched or called.

If the user proceeds, by using RESUME, signalling continues as usual. This might in fact revert control to the Debugger again. This variable is provided for debugging purposes only. It lets you trace the signalling of any condition so that you can figure out what conditions are being signalled and by what function. You can set this variable to **error** to trace all error conditions, for example, or you can be more specific.

This variable replaces the **errset** variable from earlier releases.

10.2 Breakpoints

The functions **breakon** and **unbreakon** can be used to set breakpoints in a program. They use the encapsulation mechanism like **trace** and **advise** to force the function to signal a condition when it is called. See the section "Encapsulations".

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

With no arguments, **breakon** returns a list of all functions with breakpoints set by **breakon**.

breakon sets a breakpoint for the *function-spec*. Whenever *function-spec* is called, the condition **sys:call-trap** is signalled, and the Debugger assumes control. At this point, you can inspect the state of the Lisp environment and the stack. Proceeding from the condition then causes the program to continue to run.

The first argument can be any function spec, so that you can trace methods and other functions not named by symbols. See the section "Function Specs".

condition-form can be used for making a conditional breakpoint. *condition-form* should be a Lisp form. It is evaluated when the function is called. If it returns **nil**, the function call proceeds without signalling anything. *condition-form* arguments from multiple calls to **breakon** accumulate and are treated as an **or** condition. Thus, when any of the forms becomes true, the breakpoint "goes off". *condition-form* is evaluated in the dynamic environment of the function call. You can inspect the arguments of *function-spec* by looking at the variable **arglist**.

unbreakon &optional *function-spec condition-form* *Function*

Turns off a breakpoint set by **breakon**. If *function-spec* is not provided, all breakpoints set by **breakon** are turned off. If *condition-form* is provided, it turns off only that condition, leaving any others. If *condition-form* is not provided, the entire breakpoint is turned off for that function.

Calling a function for which a breakpoint is set signals a condition with the following message:

`Break on entry to function` *name*

It provides a **:no-action** proceed type, which allows the function entry to proceed. The "trap on exit" bit is set in the stack frame of the function call, so that when the function returns or is thrown through another condition is signalled. Similarly, the "Break on exit from marked frame" message and the **:no-action** proceed type are provided, allowing the function return to proceed.

10.3 Debugger Bug Reports

The `c-M` command in the Debugger sends a bug report, creating a new process and running the **bug** function in that process. By default, the first argument to **bug** is the symbol **lispm**, so that the report is sent to the **BUG-LISPM** mailing list. Also by default, the mail-sending text buffer initially contains a standard set of information dumped by the Debugger. You can control this behavior for your own condition flavors. You can control the mailing list to which the bug report is sent by defining your own primary method for the following message.

:bug-report-recipient-system *Message*

This message is sent by the `c-M` command in the Debugger to find the mailing list to which to send the bug report mail. The default method (the one in the **condition** flavor) returns **lispm**, and this is passed as the first argument to the **bug** function.

You can control the initial contents of the mail-sending buffer by altering the handling of the following message, either by providing your own primary method to replace the default message, or by defining a **:before** or **:after** daemon to add your own specialized information before or after the default text.

:bug-report-description *stream number* *Message*

This message is sent by the c-M command in the Debugger to print out the text that is the initial contents of the mail-sending buffer. The handler should simply print whatever information it considers appropriate onto *stream*. *number* is the numeric argument given to c-M. The Debugger interprets *number* as the number of frames from the backtrace to include in the initial mail buffer.

10.4 Debugger Special Commands

When the Debugger assumes control because an error condition was signalled and not handled, it offers the user various ways to proceed or to restart. Sometimes you want to offer the user other kinds of options. In the system, the most common example of this occurs when you forget to type a package prefix. It signals a **sys:unbound-symbol** error and offers to let you use the symbol from the right package instead. This is neither a proceed type nor a restart-handler; it is a Debugger special command.

You can add one or more special commands to any condition flavor. For any particular instance, you can control whether to offer the special command. For example, the package-guessing service is not offered unless some other symbol with the same print name exists in a different package. Special commands are called only by the Debugger; **condition-bind** handler functions never see them.

Special commands provide the same kind of functionality that a **condition-bind** handler does. There is no reason, for example, that the package-prefix service could not have been provided by **condition-bind**. It is only a matter of convenience to have it in a special command.

To add special commands to your condition flavor, you must mix in the flavor **dbg:special-commands-mixin**, which provides both the instance variable **dbg:special-commands** and several method combinations. Each special command to a particular flavor is identified by a keyword symbol, just the same way that proceed types are identified. You can then create handlers for any of the following messages:

:special-command *command-type* *Message*

:special-command is sent when the user invokes the special command. It uses **:case** method-combination and dispatches on the name of the special command. No arguments are passed. The syntax is:

```
(defmethod (my-flavor :case :special-command my-command-keyword) ()
  "documentation"
  body...)
```

Any communication with the user should take place over the **query-io** stream. The method can return **nil** to return control to the Debugger or it can return the same thing that any of the **:proceed** methods would have returned in order to proceed in that manner.

:document-special-command *command-type stream* *Message*

:document-special-command prints the documentation of *command-type* onto *stream*. If you don't handle this message explicitly, the default handler uses the documentation string from the **:special-command** method. You can, however, handle this message in order to print a prompt string that has to be computed at run-time. This is analogous to **:document-proceed-type**. The syntax is:

```
(defmethod (my-flavor :case :document-special-command my-command-keyword)
           (stream)
  body...)
```

:initialize-special-commands *Message*

The Debugger sends **:initialize-special-commands** after it prints the error message. The methods are combined with **:progn** combination, so that each one can do some initialization. In particular, the methods for this message can remove items from the list **dbg:special-commands** in order to decide not to offer these special commands.

10.5 Special Keys

The system normally handles the ABORT and SUSPEND keys so that ABORT aborts what you are doing and SUSPEND enters a breakpoint. Without a CONTROL modifier, a keystroke command takes effect only when the process reads the character from the keyboard; with the CONTROL modifier, a keystroke command takes effect immediately. The META modifier means "do it more strongly"; m-ABORT resets the process entirely, and m-SUSPEND enters the Debugger instead of entering a simple read-eval-print loop.

A complete and accurate description of what these keys do requires a discussion of conditions and the Debugger.

With no CONTROL modifier, ABORT and SUSPEND are detected when your process tries to do input from the keyboard (typically by doing an input stream operation such as **:tyi** on a window). Therefore, if your process is computing or waiting for something else, the effects of the keystrokes are deferred until your process tries to do input.

With a CONTROL modifier, ABORT and SUSPEND are intercepted immediately by the Keyboard Process, which sends your process an **:interrupt** message. Thus, it performs the specified function immediately, even if it is computing or waiting.

ABORT Prints the following string on the **terminal-io** stream, unless it suspects that output on that stream might not work.

```
[Abort]
```

It then signals **sys:abort**, which is a simple condition. Programs can set up bound handlers for **sys:abort**, although most do not. Many programs set up restart handlers for **sys:abort**; most interactive programs have such a handler in their command loops. Therefore, ABORT usually restarts your program at the innermost command loop. Inside the Debugger, ABORT has a special meaning.

m-ABORT	Prints the following string on the **terminal-io** stream, unless it suspects that output on that stream might not work. `[Abort all]` It then sends your process a **:reset** message, with the argument **:always**. This has nothing to do with condition signalling. It just resets the process completely, unwinding its entire stack. What the process does after that depends on what kind of process it is and how it was created: it might start over from its initial function, or it might disappear. See the document *Processes*.
SUSPEND	Calls the **break** function with the argument **break**. This has nothing to do with condition signalling. See the special form **break**.
m-SUSPEND	Causes the Debugger to assume control without signalling any condition. The Debugger normally expects to be invoked because of some condition object, though, which it needs to interact properly with proceeding and restarting. Therefore, a condition object of flavor **break** is created in order to give the Debugger something to work with. **break** is not an error flavor; it is built on **condition**. It has no proceed types, but RESUME in the Debugger causes the Debugger to return and the process to resume what it was doing.

Several techniques are available for overriding the standard operation of ABORT and SUSPEND when they are being used with modifier keys.

- For using these keys with the CONTROL modifier, use the asynchronous character facility. See the section "Flavors and Messages".
- Defining your own hook function and binding **tv:kbd-tyi-hook** to it also overrides the interception of these characters with no CONTROL modifier. See the section "Flavors and Messages".

At the Debugger command loop, ABORT is the same as the Debugger c-Z command. It throws directly to the innermost restart handler that is appropriate for either the current error or the **sys:abort** condition.

When the Debugger assumes control, it displays a list of commands appropriate to the current condition, along with key assignments for each. Proceed types come first, followed by special commands, followed by restart handlers. One alphabetic key with the SUPER modifier is assigned to each command on the list. In addition, ABORT is always assigned to the innermost restart handler that handles **sys:abort** or the condition that was signalled; RESUME is always assigned to the first proceed type in the **:proceed-types** list. See the section "Proceed Type Messages".

If RESUME is not otherwise used, it invokes the first error restart that does not handle **abort**. When you enter the Debugger with m-SUSPEND, RESUME resumes the process.

You can customize the Debugger, assigning certain keystrokes to certain proceed types or special commands, by setting these variables in your init file:

dbg:*proceed-type-special-keys* *Variable*

The value of this variable should be an alist associating proceed types with characters. When an error supplies any of these proceed types, the Debugger assigns that proceed type to the specified key. For example, this is the mechanism by which the **:store-new-value** proceed type is offered on the m-C keystroke.

dbg:*special-command-special-keys* *Variable*

The value of this variable should be an alist associating names of special commands with characters. When an error supplies any of these special commands, the Debugger assigns that special command to the specified key. For example, this is the mechanism by which the **:package-dwim** special command is offered on the c-sh-P keystroke.

11. Condition Flavors Reference

A condition object is an instance of any flavor built out of the **condition** flavor. An error object is an instance of any flavor built out of the **error** flavor. The **error** flavor is built out of the **dbg:debugger-condition** flavor, which is built out of the **condition** flavor. Thus, all error objects are also condition objects.

Every flavor of condition that is instantiated must handle the **:report** message. (Flavors that just define sets of conditions need not handle it). This message takes a stream as its argument and prints out a textual message describing the condition on that stream. The printed representation of a condition object is like the default printed representation of any instance when slashifying is turned on. However, when slashifying is turned off (by **princ** or the **~A format** directive), the printed representation of a condition object is its **:report** message. Example:

```
(condition-case (co)
    (open "f:>a>b.c")
  (fs:file-not-found
    (prin1 co)))     prints out  #<QFILE-NOT-FOUND 33712233>

(condition-case (co)
    (open "f:>a>b.c")
  (fs:file-not-found
    (princ co)))     prints out  The file was not found
                                 For F:>a>b.c
```

11.1 Messages and Init Options

These messages can be sent to any condition object. They are handled by the basic **condition** flavor, on which all condition objects are built. Some particular condition flavors handle other messages; those are documented along with the particular condition flavors in another section. See the section "Standard Conditions".

:document-proceed-type *proceed-type stream* of **condition** *Method*

Prints out a description of what it means to proceed, using the given *proceed-type*, from this condition, on *stream*. This is used mainly by the Debugger to create its prompt messages. Phrase such a message as an imperative sentence, without any leading or trailing **return** characters. This sentence is for the human users of the machine who read this when they have just been dumped unexpectedly into the Debugger. It should be composed so that it makes sense to a person to issue that sentence as a command to the system.

:proceed-type-p *proceed-type* of **condition** *Method*

Returns **t** if *proceed-type* is one of the valid proceed types of this condition object. Otherwise, returns **nil.**

:proceed-types of **condition** *Method*

Returns a list of all the valid proceed types for this condition.

:set-proceed-types *new-proceed-types* of **condition** *Method*

Sets the list of valid proceed types for this condition to *new-proceed-types*.

:proceed-types *proceed-types* (for **condition**) *Init Option*

Defines the set of proceed types to be handled by this instance. *proceed-types* is a list of proceed types (symbols); it must be a subset of the set of proceed types understood by this flavor. If this option is omitted, the instance is able to handle all of the proceed types understood by this flavor in general, but by passing this option explicitly, a subset of acceptable proceed types can be established. This is used by **signal-proceed-case**.

If only one way to proceed exists, *proceed-types* can be a single symbol instead of a list.

If you pass a symbol that is not an understood proceed type, it is ignored. It does not signal an error because the proceed type might become understood later when a new **defmethod** is evaluated; if not, the problem is caught later.

The order in which the proceed types occur in the list controls the order in which the Debugger displays them in its list. Sometimes you might want to select an order that makes more sense for the user, although usually this is not important. The most important thing is that the RESUME command in the Debugger is assigned to the first proceed type in the list.

:special-commands of **condition** *Method*

Returns a list of all Debugger special commands for this condition. See the section "Debugger Special Commands".

:special-command-p *command-type* of **condition** *Method*

Returns **t** if *command-type* is a valid Debugger special command for this condition object; otherwise, returns **nil.**

:report *stream* of **condition** *Method*

Prints the text message associated with this object onto *stream*. The **condition** flavor does not support this itself, but it is a required message, and any flavor built on **condition** that is instantiated must support this message.

:report-string of **condition** *Method*

Returns a string containing the report message associated with this object. It works by sending **:report** to the object.

11.2 Standard Conditions

This section presents the standard condition flavors provided by the system. Some of these flavors are the flavors of actual condition objects that get instantiated in response to certain conditions. Others never actually get instantiated, but are used to build other flavors.

In some cases, the flavor that the system uses to signal an error is not exactly the one listed here, but rather a flavor built on the one listed here. This often comes up when the same error can be reported by different programs that implement a generic protocol. For example, the condition signalled by a remote file-system stream when a file is not found is different from the one signalled by a local file-system stream; however, only the generic **fs:file-not-found** condition should ever be handled by programs, so that a program works regardless of what kind of file-system stream it is using. The exact flavors signalled by each file system are considered to be internal system names, subject to change without notice and not documented herein.

Do not look at system source code to figure out the names of error flavors without being careful to choose the right level of flavor! Furthermore, take care to choose a flavor that can be instantiated if you try to signal a system-defined condition. For example, you can not signal a condition object of type **fs:file-not-found** because this is really a set of errors and this flavor does not handle the **:report** message. If you were to implement your own file system and wanted to signal an error when a file cannot be found, it should probably have its own flavor built on **fs:file-not-found** and other flavors.

Choosing the appropriate condition to handle is a difficult problem. In general you do not want to choose a condition on the basis of the apparent semantics of its name. Rather you should choose it according to the appropriate level of the condition flavor hierarchy. This holds particularly for file-related errors. See the section "File-system Errors".

11.2.1 Fundamental Conditions

These conditions are basic to the functionality of the condition mechanism, rather than having anything to do with particular system errors.

condition *Flavor*

This is the basic flavor on which all condition objects are built.

dbg:debugger-condition *Flavor*

This flavor is built on **condition**. It is used for entering the Debugger without necessarily classifying the event as an error. This is intended primarily for system use; users should normally build on **error** instead.

error *Flavor*

This flavor is built on **dbg:debugger-condition**. All flavors that represent errors, as opposed to debugger conditions or simple conditions, are built on this flavor.

ferror *Flavor*

This is a simple error flavor for the **ferror** function. Use it when you do not want to invent a new error flavor for a certain condition. Its only state information is an error message, normally created by the call to the **ferror** function. It has two gettable and inittable instance variables **format-string** and **format-args**. The **format** function is applied to these values to produce the **:report** message.

dbg:proceedable-ferror *Flavor*

This is a simple error flavor for the **fsignal** function. Use it when you do not want to invent a new error flavor for a certain condition, but you want the condition to be proceedable. Its only state information is an error message, created by the call to the **fsignal** function. Its only proceed type is **:no-action**. Proceeding in this way does nothing and causes **fsignal** (or **signal**) to return the symbol **:no-action**.

sys:no-action-mixin *Flavor*

This flavor can be mixed into any condition flavor to define a proceed type called **:no-action**. Proceeding in this way causes the computation to proceed as if no error check had occurred. The signaller might try the action again or might simply go on doing what it would have done. For example, **proceedable-ferror** is just **ferror** with this mixin.

sys:abort *Flavor*

The ABORT key on the keyboard was pressed. This is a simple condition. When **sys:abort** is signalled, control is thrown straight to a restart handler without entering the Debugger. See the section "Special Keys".

break *Flavor*

This is the flavor of the condition object passed to the Debugger as a result of the m-BREAK command. It is never actually signalled; rather, it is a convention to ensure that the Debugger always has a condition when it assumes control. This is based on **dbg:debugger-condition**. See the section "Special Keys".

11.2.2 Lisp Errors

This section describes the conditions signalled for basic Lisp errors. All of the conditions in this section are based on the **error** flavor unless otherwise indicated.

11.2.2.1 Base flavor: sys:cell-contents-error

sys:cell-contents-error *Flavor*

All of the kinds of errors resulting from finding invalid contents in a cell of virtual memory are built on this flavor. This represents a set of errors including the various kinds of unbound-variable errors, the undefined-function error, and the bad data-type error.

Proceed type	*Action*
:new-value	Takes one argument, a new value to be used instead of the contents of the cell.
:store-new-value	Takes one argument, a new value to replace the contents of the cell.
:no-action	If you have intervened and stored something into the cell, the contents of the cell can be reread.

sys:unbound-variable *Flavor*

All of the kinds of errors resulting from unbound variables are built on this flavor. Because these are a subset of the "cell contents" errors, this flavor is built on **sys:cell-contents-error**. The **:variable-name** message returns the name of the variable that was unbound (a symbol).

sys:unbound-symbol *Flavor*

An unbound symbol (special variable) was evaluated. Some instances of this flavor provide the **:package-dwim** special command, which takes no arguments and asks whether you want to examine the value of various other symbols with the same print name in other packages. This proceed type is provided only if any such symbols exist in any other packages. (See also **dbg:*defer-package-dwim*.**) This flavor is built on **sys:unbound-variable.** The proceed types from **sys:cell-contents-error** are provided, as is the **:variable-name** message from **sys:unbound-variable.**

sys:unbound-closure-variable *Flavor*

An unbound closure variable was evaluated. This flavor is built on **sys:unbound-variable.** The proceed types from **cell-contents-error** are provided, as is the **:variable-name** message from **sys:unbound-variable.**

sys:unbound-instance-variable *Flavor*

An unbound instance variable was evaluated. The **:instance** message returns the instance in which the unbound variable was found. The proceed types from **cell-contents-error** are provided, as is the **:variable-name** message from **sys:unbound-variable.**

sys:undefined-function *Flavor*

An undefined function was invoked; that is, an unbound function cell was referenced. This flavor is built on **sys:cell-contents-error** and provides all of its proceed types. The **:function-name** message returns the name of the function that was undefined (a function spec). This also provides **:package-dwim** service, like **sys:unbound-symbol**.

sys:bad-data-type-in-memory *Flavor*

A word with an invalid type code was read from memory. This flavor is built on **sys:cell-contents-error** and provides all of its proceed types.

Message	*Value returned*
:address	virtual address, as a locative pointer, from which the word was read
:data-type	numeric value of the data-type tag field of the word
:pointer	numeric value of the pointer field of the word

11.2.2.2 Location Errors

sys:unknown-setf-reference *Flavor*

setf did not find a **setf** property on the car of the form. The **:form** message returns the form that **setf** tried to operate on. This error is signalled when the **setf** macro is expanded.

sys:unknown-locf-reference *Flavor*

locf did not find a **locf** property on the car of the form. The **:form** message returns the form that **locf** tried to operate on. This error is signalled when the **locf** macro is expanded.

11.2.2.3 Base flavor: sys:arithmetic-error

sys:arithmetic-error *Flavor*

Represents the set of all arithmetic errors. No condition objects of this flavor are actually created; any arithmetic error signals a more specific condition, built on this one. This flavor is provided to make it easy to handle any arithmetic error.

All arithmetic errors handle the **:operands** message. On the 3600, this returns a list of the operands in the operation that caused the error. On the LM-2, this message almost always returns **nil**.

sys:divide-by-zero *Flavor*

Division by zero was attempted. This flavor is built on **sys:arithmetic-error**. The **:function** message returns the function that did the division.

sys:non-positive-log *Flavor*

Computation of the logarithm of a nonpositive number was attempted. This flavor is built on **sys:arithmetic-error**. The **:number** message returns the nonpositive number.

math:singular-matrix *Flavor*

A singular matrix was given to a matrix operation such as inversion, taking of the determinant, or computation of the LU decomposition. This flavor is built on **sys:arithmetic-error**.

11.2.2.4 Base flavor: sys:floating-point-exception

sys:floating-point-exception and the condition flavors based on it are designed to support IEEE floating-point standards. See the section "Numbers". By default, all IEEE traps are enabled, except for the inexact-result trap. Future releases will provide control over the floating-point operating mode, including rounding mode and enabling and disabling of traps.

The trap handlers that signal these conditions from the system all cause pressing the RESUME key to mean "return the result that would have been returned if the trap had been disabled". For example, pressing RESUME on an overflow returns the appropriately signed infinity as the result. On an underflow it returns the denormalized (possibly zero) result.

sys:floating-point-exception *Flavor*

This is the base flavor for floating-point exceptional conditions. No condition objects of this flavor are actually created. This flavor is provided to make it easy to handle any floating-point exception. It is built on **sys:arithmetic-error**.

Message	*Value returned*
:operation	A symbol indicating the operation that caused the exception.
:operands	The list of operands to the operation.
:non-trap-result	The result that would have been returned if this trap had been disabled.
:saved-float-operation-status	The value of **sys:float-operation-status** at the time of the exception.

Proceed type	*Action*
:new-value	Takes one argument and uses this value as the result of the operation.

sys:float-divide-by-zero *Flavor*

A floating-point division by zero was attempted. This flavor is built on **sys:divide-by-zero** and **sys:floating-point-exception**.

sys:floating-exponent-overflow *Flavor*

Overflow of an exponent occurred during floating-point arithmetic. This flavor is built on **sys:floating-point-exception**. The **:function** message returns the function that got the overflow, if it is known, and **nil** if it is not known. On the LM-2, the **:small-float-p** message returns **t** if "small" flonums were involved. The **:new-value** proceed type is provided with one argument, a floating-point number to use instead.

sys:floating-exponent-underflow *Flavor*

Underflow of an exponent occurred during floating-point arithmetic. This flavor is built on **sys:floating-point-exception**. The **:function** message returns the function that got the underflow, if it is known, and **nil** if it is not known. On the LM-2, the **:small-float-p** message returns **t** if "small" flonums were involved. The **:use-zero** proceed type is provided with no arguments; a **0.0** (or possibly **0.0s0** on the LM-2) is used instead.

sys:float-inexact-result *Flavor*

A floating-point result does not exactly represent the operation's result, due to the fixed precision of floating-point representation. Since most floating-point calculations are inexact, the inexact-result trap is disabled by default. This flavor is built on **sys:floating-point-exception**.

sys:float-invalid-operation *Flavor*

An invalid floating-point operation was attempted, such as dividing infinity by infinity. This flavor is built on **sys:floating-point-exception**.

sys:negative-sqrt *Flavor*

Computing the square root of a negative number was attempted. This flavor is built on **sys:float-invalid-operation**.

sys:float-divide-zero-by-zero *Flavor*

A floating-point division of zero by zero was attempted. This flavor is built on **sys:float-invalid-operation** and **sys:float-divide-by-zero**. Most programs handle not this condition itself, but rather one of the component condition flavors.

11.2.2.5 Miscellaneous System Errors Not Categorized by Base Flavor

sys:end-of-file *Flavor*

A function doing input from a stream attempted to read past the end-of-file. The **:stream** message returns the stream.

sys:wrong-stack-group-state *Flavor*

A stack group was in the wrong state to be resumed. The **:stack-group** message returns the stack group.

sys:draw-off-end-of-screen *Flavor*

Drawing graphics past the edge of the screen was attempted.

sys:draw-on-unprepared-sheet *Flavor*

A drawing primitive (such as **tv:%draw-line**) was used on a screen array not inside a **tv:prepare-sheet** special form. The **:sheet** message returns the sheet (window) that should have been prepared.

sys:bitblt-destination-too-small *Flavor*

The destination array of a **bitblt** was too small.

sys:bitblt-array-fractional-word-width *Flavor*

An array passed to **bitblt** does not have a first dimension that is a multiple of 32 bits. The **:array** message returns the array.

sys:write-in-read-only *Flavor*

Writing into a read-only portion of memory was attempted. The **:address** message returns the address at which the write was attempted.

sys:pdl-overflow *Flavor*

A stack (pdl) overflowed. The **:pdl-name** message returns the name of the stack (a string, such as "regular" or "special"). The **:grow-pdl** proceed type is provided, with no arguments; it increases the size of the stack. This is based on **dbg:debugger-condition**, not on **error**.

sys:area-overflow *Flavor*

This is signalled when the maximum-size (**:size** argument to **make-area**) is exceeded.

sys:virtual-memory-overflow *Flavor*

This is an irrecoverable error that is signalled when you run out of virtual memory.

sys:region-table-overflow *Flavor*

This is an irrecoverable error that is signalled when you run out of regions.

sys:cons-in-fixed-area *Flavor*

Allocation of storage from a fixed area of memory was attempted.

Message	*Value returned*
:area	name of the area
:region	region number

sys:throw-tag-not-seen *Flavor*

***throw** or **throw** was called, but no matching ***catch** or **catch** was found.

Message	*Value returned*
:tag	Catch-tag that was being thrown to.
:values	List of the values that were being thrown. If ***throw** was called, this is always a list of two elements, the value being thrown and the tag; if the new **throw** special form of Common Lisp (currently implemented only on the 3600) is used, the list may be of any length.

The **:new-tag** proceed type is provided with one argument, a new tag (a symbol) to try instead of the original.

sys:instance-variable-zero-referenced *Flavor*

Referencing instance variable 0 of an instance was attempted. This usually means that some method is referring to an instance variable that was deleted by a later evaluation of a **defflavor** form.

sys:instance-variable-pointer-out-of-range *Flavor*

Referencing an instance variable that does not exist was attempted. This usually means that some method is using an obsolete instance because a **defflavor** form got evaluated again and changed the flavor incompatibly.

sys:disk-error *Flavor*

An error was reported by the disk software or controller. The **:retry-disk-operation** proceed type is provided; it takes no arguments.

sys:redefinition *Flavor*

This is a simple condition rather than an error condition. It signals an attempt to redefine something by some other file than the one that originally defined it. The **:definition-type** argument specifies the kind of definition: it might be **defun** if the function cell is being defined, **defstruct** if a structure is being defined, and so on.

Message	*Value returned*
:name	symbol (or function spec) being redefined
:old-pathname	pathname that originally defined it
:new-pathname	pathname that is now trying to define it

Either pathname will be **nil** if the definition was from inside the Lisp environment rather than from loading a file.

The following proceed types are provided:

Message	*Action*
:proceed	Redefinition should go ahead; in the future no warnings should be signalled for this pair of pathnames.
:inhibit-definition	Definition is not changed and execution proceeds.
:no-action	Function should be redefined as if no warning had occurred.

Note: if this condition is not handled, the action is controlled by the value of **fs:inhibit-fdefine-warnings**.

11.2.2.6 Function-calling Errors

sys:zero-args-to-select-method *Flavor*

A select method was applied to no arguments. The **:select-method** message returns the select method. This applies only to the LM-2.

sys:too-few-arguments *Flavor*

A function was called with too few arguments.

Message	*Value returned*
:function	the function
:nargs	number of arguments supplied
:argument-list	list of the arguments passed

The **:additional-arguments** proceed type is provided with one argument, a list of additional argument values to which the function should be applied. If the error is proceeded, these new arguments are appended to the old arguments and the function is called with this new argument list.

sys:too-many-arguments *Flavor*

A function was called with too many arguments.

Message	*Value returned*
:function	the function
:nargs	number of arguments supplied
:argument-list	list of the arguments passed

The **:fewer-arguments** proceed type is provided with one argument, the new number of arguments with which the function should be called. In proceeding from this error, the function is called with the first *n* arguments only, where *n* is the number specified.

sys:wrong-type-argument *Flavor*

A function was called with at least one argument of invalid type.

Message	*Value returned*
:function	function with invalid argument(s)
:old-value	invalid value
:description	description of valid value
:arg-name	name of the argument
:arg-number	number of the argument (the first argument to a function is **0**, and so on) or **nil** if this does not apply

:description, **:arg-name**, and **:arg-number** are valid messages only when the error was signalled by **check-arg**, **check-arg-type**, or **argument-typecase**. Check to be sure that the message is valid before sending it (remember **:operation-handled-p**).

Proceed type	*Action*
:argument-value	Takes one argument, the new value to use for the argument.
:store-argument-value	Takes one argument, the new value to use and to store back into the local variable in which it was found. (Currently valid only on the 3600.)

11.2.2.7 Array Errors

dbg:bad-array-mixin *Flavor*

Errors involving an array that seems to be the wrong object include this flavor. It provides the **:array** message, which returns the array.

Proceed type	*Action*
:new-array	Takes one argument, an array to use instead of the old one.
:store-new-array	Takes one argument, an array to use instead of the old one and to store back into the local variable in which it was found. (Currently valid only on the 3600.)

sys:bad-array-type *Flavor*

A meaningless array type code was found in virtual memory, indicating a system bug. The **:type** message returns the numeric type code.

sys:array-has-no-leader *Flavor*

Using the leader of an array that has no array leader was attempted. The **:array** message returns the array. This includes the **bad-array-mixin** flavor.

sys:fill-pointer-not-fixnum *Flavor*

The fill pointer of an array was not a fixnum. The **:array** message returns the array. This includes the **bad-array-mixin** flavor.

sys:array-wrong-number-of-dimensions *Flavor*

The wrong number of subscripts was presented to an array.

Message	*Value returned*
:dimensions-given	number of subscripts presented
:dimensions-expected	number that should have been given
:array	the array

This includes the **bad-array-mixin** flavor.

sys:number-array-not-allowed *Flavor*

A number array (such as an **art-4b** or **art-16b**) was used in a context in which number arrays are not valid, such as an attempt to make a pointer to an element with **aloc** or **locf**. This includes the **bad-array-mixin** flavor.

sys:subscript-out-of-bounds *Flavor*

An attempt was made to reference an array using out-of-bounds subscripts, an out-of-bounds array leader element, or an out-of-bounds instance variable.

Message	*Value returned*
:object	the object (an array or instance) if it is known, and **nil** otherwise
:function	function that did the reference, or **nil** if it is not known
:subscript-used	the subscript that was actually used
:subscript-limit	the limit that it passed

The messages to access the subscripts and limits are complex due to differences between the **LM-2** and the 3600. (The **LM-2** can report only the computed product, not the individual subscripts of a reference to a multidimensional array; the 3600 can report the individual subscripts.) These values are fixnums; if a multidimensional array was used, they are computed products. The **:subscripts-used** and **:subscripts-limit** messages always return lists of the values.

Proceed type	*Action*
:new-subscript	Takes an arbitrary number of arguments, the new subscripts for the array reference.
:store-new-subscript	Takes the same arguments as **:new-subscript** and stores them back into the local variables in which they were found. (Currently only on the 3600.)

11.2.2.8 Eval Errors

sys:invalid-form *Flavor*

The evaluator attempted to evaluate an invalid form. The **:form** message returns the form.

sys:invalid-function *Flavor*

The evaluator attempted to apply an object that is not a function or a symbol whose definition is an object that is not a function. The **:function** message returns the object that was applied. The **:new-function** proceed type is provided, with one argument: a new function to be used.

sys:invalid-lambda-list *Flavor*

The evaluator attempted to apply a function with an invalid lambda list. This is built on **sys:invalid-function.** The **:function** message and the **:new-function** proceed type are provided.

sys:undefined-keyword-argument *Flavor*

The evaluator attempted to pass a keyword to a function that does not recognize that keyword.

Message	*Value returned*
:keyword	Unrecognized keyword
:value	The value passed with it

Proceed type	*Action*
:no-action	The keyword and its value are ignored.
:new-keyword	Specifies a new keyword to use instead. Its one argument is the new keyword.

sys:funcall-macro *Flavor*

The evaluator attempted to apply a symbol whose definition is a macro as if it were a function. The **:eval-macro-funcall** proceed type is defined with no arguments. It proceeds by building a form in which the macro is given these arguments and evaluating that form.

sys:unclaimed-message *Flavor*

This flavor is built on **error**. The flavor system signals this error when it finds a message for which no method is available.

Message	*Value returned*
:object	the object
:message	the message-name
:arguments	the arguments of the message[1]

The object can be an instance or a select method.

11.2.2.9 Interning errors based on sys:package-error

sys:package-error *Flavor*

All package-related error conditions are built on **sys:package-error**.

sys:package-not-found *Flavor*

A package-name lookup did not find any package by the specified name.

The **:name** message returns the name. The **:relative-to** message returns **nil** if only absolute names are being searched, or else the package whose relative names are also searched.

The **:no-action** proceed type may be used to try again. The **:new-name** proceed type may be used to specify a different name or package. The **:create-package** proceed type creates the package with default characteristics.

[1]The **:arguments** message on the LM-2 for a select-method always returns **nil**, because the information is lost.

sys:package-locked *Flavor*

There was an attempt to intern a symbol in a locked package.

The **:symbol** message returns the symbol. The **:package** message returns the package.

The **:no-action** proceed type interns the symbol just as if the package had not been locked. Other proceed types are also available when interning the symbol would cause a name conflict.

11.2.2.10 Errors Involving Lisp Printed Representations

si:*print-object-error-message* *Variable*

Controls what happens when errors are signalled inside the Lisp printer. When **nil** (the default), the error is not handled. Otherwise, the value should be a string. If an error is signalled during the printing of an object, that string is sent to the stream instead of the printed representation of the object, and the printing function immediately returns to its caller. This applies to all functions that are entries to the Lisp printer, including **print**, **princ**, and **prin1**.

Example:

```
(let ((si:*print-object-error-message* "[Error printing object]"))
  (format t "foo: ~S, bar: ~S" foo bar))
```

This is useful because **bar** is printed even if the printing of **foo** causes an error.

sys:print-not-readable *Flavor*

The Lisp printer encountered an object that it cannot print in a way that the Lisp reader can understand. The printer signals this condition only if **si:print-readably** is not **nil** (it is normally **nil**). The **:object** message returns the object. The **:no-action** proceed type is provided; proceeding this way causes the object to be printed as if **si:print-readably** were **nil**.

sys:read-error *Flavor*

This flavor, built on **sys:parse-error**, includes errors encountered by the Lisp reader.

sys:read-end-of-file *Flavor*

The Lisp reader encountered an end-of-file while in the middle of a string or list. This flavor is built on **sys:read-error** and **sys:end-of-file**.

sys:read-list-end-of-file *Flavor*

The Lisp reader attempted to read past the end-of-file while it was in the middle of reading a list. This is built on **sys:read-end-of-file**. The **:list** message returns the list that was being built.

sys:read-string-end-of-file *Flavor*

The Lisp reader attempted to read past the end-of-file while it was in the middle of reading a string. This is built on **sys:read-end-of-file**. The **:string** message returns the string that was being built.

11.2.3 File-system Errors

The following condition flavors are part of the Lisp Machine's generic file system interface. These flavors work for all file systems, whether local Lisp Machine file systems, remote Lisp Machine file systems (accessed over a network), or remote file systems of other kinds, such as UNIX or TOPS-20. All of them report errors uniformly.

Some of these condition flavors describe situations that can occur during any file system operation. These include not only the most basic flavors, such as **fs:file-request-failure** and **fs:data-error**, but also flavors such as **fs:file-not-found** and **fs:directory-not-found**. Other file system condition flavors describe failures related to specific file system operations, such as **fs:rename-failure**, and **fs:delete-failure**. Given all these choices, you have to determine what condition is appropriate to handle, for example in checking for success of a rename operation. Would **fs:rename-failure** include cases where, say, the directory of the file being renamed is not found?

The answer to this question is that you should handle **fs:file-operation-failure**. **fs:rename-failure** and all other conditions at that level are signalled only for errors that relate specifically to the semantics of the operation involved. If you cannot delete a file because the file is not found, **fs:file-not-found** would be signalled. Suppose you cannot delete the file because its "don't delete switch" is set, which is an error relating specifically to deletion. **fs:delete-failure** would be signalled. Therefore, since you cannot know whether a condition flavor related to an operation requested or some more general error will be signalled, you will usually want to handle one of the most general flavors of file system error.

Under normal conditions, you would bind only for **fs:file-request-failure** or **fs:file-operation-failure** rather than for the more specific condition flavors described in this section. Some guidelines for using the different classes of errors:

error	Any error at all. It is not wise in general to attempt to handle this, because it catches program and operating system bugs as well as file-related bugs, thus "hiding" knowledge of the system problems from you.
fs:file-error	Any file related error at all. This includes **fs:file-operation-failure** as well as **fs:file-request-failure**. Condition objects of flavor **fs:file-request-failure** usually indicate that the file system, host operating system, or network did not operate properly. If your program is attempting to handle file-

related errors, it should not handle these: it is usually better to allow the program to enter the debugger. Thus it is very rare that one would want to handle **fs:file-error**.

fs:file-operation-failure	This includes almost all predictable file-related errors, whether they are related to the semantics of a specific operation, or are capable of occurring during many kinds of operations. Therefore, **fs:file-operation-failure** is usually the appropriate condition to handle.
Specific conditions	It is appropriate and correct to handle specific conditions, like **fs:delete-failure**, if your program assigns specific meaning to (or has specific actions associated with) specific occurrences, such as a nonexistent directory or an attempt to delete a protected file. If you do not "care" about specific conditions, but you wish to handle predictable file-related errors, you should handle **fs:file-operation-failure**. You should *not* attempt to handle, say, **fs:delete-failure** to test for any error occurring during deletion; it does not mean that.

fs:file-error *Flavor*

This set includes errors encountered during file operations. This flavor is built on **error**.

Message	*Value returned*
:pathname	pathname that was being operated on or **nil**
:operation	name of the operation that was being done: this is a keyword symbol such as **:open**, **:close**, **:delete**, or **:change-properties**, and it might be **nil** if the signaller does not know what the operation was or if no specific operation was in progress

In a few cases, the **:retry-file-error** proceed type is provided, with no arguments; it retries the file system request. All flavors in this section accept these messages and might provide this proceed type.

fs:file-request-failure *Flavor*

This set includes all file-system errors in which the request did not manage to get to the file system.

fs:file-operation-failure *Flavor*

This set includes all file-system errors in which the request was delivered to the file system, and the file system decided that it was an error.

Note: every file-system error is either a request failure or an operation failure, and the rules given above explain the distinction. However, these rules are slightly unclear in some cases. If you want to be sure whether a certain error is a request failure or an operation failure, consult the detailed descriptions in the rest of this section.

11.2.3.1 Request failures based on fs:file-request-failure

fs:data-error *Flavor*

Bad data are in the file system. This might mean data errors detected by hardware or inconsistent data inside the file system. This flavor is built on **fs:file-request-failure**. The **:retry-file-operation** proceed type from **fs:file-error** is provided in some cases; send a **:proceed-types** message to find out.

fs:host-not-available *Flavor*

The file server or file system is intentionally denying service to users. This does *not* mean that the network connection failed; it means that the file system explicitly does not care to be available. This flavor is built on **fs:file-request-failure**.

fs:no-file-system *Flavor*

The file system is not available. For example, this host does not have any file system, or this host's file system cannot be initialized for some reason. This flavor is built on **fs:file-request-failure**.

fs:network-lossage *Flavor*

The file server had some sort of trouble trying to create a new data connection and was unable to do so. This flavor is built on **fs:file-request-failure**.

fs:not-enough-resources *Flavor*

Some resource was not available in sufficient supply. Retrying the operation might work if you wait for some other users to go away or if you close some of your own files. This flavor is built on **fs:file-request-failure**.

fs:unknown-operation *Flavor*

An unsupported file-system operation was attempted. This flavor is built on **fs:file-request-failure**.

11.2.3.2 Login Problems

Some login problems are correctable and some are not. To handle any correctable login problem, you set up a handler for **fs:login-required** rather than handling the individual conditions.

The correctable login problem conditions work in a special way. The Lisp Machine's generic file system interface, in the user-end of the remote file protocol, always handles these errors with its own condition handler; it then signals the **fs:login-required** condition. Therefore to handle one of these problems, you set up a handler for **fs:login-required**. The condition object for the correctable login problem can be obtained from the condition object for **fs:login-required** by sending it an **:original-condition** message.

fs:login-problems *Flavor*

This set includes all problems encountered while trying to log in to the file system. Currently, none of these ever happen when you use a local file system. This flavor is built on **fs:file-request-failure**.

fs:correctable-login-problems *Flavor*

This set includes all correctable problems encountered while trying to log in to the foreign host. None of these ever happen when you use a local file system. This flavor is built on **fs:login-problems**.

fs:unknown-user *Flavor*

The specified user name is unknown at this host. The **:user-id** message returns the user name that was used. This flavor is built on **fs:correctable-login-problems**.

fs:invalid-password *Flavor*

The specified password was invalid. This flavor is built on **fs:correctable-login-problems**.

fs:not-logged-in *Flavor*

A file operation was attempted before logging in. Normally the file system interface always logs in before doing any operation, but this problem can come up in certain unusual cases in which logging in has been aborted. This flavor is built on **fs:correctable-login-problems**.

fs:login-required *Flavor*

This is a simple condition built on **condition**. It is signalled by the file-system interface whenever one of the correctable login problems happens.

Message	*Value returned*
(send (send error :access-path) :host)	the foreign host
:host-user-id	user name that would be the default for this host
:original-condition	condition object of the correctable login problem

The **:password** proceed type is provided with two arguments, a new user name and a new password, both of which should be strings. If the condition is not handled by any handler, the file system prompts the user for a new user name and password, using the **query-io** stream.

11.2.3.3 File Lookup

fs:file-lookup-error *Flavor*

This set includes all file-name lookup errors. This flavor is built on **fs:file-operation-failure**.

fs:file-not-found *Flavor*

The file was not found in the containing directory. The TOPS-20 and TENEX "no such file type" and "no such file version" errors also signal this condition. This flavor is built on **fs:file-lookup-error**.

fs:multiple-file-not-found *Flavor*

None of a number of possible files was found. This flavor is built on **fs:file-lookup-error**. It is signalled when **load** is not given a specific file type but cannot find either a source or a binary file to load.

The flavor allows three init keywords of its own. These are also the names of messages that return the following:

:operation	The operation that failed
:pathname	The pathname given to the operation
:pathnames	A list of pathnames that were sought unsuccessfully

The condition has a **:new-pathname** proceed type to prompt for a new pathname.

fs:directory-not-found *Flavor*

The directory of the file was not found or does not exist. This means that the containing directory was not found. If you are trying to open a directory, and the actual directory you are trying to open is not found, **fs:file-not-found** is signalled. This flavor is built on **fs:file-lookup-error**.

This flavor has two Debugger special commands: **:create-directory**, to create only the lowest level of directory, and **:create-directories-recursively**, to create any missing superiors as well.

fs:device-not-found *Flavor*

The device of the file was not found or does not exist. This flavor is built on **fs:file-lookup-error**.

fs:link-target-not-found *Flavor*

The target of the link that was opened did not exist. This flavor is built on **fs:file-lookup-error**.

11.2.3.4 fs:access-error

fs:access-error *Flavor*

This set includes all protection-violation errors. This flavor is built on **fs:file-operation-failure**.

fs:incorrect-access-to-file *Flavor*

Incorrect access to the file in the directory was attempted. This flavor is built on **fs:access-error**.

fs:incorrect-access-to-directory *Flavor*
Incorrect access to some directory containing the file was attempted. This flavor is built on **fs:access-error**.

11.2.3.5 fs:invalid-pathname-syntax

fs:invalid-pathname-syntax *Flavor*
This set includes all invalid pathname syntax errors. This is not the same as **fs:parse-pathname-error**. (See the flavor **fs:parse-pathname-error**.) These errors occur when a successfully parsed pathname object is given to the file system, but something is wrong with it. See the specific conditions that follow. This flavor is built on **fs:file-operation-failure**.

fs:invalid-wildcard *Flavor*
The pathname is not a valid wildcard pathname. This flavor is built on **fs:invalid-pathname-syntax**.

fs:wildcard-not-allowed *Flavor*
A wildcard pathname was presented in a context that does not allow wildcards. This flavor is built on **fs:invalid-pathname-syntax**.

11.2.3.6 fs:wrong-kind-of-file

fs:wrong-kind-of-file *Flavor*
This set includes errors in which an invalid operation for a file, directory, or link was attempted.

fs:invalid-operation-for-link *Flavor*
The specified operation is invalid for links, and this pathname is the name of a link. This flavor is built on **fs:wrong-kind-of-file**.

fs:invalid-operation-for-directory *Flavor*
The specified operation is invalid for directories, and this pathname is the name of a directory. This flavor is built on **fs:wrong-kind-of-file**.

11.2.3.7 fs:creation-failure

fs:creation-failure *Flavor*
This set includes errors related to attempts to create a file, directory, or link. This flavor is built on **fs:file-operation-failure**.

fs:file-already-exists *Flavor*
A file of this name already exists. This flavor is built on **fs:creation-failure**.

fs:create-directory-failure *Flavor*
This set includes errors related to attempts to create a directory. This flavor is built on **fs:creation-failure**.

fs:directory-already-exists *Flavor*

A directory or file of this name already exists. This flavor is built on **fs:creation-directory-failure**.

fs:create-link-failure *Flavor*

This set includes errors related to attempts to create a link. This flavor is built on **fs:creation-failure**.

11.2.3.8 fs:rename-failure

fs:rename-failure *Flavor*

This set includes errors related to attempts to rename a file. The **:new-pathname** message returns the target pathname of the rename operation. This flavor is built on **fs:file-operation-failure**.

fs:rename-to-existing-file *Flavor*

The target name of a rename operation is the name of a file that already exists. This flavor is built on **fs:rename-failure**.

fs:rename-across-directories *Flavor*

The devices or directories of the initial and target pathnames are not the same, but on this file system they are required to be. This flavor is built on **fs:rename-failure**.

fs:rename-across-hosts *Flavor*

The hosts of the initial and target pathnames are not the same. This flavor is built on **fs:rename-failure**.

11.2.3.9 fs:change-property-failure

fs:change-property-failure *Flavor*

This set includes errors related to attempts to change properties of a file. This might mean that you tried to set a property that only the file system is allowed to set. For file systems without user-defined properties, it might mean that no such property exists. This flavor is built on **fs:file-operation-failure**.

fs:unknown-property *Flavor*

The property is unknown. This flavor is built on **fs:change-property-failure**.

fs:invalid-property-value *Flavor*

The new value provided for the property is invalid. This flavor is built on **fs:change-property-failure**.

11.2.3.10 fs:delete-failure

fs:delete-failure *Flavor*

This set includes errors related to attempts to delete a file. It applies to cases where the file server reports that it cannot delete a file. The exact events involved depend on what the host file server has received from the host. This flavor is built on **fs:file-operation-failure**.

fs:directory-not-empty *Flavor*

An invalid deletion of a nonempty directory was attempted. This flavor is built on **fs:delete-failure**.

fs:dont-delete-flag-set *Flavor*

Deleting a file with a "don't delete" flag was attempted. This flavor is built on **fs:delete-failure**.

11.2.3.11 Miscellaneous Operations Failures

fs:circular-link *Flavor*

The pathname is a link that eventually gets linked back to itself. This flavor is built on **fs:file-operation-failure**.

fs:unimplemented-option *Flavor*

This set includes errors in which an option to a command is not implemented. This flavor is built on **fs:file-operation-failure**.

fs:inconsistent-options *Flavor*

Some of the options given in this operation are inconsistent with others. This flavor is built on **fs:file-operation-failure**.

fs:invalid-byte-size *Flavor*

The value of the "byte size" option was not valid. This flavor is built on **fs:unimplemented-option**.

fs:no-more-room *Flavor*

The file system is out of room. This can mean any of several things:

- the entire file system might be full
- the particular volume that you are using might be full
- your directory might be full
- you might have run out of your allocated quota
- other system-dependent things

This flavor is built on **fs:file-operation-failure**. The **:retry-file-operation** proceed type from **fs:file-error** is sometimes provided.

fs:filepos-out-of-range *Flavor*

Setting the file pointer past the end-of-file position or to a negative position was attempted. This flavor is built on **fs:file-operation-failure**.

fs:file-locked *Flavor*

The file is locked. It cannot be accessed, possibly because it is in use by some other process. Different file systems can have this problem in various system-dependent ways. This flavor is built on **fs:file-operation-failure**.

fs:file-open-for-output *Flavor*

Opening a file that was already opened for output was attempted. This flavor is built on **fs:file-operation-failure**. Note: ITS, TOPS-20, and TENEX file servers do not use this condition; they signal **fs:file-locked** instead.

fs:not-available *Flavor*

The file or device exists but is not available. Typically, the disk pack is not mounted on a drive, the drive is broken, or the like. Probably operator intervention is required to fix the problem, but retrying the operation is likely to work after the problem is solved. This flavor is built on **fs:file-operation-failure**. Do not confuse this with **fs:host-not-available**.

11.2.4 Pathname Errors

fs:pathname-error *Flavor*

This set includes errors related to pathnames. This is built on the **error** flavor. The following flavors are built on this one.

fs:parse-pathname-error *Flavor*

A problem occurred in attempting to parse a pathname.

fs:invalid-pathname-component *Flavor*

Attempt to create a pathname with an invalid component.

Message	*Value returned*
:pathname	the pathname
:component-value	the invalid value
:component	the name of the component (a keyword symbol such as **:name** or **:directory**)
:component-description	a "pretty name" for the component (such as file name or directory)

The **:new-component** proceed type is defined with one argument, a component value to use instead.

At the time this is signalled, a pathname object with the invalid component has actually been created; this is what the **:pathname** message returns. The error is signalled just after the pathname object is created before it goes in the pathname hash table.

fs:unknown-pathname-host *Flavor*

The function **fs:get-pathname-host** was given a name that is not the name of any known file computer. The **:name** message returns the name (a string).

fs:undefined-logical-pathname-translation *Flavor*

A logical pathname was referenced but is not defined. The **:logical-pathname** message returns the logical pathname. This flavor has a **:define-directory** proceed type, which prompts for a physical pathname whose directory component is the translation of the logical directory on the given host.

11.2.5 Network Errors

sys:network-error *Flavor*

This set includes errors signalled by networks. These are generic network errors that are used uniformly for any supported networks. This flavor is built on **error**.

11.2.5.1 Local Network Problems

sys:local-network-error *Flavor*

This set includes network errors related to problems with one's own Lisp Machine rather than with the network or the foreign host. This flavor is built on **sys:network-error**.

sys:network-resources-exhausted *Flavor*

The local network control program exhausted some resource; for example, its connection table is full. This flavor is built on **sys:local-network-error**.

sys:unknown-address *Flavor*

The network control program was given an address that is not understood. The **:address** message returns the address. This flavor is built on **sys:local-network-error**.

sys:unknown-host-name *Flavor*

The host parser (**si:parse-host**) was given a name that is not the name of any known host. The **:name** message returns the name. This flavor is built on **sys:local-network-error**.

11.2.5.2 Remote Network Problems

sys:remote-network-error *Flavor*

This set includes network errors related to problems with the network or the foreign host, rather than with one's own Lisp Machine.

Message	*Value returned*
:foreign-host	the remote host
:connection	the connection or **nil** if no particular connection is involved

This flavor is built on **sys:network-error**.

sys:bad-connection-state *Flavor*

This set includes remote errors in which a connection enters a bad state. This flavor is built on **sys:remote-network-error**. It actually can happen due to local causes, however. In particular, if your Lisp Machine stays inside a **without-interrupts** for a long time, the network control program might decide that a host is not answering periodic status requests and put its connections into a closed state.

sys:connection-error *Flavor*

This set includes remote errors that occur while trying to establish a new network connection. The **:contact-name** message to any error object in this set returns the contact name that you were trying to connect to. This flavor is built on **sys:remote-network-error**.

sys:host-not-responding *Flavor*

This set includes errors in which the host is not responding, whether during initial connection or in the middle of a connection. This flavor is built on **sys:remote-network-error**.

11.2.5.3 Connection Problems

sys:host-not-responding-during-connection *Flavor*

The network control program timed out while trying to establish a new connection to a host. The host might be down, or the network might be down. This flavor is built on **sys:host-not-responding** and **sys:connection-error**.

sys:host-stopped-responding *Flavor*

A host stopped responding during an established network connection. The host or the network might have crashed. This flavor is built on **sys:host-not-responding** and **sys:bad-connection-state**.

sys:connection-refused *Flavor*

The foreign host explicitly refused to accept the connection. The **:reason** message returns a text string from the foreign host containing its explanation, or **nil** if it had none. This flavor is built on **sys:connection-error**.

sys:connection-closed *Flavor*

An established connection became closed. The **:reason** message returns a text string from the foreign host containing its explanation, or **nil** if it had none. This flavor is built on **sys:bad-connection-state**.

sys:connection-closed-locally *Flavor*

The local host closed the connection and then tried to use it. This flavor is built on **sys:bad-connection-state**.

sys:connection-lost *Flavor*

The foreign host reported a problem with an established connection and that connection can no longer be used. The **:reason** message returns a text string from the foreign host containing its explanation, or **nil** if it had none. This flavor is built on **sys:bad-connection-state**.

sys:connection-no-more-data *Flavor*

No more data remain on this connection. This flavor is built on **sys:bad-connection-state**.

11.2.6 Tape Errors

tape:tape-error *Flavor*

This set includes all tape errors. This flavor is built on **error**.

tape:mount-error *Flavor*

A set of errors signalled because a tape could not be mounted. This includes problems such as no ring and drive not ready. Normally, **tape:make-stream** handles these errors and manages mount retry. This flavor is built on **tape:tape-error**.

tape:tape-device-error *Flavor*

A hardware data error, such as a parity error, controller error, or interface error, occurred. This flavor has **tape:tape-error** as a **:required-flavor**.

tape:end-of-tape *Flavor*

The end of the tape was encountered. When this happens on writing, the tape usually has a few more feet left, in which the program is expected to finish up and write two end-of-file marks. Normally, closing the stream does this automatically. Whether or not this error is ever seen on input depends on the tape controller. Most systems do not see the end of tape on reading, and rely on the software that wrote the tape to have cleanly terminated its data, with EOFs.

This flavor is built on **tape:tape-device-error** and **tape:tape-error**.

Index

symbolics™

PKG Packages

Cambridge, Massachusetts

Packages
# 990086

February 1984

This document corresponds to Release 5.0.

This document was prepared by the Documentation Group of Symbolics, Inc.

Printing year and number: 87 86 85 84 9 8 7 6 5 4 3 2 1

Table of Contents

1. The Need for Multiple Contexts

A Lisp program is a collection of function definitions. The functions are known by their names, and so each must have its own name to identify it. Clearly a programmer must not use the same name for two different functions.

The Lisp Machine consists of a huge Lisp environment, in which many programs must coexist. All of the "operating system", the compiler, the editor, and a wide variety of programs are provided in the initial environment. Furthermore, every program that you use during his session must be loaded into the same environment. Each of these programs is composed of a group of functions; each function must have its own distinct name to avoid conflicts. For example, if the compiler had a function named **pull**, and you loaded a program that had its own function named **pull**, the compiler's **pull** would be redefined, probably breaking the compiler.

It would not really be possible to prevent these conflicts, since the programs are written by many different people who could never get together to hash out who gets the privilege of using a specific name such as **pull**.

Now, if we are to enable two programs to coexist in the Lisp world, each with its own function **pull**, then each program must have its own symbol named "**pull**", because one symbol cannot have two function definitions. The same reasoning applies to any other use of symbols to name things. Not only functions but variables, flavors, and many other things are named with symbols, and hence require isolation between symbols belonging to different programs.

A *package* is a mapping from names to symbols. When two programs are not closely related and hence are likely to have conflicts over the use of names, the programs can use separate packages to enable each program to have a different mapping from names to symbols. In the example above, the compiler can use a package that maps the name **pull** into a symbol whose function definition is the compiler's **pull** function. Your program can use a different package that maps the name **pull** into a different symbol whose function definition is your function. When your program is loaded, the compiler's **pull** function is not redefined, because it is attached to a symbol that is not affected by your program. The compiler does not break.

The word "package" is used to refer to a mapping from names to symbols because a number of related symbols are packaged together into a single entity. Since the substance of a program (such as its function definitions and variables) consists of attributes of symbols, a package also packages together the parts of a program. The package system allows the author of a group of closely related programs that should share the same symbols to define a single package for those programs.

It is important to understand the distinction between a name and a symbol. A name is a sequence of characters that appears on paper (or on a screen or in a file). This is often called a *printed representation.* A symbol is a Lisp object inside the machine. You should keep in mind how Lisp reading and loading work. When a source file is read into the Lisp Machine, or a compiled binary file is loaded in, the file itself obviously cannot contain Lisp objects; it contains printed representations of those objects. When the reader encounters a printed representation of a symbol, it uses a package to map that printed representation (a name) into the symbol itself. The loader does the same thing. The package system arranges to use the correct package whenever a file is read or loaded; this is explained in detail in section 3, page 13.

Here is another example: suppose there are two programs named **chaos** and **arpa**, for handling the Chaosnet and Arpanet respectively. The author of each program wants to have a function called **get-packet**, which reads in a packet from the network. Also, each wants to have a function called **allocate-pbuf**, which allocates the packet buffer. Each "get" routine first allocates a packet buffer, and then reads bits into the buffer; therefore, each version of **get-packet** should call the respective version of **allocate-pbuf**.

Without the package system, the two programs could not coexist in the same Lisp environment. But the package system can be used to provide a separate space of names for each program. What is required is to define a package named **chaos** to contain the Chaosnet program, and another package **arpa** to hold the Arpanet program. When the Chaosnet program is read into the machine, the names it uses are translated into symbols via the **chaos** package. So when the Chaosnet program's **get-packet** refers to **allocate-pbuf**, the **allocate-pbuf** in the **chaos** package is found, which is the **allocate-pbuf** of the Chaosnet program — the right one. Similarly, the Arpanet program's **get-packet** would be read in using the **arpa** package and would refer to the Arpanet program's **allocate-pbuf**.

2. Sharing of Symbols Among Packages

So far we have seen how the package system keeps programs isolated by giving each program its own set of symbols. Another important job for the package system is to provide controlled sharing of symbols between packages. It would not be adequate for each package's set of symbols to be completely disjoint from the symbols of every other package. For example, almost every package ought to include the whole Lisp language: **car**, **cdr**, **format**, and so on should be available to every program.

There is a critical tension between these two goals of the package system. On the one hand, we want to keep the packages isolated, to avoid the need to think about conflicts between programs in the choice of names for things. On the other hand, we want to provide connections among packages so that the facilities of one program can be made available to other programs. All of the complexity of the package system arises from this tension. Almost all of the features described in this document exist to provide easy ways to control the sharing of symbols among packages, while avoiding accidental unwanted sharing of symbols. Unexpected sharing of a symbol between packages, when the authors of the programs in those packages expected to have private symbols of their own, is a *name conflict* and can cause programs to go awry. See the section "Name Conflict: Packages".

Note that sharing symbols is not as simple as merely making the symbols defined by the Lisp language available in every package. A very important feature of the Lisp Machine is *shared programs*; if one person writes a function to, say, print numbers in Roman numerals, any other function can call it to print Roman numerals. This contrasts sharply with many other systems, where many different programs have been written to accomplish the same thing.

For example, the routines to manipulate a robot arm might be a separate program, residing in its own package. A second program called **blocks** (the blocks world, of course) wants to manipulate the arm, so it would want to call functions from the arm package. This means that the blocks program must have a way to name those robot arm functions. One way to do this is to arrange for the name-to-symbol mapping of the blocks package to map the names of those functions into the same identical symbols as the name-to-symbol mapping of the arm package. These symbols would then be shared between the two packages.

This sharing must be done with great care. The symbols to be shared between the two packages constitute an interface between two modules. The names to be shared must be agreed upon by the authors of both programs, or at least known to them. They cannot simply make every symbol in the arm program available to the blocks program. Instead they must define some subset of the symbols used by the arm program as its *interface* and make only those symbols available. Typically each name in the interface is carefully chosen (more carefully than names that are only used internally). The arm program comes with documentation listing the symbols that constitute its interface and describing what each is used for. This tells the author of the blocks program not only that a particular symbol is being used as the name of a function in the arms program (and thus can't be used for a function elsewhere), but also what that function does (move the arm, for instance) when it is called.

The package system provides for several styles of interface between modules. For several examples of how the blocks program and the arm program might communicate: See the section "Examples: Packages".

An important aspect of the package system is that it makes it necessary to clarify the modularity of programs and the interfaces between them. The package system provides some tools to allow the interface to be explicitly defined and to check that everyone agrees on the interface. These are explained in the rest of the document.

2.1 External Symbols

The name-to-symbol mappings of a package are divided into two classes, *external* and *internal*. We refer to the symbols accessible via these mappings as being *external* and *internal* symbols of the package in question, though really it is the mappings that are different and not the symbols themselves. Within a given package, a name refers to one symbol or to none; if it does refer to a symbol, that symbol is either external or internal in that package, but not both.

External symbols are part of the package's public interface to other packages. These are supposed to be chosen with some care and are advertised to outside users of the package. Internal symbols are for internal use only, and these symbols are normally hidden from other packages. Most symbols are created as internal symbols; they become external only if they are explicitly *exported* from a package.

A symbol may appear in many packages. It may be external in one package and internal in another. It is valid for a symbol to be internal in more than one package, and for a symbol to be external in more than one package. A name may refer to different symbols in different packages. However, a symbol always has the same name no matter where it appears. This restriction is imposed both for conceptual simplicity and for ease of implementation.

2.2 Package Inheritance

The name-to-symbol mappings of a package are divided into two classes in another way. Some mappings are established by the package itself, while others are inherited from other packages. When package A inherits mappings from package B, package A is said to *use* package B. A symbol is said to be *accessible* in a package if its name maps to it in that package, whether directly or by inheritance. A symbol is said to be *present* in a package if its name maps to it directly (not by inheritance).

If a symbol is accessible to a package, then it can be referenced by a program that is read into that package. Inheritance allows a package to be built up by combining symbols from a number of other packages.

Package inheritance interacts with the distinction between internal and external symbols. When one package uses another, it inherits only the external symbols of that package. This is necessary in order to provide a well-defined interface and avoid accidental name conflicts. The external symbols are the ones that are carefully chosen and advertised. If internal symbols were inherited, it would be hard to predict just which symbols were shared between packages.

A package may use any number of other packages; it inherits the external symbols of all of them. If two of these external symbols had the same name it would be unpredictable which one would be inherited, so this is considered to be a name-conflict error. Consequently the order of the used packages is immaterial and does not affect what symbols are accessible.

Only symbols that are present in a package can be external symbols of that package. However, the package system hides this restriction by copying an inherited mapping directly into a package if you request that the symbol be exported. Note: When package A uses package B, it inherits the external symbols of B. But these do not become external symbols of A, and will not be inherited by package C that uses package A. A symbol becomes an external symbol of A only by an explicit request to export it from A.

A package may be made to use another package by the **:use** option to **defpackage** or **make-package** or by calling the **use-package** function. See the section "Defining a Package". See the section "Interpackage Relations".

2.3 The global Package

Almost every package should have the basic symbols of the Lisp language accessible to it. This includes symbols that are names of useful functions, such as **cdr**, **cons**, and **print**; symbols that are names of special forms, such as **cond** and **selectq**; symbols that are names of useful variables, such as **base**, **standard-output**, and *****; symbols that are names of useful constants, such as **lambda-list-keywords** and **%%kbd-control-meta**; and symbols that are used by the language as symbols in their own right, such as **&optional**, **t**, **nil**, and **special**.

Rather than providing an explicit interface between every program and the Lisp language, listing explicitly the particular symbols from the Lisp language that that program intends to use, it is more convenient to make all the Lisp symbols accessible. Unless otherwise specified, every package inherits from the **global** package. The external symbols of **global** are all the symbols of the Lisp language, including all the symbols documented without a colon (:) in their name. The **global** package has no internal symbols.

All programs share the global symbols, and cannot use them for private purposes. For example, the symbol **delete** is the name of a Lisp function and thus is in the **global** package. Even if a program does not use the **delete** function, it will inherit the global symbol named **delete** and therefore cannot define its own function with that name to do something different. Furthermore, if two programs each want to use the symbol **delete** as a property list indicator, they may bump into each other because they will not have private symbols. There exists a mechanism called *shadowing* that you can use to declare that a private symbol is desired rather than inheriting the global symbol. See the section "Shadowing Symbols". You can also use the **where-is** function and the Where Is Symbol (m-X) editor command to determine whether a symbol is private or shared when writing a program.

Similar to the **global** package is the **system** package, which contains all the symbols that are part of the "operating system" interface or the machine architecture, but not regarded as part of the Lisp language. The **system** package is not inherited unless specifically requested.

Here is how package inheritance works in the example of the two network programs. (See the section "The Need for Multiple Contexts".) When the Chaosnet program is read into the Lisp world, the current package is the **chaos** package. Thus all of the names in the Chaosnet program are mapped into symbols by the **chaos** package. If there is a reference to some well-known global symbol such as **append**, it is found by inheritance from the **global** package, assuming no symbol by that name is present in the **chaos** package. If, however, there is a reference to a symbol that you created, a new symbol will be created in the **chaos** package. Suppose the name **get-packet** is referenced for the first time. No symbol by this name is directly present in the **chaos** package, nor is such a symbol inherited from **global**. Therefore the reader (actually the **intern** function) creates a new symbol named **get-packet** and makes it present in the **chaos** package. When **get-packet** is referred to later in the Chaosnet program, that symbol will be found.

When the Arpanet program is read in, the current package is **arpa** instead of **chaos**. When the Arpanet program refers to **append**, it gets the **global** one; that is, it shares the same symbol that the Chaosnet program got. However, if it refers to **get-packet**, it does *not* get the same symbol the Chaosnet program got, because the **chaos** package is not being searched. Rather, the **arpa** and **global** packages are searched. A new symbol named **get-packet** is created and made present in the **arpa** package.

So what has happened is that there are two **get-packets**: one for **chaos** and one for **arpa**. The two programs are loaded together without name conflicts.

2.4 Home Package of a Symbol

Every symbol has a *home package*. When a new symbol is created by the reader and made present in the current package, its home package is set to the current package. The home package of a symbol may be obtained with the **symbol-package** function.

Most symbols are present only in their home package; however, it is possible to make a symbol be present in any number of packages. Only one of those packages can be distinguished as the home package; normally this will be the first package in which the symbol was present. The package system makes an effort to ensure that a symbol *is* present in its home package. When a symbol is first created by the reader (actually by the **intern** function), it is guaranteed to be present in its home package. If the symbol is removed from its home package (by the **remob** function), the home package of the symbol will be set to **nil**, even if the symbol is still present in some other package.

Some symbols are not present in any package; they are said to be *uninterned*. See the section "Mapping Names to Symbols". The **make-symbol** function can be used to create such a symbol. An uninterned symbol has no home package; the **symbol-package** function will return **nil** given such a symbol.

When a symbol is printed, for example, with **prin1**, the printer produces a printed representation that the reader will turn back into the same symbol. If the symbol is not accessible to the current package, a qualified name is printed. See the section "Qualified Names". The symbol's home package is used as the prefix in the qualified name.

2.5 Importing and Exporting Symbols

A symbol may be made accessible to packages other than its home package in two ways, *importing* and *exporting*.

Any symbol may be made present in a package by *importing* it into that package. This is how a symbol can be present in more than one package at the same time. After importing a symbol into the current package, it may be referred to directly with an unqualified name. Importing a symbol does not change its home package, and does not change its status in any other packages in which it is present.

When a symbol is imported, if another symbol with the same name is already accessible to the package, a name-conflict error is signalled. The *shadowing-import* operation is a combination of shadowing (described in the next section) and importing; it resolves a name conflict by getting rid of any existing symbol accessible to the package.

Any number of symbols may be *exported* from a package. This declares them to be *external* symbols of that package and makes them accessible in any other packages that *use* the first package. To use a package means to inherit its external symbols.

When a symbol is exported, the package system makes sure that no name conflict is caused in any of the packages that inherit the newly exported symbol.

A symbol may be imported by using the **:import**, **:import-from**, or **:shadowing-import** option to **defpackage** and **make-package**, or by calling the **import** or **shadowing-import** function. A symbol may be exported by using the **:export** option to **defpackage** or **make-package**, or by calling the **export** function. See the section "Defining a Package". See the section "Import, Export, and Shadow".

2.6 Shadowing Symbols

You can avoid inheriting unwanted symbols by *shadowing* them. To shadow a symbol that would otherwise be inherited, you create a new symbol with the same name and make it present in the package. The new symbol is put on the package's list of shadowing symbols, to tell the package system that it is not an accident that there are two symbols with the same name. A shadowing symbol takes precedence over any other symbol of the same name that would otherwise be accessible to the package. Shadowing allows the creator of a package to avoid name conflicts that are anticipated in advance.

Here is an example of shadowing. Suppose you want to define a function named **nth** that is different from the normal **nth** function. (Perhaps you want **nth** to be compatible with the Interlisp function of that name.) Simply writing **(defun nth ...)** in your program would redefine the system-provided **nth** function, probably breaking other programs that use it. (The system detects this and queries you before proceeding with the redefinition.)

The way to resolve this conflict is to put the program (call it snail) that needs the incompatible definition of **nth** in its own package and to make the **snail** package shadow the symbol **nth**.

Now there are two symbols named **nth**, so defining **snail**'s **nth** to be an Interlisp-compatible function will not affect the definition of the global **nth**. Inside the snail program, the global symbol **nth** cannot be seen, which is why we say that it is shadowed. If some reason arises to refer to the global symbol **nth** inside the snail program, the qualified name **global:nth** can be used.

A shadowing symbol may be established by the **:shadow** or **:shadowing-import** option to **defpackage** or **make-package**, or by calling the **shadow** or **shadowing-import** function. See the section "Import, Export, and Shadow".

2.7 Keywords

The Lisp reader is not context-sensitive; it reads the same printed representation as the same symbol regardless of whether the symbol is being used as the name of a function, the name of a variable, a quoted constant, a syntactic word in a special form, or anything else. The consistency and simplicity afforded by this lack of context sensitivity are very important to Lisp's interchangeability of programs and data, but they do cause a problem in connection with packages. If a certain function is to be shared between two packages, then the symbol that names that function has to be shared for all contexts, not just for functional context. This can accidentally cause a variable, or a property list indicator, or some other use of a symbol, to be shared between two packages when not desired. Consequently, it is important to minimize the number of symbols that are shared between packages, since every such symbol becomes a "reserved word" that cannot be used without thinking about the implications. Furthermore, the set of symbols shared among all the packages in the world is not legitimately user-extensible, because adding a new shared symbol could cause a name conflict between unrelated programs that use symbols by that name for their own private purposes.

On the other hand, there are many important applications for which the package system just gets in the way and one would really like to have *all* symbols shared between packages. Typically this occurs when symbols are used as objects in their own right, rather than just as names for things.

This dilemma is partially resolved by the introduction of *keywords* into the language. Keywords are a set of symbols that is disjoint from all other symbols and exist as a completely independent set of names. There is no separation of packages as far as keywords are concerned; all keywords are available to all packages and there cannot be two distinct keywords with the same name. There can, of course, be a keyword with the same name as one or more ordinary symbols. To distinguish keywords from ordinary symbols, the printed representation of a keyword starts with a colon (:) character.

Since keywords are disjoint from ordinary symbols, the sharing of keywords among all packages does not affect the separation of ordinary symbols into private symbols of each package. The set of keywords is user-extensible; simply reading the printed representation of a new keyword is enough to create it.

Keywords are implemented as symbols whose home package is the **keyword** package, which has the empty string as a nickname. See the section "Package Names". Hence the printed representation of a keyword, a symbol preceded by a colon, is actually just a qualified name. As a matter of style, keywords are never imported into other packages and the **keyword** package is never inherited (used) by another package.

As a syntactic convenience, every keyword is a constant that evaluates to itself (just like numbers and strings). This eliminates the need to write a lot of " ' " marks when calling a function that takes **&key** arguments, but makes it impossible to have a variable whose name is a keyword. However, there is no desire to use keywords as names of variables (or of functions), because the colon would look ugly. In fact, no syntactic words of the Lisp language are keywords. Names of special forms, the **otherwise** that goes at the end of a **selectq**, the **lambda** that identifies an interpreted function, names of declarations such as **special** and **arglist**, all are not keywords.

The only aspects of symbols significant to keywords are name and property list; otherwise, keywords could just as easily be some other data type. (Note that keywords are referred to as enumeration types in some other languages.)

Some examples of how keywords are used:

Keywords can be used as symbolic names for elements of a finite set. For example, when opening a file with the **open** function one must specify a direction. The various directions are named with keywords, such as **:input** and **:output**. See the document *Streams*.

One of the most common uses of keywords is to name arguments to functions that take a large number of optional arguments and therefore are inconvenient to call with arguments identified positionally. Each argument is preceded by a keyword that tells the function how to use that argument. When the function is called, it compares each keyword that was passed to it against each of the keywords it knows, using **eq**.

Another common use for keywords is as names for messages that are passed to active objects such as instances. When an instance receives a message, it compares its first argument against all the message names it knows, using **eq**. See the section "Generic Operations".

Since there cannot be two distinct keywords with the same name, keywords are not used for applications in which name conflicts can arise. For example, suppose a program stores data on the property lists of symbols. The data are internal to the program but the symbols may be global. An example of this would be a program-understanding program that puts some information about each Lisp function and special form on the symbol that names that function or special form. The indicator used should not be a keyword, because some other program might choose the same keyword to store its own internal data on the same symbol, and there would be a name conflict.

It is permissible, and in fact quite common, to use the same keyword for two different purposes when the two purposes are always separable by context. For instance, the use of keywords to name arguments to functions does not permit the possibility of a name conflict if you always know what function you are calling.

To see why keywords are used to name **&key** arguments, consider the function **make-array**, which takes one required argument followed by any number of keyword arguments. For example, the following specifies, after the first required argument, two options with names **:leader-length** and **:type** and values **10** and **art-string**.

```
(make-array 100 :leader-length 10 :type 'art-string)
```

The file containing **make-array**'s definition is in the **system-internals** package, but the function is accessible to everyone without the use of a qualified name because the symbol **make-array** is itself inherited from **global**. But all the keyword names, such as **type**, are short and should not have to exist in **global** where they would either cause name conflicts or use up all the "good" names by turning them into reserved words. However, if all callers of **make-array** had to specify the options using long-winded qualified names such as **system-internals:leader-length** and **system-internals:type** (or even **si:leader-length** and **si:type**) the point of making **make-array** global so that one can write **make-array** rather than **system-internals:make-array** would be lost. Furthermore, by rights one should not have to know about internal symbols of another package in order to use its documented external interface. By using keywords to name the arguments, we avoid this problem while not increasing the number of characters in the program, since we trade a "'" for a ":".

The data type names used with the **typep** function and the **typecase** and **check-arg-type** special forms are sometimes keywords and sometimes not keywords. The names of data types that are built into the machine, such as **:symbol**, **:list**, **:fixnum**, and **:compiled-function**, are keywords. On the other hand, the names of data types that are defined as flavors or structures, such as **package** or **tv:window**, are not keywords. This unfortunate anomaly exists for historical reasons and will be removed by Common Lisp, where names of data types, like names of functions, are never keywords.

When in doubt as to whether or not a symbol of the language is supposed to be a keyword, check the manual to see whether it is documented with a colon at the front of its name.

3. Packages and Writing Programs

If you are an inexperienced user, you need never be aware of the existence of packages when writing programs. The **user** package is selected by default as the package for reading expressions typed at the Lisp Listener. Files will be read in the **user** package if no package is specified. Since all the functions that users are likely to need are provided in the **global** package, which is used by **user**, they are all accessible. In the documentation, functions that are not in the **global** package are documented with colons in their names, so typing the name the way it is documented will work. Keywords, of course, must be typed with a prefix colon, but since that is the way they are documented it is possible to regard the colon as just part of the name, not as anything having to do with packages.

The current package is the value of the variable **package**. The current package in the "selected" process is displayed in the status line. This allows you to tell how forms you type in will be read.

If you are writing a program that you expect others to use, you should put it in some package other than **user**, so that its internal functions will not conflict with names other users use. Whether for this reason or for any other, if you are loading your programs into packages other than **user** there are special constructs that you will need to know about, including **defpackage**, qualified names, and file attribute lists. See the section "Defining a Package". See the section "Qualified Names".

Obviously, every file must be loaded into the right package to serve its purpose. It may not be so obvious that every file must be compiled in the right package, but it's just as true. Any time the names of symbols appearing in the file must be converted to the actual symbols, the conversion must take place relative to a package.

The way that the system usually decides which package to use for a file is the file's *attribute list*. See the section "File Property Lists". The package can also be selected by **make-system**. See the section "Making a System". A compiled file remembers the name of the package it was compiled in, and loads into the same package. In the absence of any of these specifications, the package defaults to the current value of **package**, which is usually the **user** package unless you change it.

The file attribute list of a character file is the line at the front of the file that looks something like:

```
;;; -*- Mode:Lisp; Package:System-Internals -*-
```

This specifies that the package whose name or nickname is **system-internals** is to be used. Alphabetic case does not matter in these specifications. Relative package names are not used, since there is no meaningful package to which the name could be relative. See the section "Relative Package Names".

If the package attribute contains parentheses, then the package will be automatically created if it is not found. This is useful when a single file is in its own package, not shared with any other files, and no special options are required to set up that package. The valid forms of package attribute are:

-*- Package: *Name* **-*-**
: Signal an error if the package is not found, allowing you to load the package's definition from another file, specify the name of an existing package to use instead, or create the package with default characteristics.

-*- Package: (*Name***) -*-**
: If the package is not found, create it with the specified name and default characteristics. It will use **global** so that it inherits the Lisp language symbols.

-*- Package: (*Name use***) -*-**
: If the package is not found, create it with the specified name and make it use *use*, which can be the name of a package or a list of names of packages.

-*- Package: (*Name use size***) -*-**
: If the package is not found, create it with the specified name and make it use *use*, which can be the name of a package or a list of names of packages. *size* is a decimal number, the expected number of symbols that will be present in the package.

-*- Package: (*Name keyword value keyword value...***) -*-**
: If the package is not found, create it with the specified name. The rest of the list supplies the keyword arguments to **make-package**. In the event of an ambiguity between this form and the previous one, the previous one is preferred. You can avoid ambiguity by specifying more than one keyword.

Binary files have similar file attribute lists. The compiler always puts in a **:package** attribute to cause the binary file to be loaded into the same package it was compiled in, unless this attribute is overridden by arguments to **load**.

4. Package Names

Each package has a name and perhaps some nicknames. These are assigned when the package is created, though they can be changed later. A package's name should be something long and self-explanatory like **editor**; there might be a nickname that is shorter and easier to type, like **ed**. Typically the name of a package is also the name of the program that resides in that package.

There is a single name space for packages. Instead of setting up a second-level package system to isolate names of packages from each other, we simply say that package name conflicts are to be resolved by using long explanatory names. There are sufficiently few packages in the world that a mechanism to allow two packages to have the same name does not seem necessary. Note that for the most frequent use of package names, qualified names of symbols, name clashes between packages can be alleviated using the mechanism described in the next section — relative names.

The syntax conventions for package names are the same as for symbols. When the reader sees a package name (as part of a qualified symbol name), alphabetic characters in the package name are converted to uppercase unless preceded by the "/" escape character or unless the package name is surrounded by "|" characters. When a package name is printed by the printer, if it does not consist of all uppercase alphabetics and non-delimiter characters, the "/" and "|" escape characters are used.

Package name lookup is currently case-insensitive, but it may be changed in the future to be case-sensitive. In any case it would not be a good idea to make two packages whose names differ only in alphabetic case.

Internally names of packages are strings, but the functions that require a package-name argument from the user accept either a symbol or a string. If you supply a symbol, its print-name will be used, and this will already have undergone case conversion by the usual rules. If you supply a string, you must be careful to capitalize the string in the same way that the package's name is capitalized.

Note that **|Foo|:|Bar|** refers to a symbol whose name is **"Bar"** in a package whose name is **"Foo"**. By contrast, **|Foo:Bar|** refers to a 7-character symbol with a colon in its name, and is interned in the current package. Following the convention used in the documentation for symbols, we will show package names as being in lowercase, even though the name string is really in uppercase.

In addition to the normal packages discussed up until now, there can be *invisible* packages. An invisible package has a name, but it is not entered into the system's table that maps package names to packages. An invisible package cannot be referenced via a qualified name (unless you set up a relative name for it) and cannot be used in such contexts as the **:use** keyword to **defpackage** and **make-package** (unless you pass the package object itself, rather than its name). Invisible packages are useful if you simply want a package to use as a data structure, rather than as the package in which to write a program.

4.1 Relative Package Names

In addition to the absolute package names (and nicknames) described in the previous section, there can be *relative* names for packages. If **p** is a relative name for package B, relative to package A, then in contexts where relative names are allowed and A is the contextually relevant package the name **p** may be used instead of **b**. The relative name mapping *belongs to* package A and defines a new name (**p**) *for* package B. It is important not to confuse the package that the name is relative to with the package that is named.

There are two important differences between relative names and absolute names: relative names are recognized only in certain contexts, and relative names may "shadow" absolute names. One application for relative names is to replace one package by another. Thus if a program residing in package A normally refers to the **thermodynamics** package, but for testing purposes we would like it to use the **phlogiston** package instead, we can give A a relative name mapping from the name **thermodynamics** to the **phlogiston** package. This relative name shadows the absolute name **thermodynamics**.

Another application for relative names is to ease the establishment of a family of mutually dependent packages. For example, if you have three packages named **algebra**, **rings**, and **polynomials**, these packages may refer to each other so frequently that you would like to use the nicknames **a**, **r**, and **p** rather than spelling out the full names each time. It would obviously be bad to use up these one-letter names in the system-wide space of package names; what if someone else has a program with two packages named **reasoning** and **truth-maintenance**, and would like to use the nicknames **r** and **t**? The solution to this name conflict is to make the abbreviated names be relative names defined in the **algebra**, **rings**, and **polynomials** packages. These abbreviations will only be seen by references emanating from those packages, and there is no conflict with other abbreviations defined by other packages.

An extension of the shadowing application for relative names is to set up a complete family of packages parallel to the normal one, such as **experimental-global** and **experimental-user**. Within this family of packages you establish relative name mappings so that the usual names such as **global** and **user** may be used. Certain system utility programs work this way.

When package A uses package B, in addition to inheriting package B's external symbols, any relative name mappings established by package B will be inherited. In the event of a name conflict between relative names defined directly by A and inherited relative names, the inherited name will be ignored. The results if two relative name mappings inherited from two different packages conflict are unpredictable.

The Lisp system does not itself use relative names, so a freshly booted Lisp Machine will contain no relative-name mappings.

Relative names are recognized in the following contexts:

- Qualified symbol names -- The package name before the colon is relative to the package in which the symbol is being read (the value of the variable **package**). The printer prefers a relative package name to an absolute package name when it prints a qualified symbol name.

- Package references in package-manipulating functions -- For example, the package names in the **:use** option to **defpackage** and in the first argument to **use-package** may be relative names. All such relative names are relative to the value of the variable **package**.

- Package arguments that default to the current package -- The functions **intern**, **intern-local**, **intern-soft**, **intern-local-soft**, **remob**, **export**, **unexport**, **import**, **shadow**, **shadowing-import**, **use-package**, and **unuse-package** all take an optional second argument that defaults (except in the case of **remob**) to the current package. If supplied, this argument may be a package, an absolute name of a package, or a relative name of a package. All such relative names are relative to the value of the variable **package**.

Relative names are not recognized in "global" contexts, where there is no obvious contextual package to be relative to, such as:

- File attribute lists ("-*-" lines)

- Package names requested from you as part of error recovery, or in commands such as the Set Package (m-X) editor command.

- The **pkg-find-package** function (unless its optional third argument is specified).

- Package arguments to the **mapatoms**, **pkg-goto**, **describe-package**, and **pkg-kill** functions.

- Package specifiers in the **do-symbols**, **do-local-symbols**, and **do-external-symbols** special forms, and the **interned-symbols** and **local-interned-symbols loop** iteration paths.

When a package object is printed, if it has a relative name (relative to the value of **package**) that differs from its absolute name, both names are printed.

Relative names are established with the **:relative-names** and **:relative-names-for-me** options to **defpackage** and **make-package**. You can also use the **pkg-add-relative-name** function to establish a relative name. The **pkg-delete-relative-name** function removes a relative name.

5. Qualified Names

Often it is desirable to refer to an external symbol in some package other than the current one. You do this through the use of a *qualified name*, consisting of a package name, then a colon, then the name of the symbol. This causes the symbol's name to be looked up in the specified package, rather than in the current one. For example, **editor:buffer** refers to the external symbol named **buffer** of the package named **editor**, regardless of whether there is a symbol named **buffer** in the current package. If there is no package named **editor**, or if no symbol named **buffer** is present in **editor** or if **buffer** is an internal symbol of **editor**, an error is signalled.

On rare occasions, you may need to refer to an *internal* symbol of some package other than the current one. It is invalid to do this with the colon qualifier, since accessing an internal symbol of some other package is usually a mistake. See the section "Avoiding the Internal/External Distinction". However, this operation is legal if you use "::" as the separator in place of the usual colon. If the reader sees **editor::buffer**, the effect is exactly the same as reading **buffer** with **package** temporarily rebound to the package whose name is **editor**. This special-purpose qualifier should be used with caution.

Qualified names are implemented in the Lisp reader by treating the colon character (**:**) specially. When the reader sees one or two colons preceded by the name of a package, it will read in the next Lisp object with **package** bound to that package. Note that the next Lisp object need not be a symbol; the printed representation of any Lisp object may follow a package prefix. If the object is a list, the effect is exactly as if every symbol in that list had been written as a qualified name, using the prefix that appears in front of the list. When a qualified name is among the elements of the list, the package name in the second package prefix is taken relative to the package selected by the first package prefix. The internal/external mode is controlled entirely by the innermost package prefix in effect.

5.1 Avoiding the Internal/External Distinction

To ease the transition for people whose programs are not yet organized according to the distinction between internal and external symbols, a package may be set up so that the ":" type of qualified name does the same thing as the "::" type. This is controlled by the package that appears before the colon, not by the package in which the whole expression is being read. To set this attribute of a package, use the **:colon-mode** keyword to **defpackage** and **make-package**. **:external** causes "**:**" to behave as described above, accessing only external symbols. **:internal** causes "**:**" to behave the same as "**::**", accessing all symbols. Note that **:internal** mode is compatible with **:external** mode except in cases where an error would be signalled. In Release 5.0, the default mode is **:internal** and all predefined system packages are created with this mode. In Common Lisp the default mode is **:external**.

5.2 Qualified Names as Interfaces

In the example of the blocks world and the robot arm, a program in the **blocks** package could call a function named **go-up** defined in the **arm** package by calling **arm:go-up**. **go-up** would be listed among the external symbols of **arm**, using **:export** in its **defpackage**, since it is part of the interface allowing the outside world to operate the arm. If the **blocks** program uses qualified names to refer to functions in the **arm** program, rather than sharing symbols as in the original example, then the possibility of name conflicts between the two programs is eliminated.

Similarly, if the **chaos** program wanted to refer to the **arpa** program's **allocate-pbuf** function, it would simply call **arpa:allocate-pbuf**, assuming that function had been exported. If it was not exported (because **arpa** thought no one from the outside had any business calling it), the **chaos** program would call **arpa::allocate-pbuf**.

5.3 Printed Representation of Symbols

The printer uses qualified names when necessary. (The **princ** function, however, never prints qualified names for symbols.) The goal of the printer (for example, the **prinl** function) when printing a symbol is to produce a printed representation that the reader will turn back into the same symbol. When a symbol that is accessible in the current package (the value of **package**) is printed, a qualified name is not used, regardless of whether the symbol is present in the package. This happens for one of three reasons: because this is its home package, is present because it was imported, or is not present but was inherited. When an inaccessible symbol is printed, a qualified name is used. The printer chooses whether to use ":" or "::" based on whether the symbol is internal or external and the **:colon-mode** of its home package. The qualified name used by the printer may be read back in and will yield the same symbol. If the inaccessible symbol were printed without qualification, the reader would translate that printed representation into a different symbol, probably an internal symbol of the current package.

The qualified name used by the printer is based on the symbol's home package, not on the path by which it was originally read (which of course cannot be known). Suppose **foo** is an internal symbol of package A, has been imported into package B, and has then been exported from package B. If it is printed while **package** is neither A nor B, nor a package that uses B, the name printed will be **a::foo**, not **b:foo**, because **foo**'s home package is A. This is an unlikely case, of course.

In addition to the simplest printed representation of a symbol, its name standing by itself, there are four forms of qualified name for a symbol. These are accepted by the reader and are printed by the printer when necessary; except when printing an uninterned symbol, the printer prints some printed representation that will yield the same symbol when read. The following table shows the four forms of qualified name, assuming that the **foo** package specifies **:colon-mode :external**. If **foo** specifies **:colon-mode :internal**, as is currently the default, the first and second forms are equivalent.

foo:bar	When read, looks up **bar** among the external symbols of the package named **foo**. Printed the when symbol **bar** is external in its home package **foo** and is not accessible in the current package.
foo::bar	When read, interprets **bar** as if **foo** were the current package. Printed when the symbol **bar** is internal in its home package **foo** and is not accessible in the current package.
:bar	When read, interprets **bar** as an external symbol in the **keyword** package. Printed when the home package of the symbol **bar** is **keyword**.
#:bar	When read, creates a new uninterned symbol named **bar**. Printed when the symbol named **bar** has no home package.

5.4 Multilevel Qualified Names

Due to shadowing by relative names, a given package may sometimes be inaccessible. In this case a multilevel qualified name, containing more than one package prefix, may be used.

Suppose packages **moe**, **larry**, **curly**, and **shemp** exist. For its own reasons, the **moe** package uses **curly** as a relative name for the **shemp** package. Thus, when the current package is **larry** the printed representation **curly:hair** designates a symbol in the **curly** package, but when the current package is **moe** the same printed representation designates a symbol in the **shemp** package.

If the **moe** package is current and the symbol **hair** in the **curly** package needs to be read or printed, the printed representation **curly:hair** cannot be used since it refers to a different symbol. If **curly** had a nickname that is not also shadowed by a relative name it would be used, but suppose there is no nickname. In this case the only possible way to refer to that symbol is with a multilevel qualified name. **larry:curly:hair** would work, since the **larry:** escapes from the scope of **moe**'s relative name. The printer actually prefers to print **global:curly:hair** because of the way it searches for a usable qualified name.

6. Examples

Consider again the example of the robot arm in the blocks world. There are two separate programs, written by different people, interacting with each other in a single Lisp environment. The arm-control program resides in a package named **arm**, and the blocks-world program resides in a package named **blocks**. The operation of the two programs requires them to interact. For example, to move a block from one place to another the **blocks** program calls functions in the **arm** program with names like **raise-arm**, **move-arm**, and **grasp**. To find the edges of the table, the **arm** program accesses variables of the **blocks** program.

Communication between the two programs requires that they both know about certain objects. Usually these objects are the sort that have names (for example, functions or variables). The names are symbols. Thus each program must be able to name some symbols and to know that the other program is naming the same symbols.

Let us consider the case of the function **grasp** in the arm-control program, which the blocks-world program must call in order to pick up a block with the arm. The **grasp** function is named by the symbol **grasp** in the **arm** package. Assume that we are not going to use either of the mechanisms (keywords and the **global** package) that make symbols available to *all* packages; we only want **grasp** to be shared between the two specific packages that need it. There are basically three ways provided by the package system for a symbol to be known by two separate programs in two separate packages.

If the **blocks** package *imports* the symbol **grasp** from the **arm** package, then both packages will map the name **grasp** into the same symbol. The **blocks** package could be defined by:

```
(defpackage blocks
        (:import-from arm grasp))
```

The **arm** package can *export* the symbol **grasp**, along with whatever other symbols constitute its interface to the outside world. If the **blocks** package *uses* the **arm** package, then both packages will again map the name **grasp** into the same symbol. The package definitions would look like:

```
(defpackage arm
        (:export grasp move-arm raise-arm ...))

(defpackage blocks
        (:use arm global))
```

Note that the **blocks** package must explicitly mention that it is using the **global** package as well as the **arm** package, since it is not letting its **:use** clause default. The difference between this method (the export method) and the first method (the import method) is that the list of symbols that is to constitute the interface is associated with the **arm** package, that is, the package that *provides* the interface, not the package that *uses* the interface.

In the third method, we do not have the two packages map the same name into the same symbol. Instead we use a different, longer name for the symbol in the blocks program than the name used by the arm program. This makes it clear, when reading the text of the blocks program, which symbol references are connected with the interface between the two programs. These longer names are called *qualified names*. Again, the **arm** package defines the interface:

```
(defpackage arm
       (:export grasp move-arm raise-arm ...))
```

A fragment of the blocks-world program might look like

```
(defun pick-up (block)
  (clear-top block)
  (arm:grasp (block-coordinates block ':top))
  (arm:raise-arm))
```

arm:grasp and **arm:raise-arm** are qualified names. **pick-up**, **block**, **clear-top**, and **block-coordinates** are internal symbols of the blocks-world program. **defun** is inherited from the **global** package. **:top** is a keyword. Note that although the two programs do not use the same names to refer to the same symbol, the names they use are related in an obvious way, avoiding confusion. The package system makes no provision for the same symbol to be named by two completely arbitrary names.

7. Consistency Rules

Package-related bugs can be very subtle and confusing: the program is not using the same symbols as you think it is using. The package system is designed with a number of safety features to prevent most of the common bugs that would otherwise occur in normal use. This may seem overprotective, but experience with earlier package systems has shown that such safety features are needed.

In dealing with the package system, it is useful to keep in mind the following consistency rules, which remain in force as long as the value of **package** is not changed by you or your code:

- *Read-Read consistency:* Reading the same print name always gets you the same (**eq**) symbol.

- *Print-Read consistency:* An interned symbol always prints as a sequence of characters that, when read back in, yields the same (**eq**) symbol.

- *Print-Print consistency:* If two interned symbols are not **eq**, then their printed representations will not be the same sequence of characters.

These consistency rules remain true in spite of any amount of implicit interning caused by typing in Lisp forms, loading files, and so on. This has the important implication that results are reproducible regardless of the order of loading files or the exact history of what symbols were typed in when. The rules can only be violated by explicit action: changing the value of **package**, forcing some action by continuing from an error, or calling a function that makes explicit modifications to the package structure (**remob**, for example).

In order to ensure that the consistency rules are obeyed, the system ensures that certain aspects of the package structure are chosen by conscious decision of the programmer, not by accidents such as which symbols happen to be typed in by a user. External symbols, the symbols that are shared between packages without being explicitly listed by the "accepting" package, must be explicitly listed by the "providing" package. No reference to a package can be made before it has been explicitly defined.

8. Name Conflicts: Packages

A fundamental invariant of the package system is that within one package any particular name can only refer to one symbol. A *name conflict* is said to occur when there is more than one candidate symbol and it is not obvious which one to choose. If the system does not always choose the same way, the read-read consistency rule would be violated. For example, some programs or data might have been read in under a certain mapping of the name to a symbol. If the mapping changes to a different symbol, then additional programs or data are read, the two programs will not access the same symbol even though they use the same name. Even if the system did always choose the same way, a name conflict is likely to result in a different mapping from names to symbols than you expected, causing programs to execute incorrectly. Therefore, any time a name conflict occurs, an error is signalled. You may continue from the error and tell the package system how to resolve the conflict.

Note that if the same symbol is accessible to a package through more than one path, for instance as an external of more than one package, or both through inheritance and through direct presence in the package, there is no name conflict. Name conflicts only occur between distinct symbols with the same name.

See the section "Shadowing Symbols". As discussed there, the creator of a package can tell the system in advance how to resolve a name conflict through the use of *shadowing*. Every package has a list of shadowing symbols. A shadowing symbol takes precedence over any other symbol of the same name that would otherwise be accessible to the package. A name conflict involving a shadowing symbol is always resolved in favor of the shadowing symbol, without signalling an error (except for one exception involving **import** described below). The **:shadow** and **:shadowing-import** options to **defpackage** and **make-package** may be used to declare shadowing symbols. The functions **shadow** and **shadowing-import** may also be used.

Name conflicts are detected when they become possible, that is, when the package structure is altered. There is no need to check for name conflicts during every name lookup. The functions **use-package**, **import**, and **export** check for name conflicts.

Using a package makes the external symbols of the package being used accessible to the using package; each of these symbols is checked for name conflicts with the symbols already accessible.

Importing a symbol adds it to the internals of a package, checking for a name conflict with an existing symbol either present in the package or accessible to it. **import** signals an error even if there is a name conflict with a shadowing symbol, because two explicit directives from you are inconsistent.

Exporting a symbol makes it accessible to all the packages that use the package from which the symbol is exported. All of these packages are checked for name conflicts. **(export** *s p***)** does **(intern-soft** *s q***)** for each package *q* in **(package-used-by-list** *p***)**. Note that in the usual case of exporting symbols only during the initial definition of a package, there will be no users of the package yet and the name-conflict checking will take no time.

intern does not need to do any name-conflict checking, because it never creates a new symbol if there is already an accessible symbol with the name given.

Note that the function **intern-local** can create a new symbol with the same name as an already accessible symbol. Nevertheless, **intern-local** does not check for name conflicts. This function is considered to be a low-level primitive and indiscriminate use of it can cause undetected name conflicts. Use **import**, **shadow**, or **shadowing-import** for normal purposes.

shadow and **shadowing-import** never signal a name-conflict error, because by calling these functions the user has specified how any possible conflict is to be resolved. **shadow** does name-conflict checking to the extent that it checks whether a distinct existing symbol with the specified name is accessible, and if so whether it is directly present in the package or inherited; in the latter case a new symbol is created to shadow it. **shadowing-import** does name-conflict checking to the extent that it checks whether a distinct existing symbol with the same name is accessible; if so it is shadowed by the new symbol, which implies that it must be **remob**ed if it was directly present in the package.

unuse-package, **unexport**, and **remob** (when the symbol being **remob**ed is not a shadowing symbol) do not need to do any name-conflict checking, because they only remove symbols from a package; they do not make any new symbols accessible.

remob of a shadowing symbol can uncover a name conflict that had previously been resolved by the shadowing. If package A uses packages B and C, A contains a shadowing symbol **x**, and B and C each contain external symbols named **x**, then **remob**ing **x** from A will reveal a name conflict between **b:x** and **c:x** if those two symbols are distinct. In this case **remob** will signal an error.

Aborting from a name-conflict error leaves the original symbol accessible. Package functions always signal name-conflict errors before making any change to the package structure. Note: when multiple changes are to be made, for example when exporting a list of symbols, it is valid for each change to be processed separately, so that aborting from a name conflict caused by the second symbol in the list will not unexport the first symbol in the list. However, aborting from a name-conflict error caused by exporting a single symbol will not leave that symbol accessible to some packages and inaccessible to others; exporting appears as an atomic operation.

Continuing from a name-conflict error offers you a chance to resolve the name conflict in favor of either of the candidates. This can involve shadowing or **remob**ing. Another possibility that is offered to the user is to merge together the conflicting symbols' values, function definitions, and property lists in the same way as **globalize**. This is useful when the conflicting symbols are not being used as objects, but only as names for functions (or variables, or flavors, for example). You are also offered the choice of simply skipping the particular package operation that would have caused a name conflict.

A name conflict in **use-package** between a symbol directly present in the using package and an external symbol of the used package may be resolved in favor of the first symbol by making it a shadowing symbol, or in favor of the second symbol by **remob**ing the first symbol from the using package. The latter resolution is dangerous if the symbol to be **remob**ed is an external symbol of the using package, since it will cease to be an external symbol.

A name conflict in **use-package** between two external symbols inherited by the using package from other packages may be resolved in favor of either symbol by importing it into the using package and making it a shadowing symbol.

A name conflict in **export** between the symbol being exported and a symbol already present in a package that would inherit the newly exported symbol may be resolved in favor of the exported symbol by **remob**ing the other one, or in favor of the already present symbol by making it a shadowing symbol.

A name conflict in **export** or **remob** due to a package inheriting two distinct symbols with the same name from two other packages may be resolved in favor of either symbol by importing it into the using package and making it a shadowing symbol, just as with **use-package**.

A name conflict in **import** between the symbol being imported and a symbol inherited from some other package may be resolved in favor of the symbol being imported by making it a shadowing symbol, or in favor of the symbol already accessible by not doing the **import**. A name conflict in **import** with a symbol already present in the package may be resolved by **remob**ing that symbol, or by not doing the **import**.

Good user-interface style dictates that **use-package** and **export**, which can cause many name conflicts simultaneously, first check for all of the name conflicts before presenting any of them to you. You may then choose to resolve all of them wholesale, or to resolve each of them individually, requiring a lot of interaction but permitting different conflicts to be resolved different ways.

9. Package Functions, Special Forms, and Variables

Packages are represented as Lisp objects. A package is a structure that contains various fields and a hash table that maps from names to symbols. Most of the structure field accessor functions for packages are only used internally by the package system and are not documented.

The **typep** function with one argument returns the symbol **package** if given a package object. **(typep** *obj* **'package)** is a predicate that is true if *obj* is a package object.

Many of the functions that operate on packages will accept either an actual package or the name of a package. A package name may be either a string or a symbol.

Many of the functions and variables associated with packages have names that begin with "**pkg-**". This naming convention is considered to be obsolescent and will eventually be phased out in favor of the Common Lisp-compatible naming convention that uses a prefix of "**package-**" on names that do not already contain the word **package**. Currently, however, only "**pkg-**" is valid.

9.1 The Current Package

package *Variable*

The value of **package** is the current package; many functions that take packages as optional arguments default to the value of **package**, including **intern** and related functions. The reader and the printer deal with printed representations that depend on the value of **package**. Hence the current package is part of the user interface and is displayed in the status line at the bottom of the screen.

It is often useful to bind **package** to a package around some code that deals with that package. The operations of loading, compiling, and editing a file all bind **package** to the package associated with the file.

pkg-goto &optional *pkg globally* *Function*

pkg may be a package or the name of a package. *pkg* is made the current package; in other words, the variable **package** is set to the package named by *pkg*. **pkg-goto** can be useful to "put the keyboard inside" a package when you are debugging.

pkg defaults to the **user** package.

If *globally* is specified non-**nil**, then **package** is set with **setq-globally** instead of **setq**. This is useful mainly in an init file, where you want to change the default package for user interaction, and a simple **setq** of

package will not work because it is bound by **load** when it loads the init file.

pkg-bind *pkg body...* *Macro*

pkg may be a package or a package name. The forms of the *body* are evaluated with the variable **package** bound to the package named by *pkg*. The values of the last form are returned.

```
Example:
(pkg-bind "zwei"
  (read-from-string function-name))
```

The difference between **pkg-bind** and a simple **let** of the variable **package** is that **pkg-bind** ensures that the new value for **package** is actually a package; it coerces package names (strings or symbols) into actual package objects.

9.2 Defining a Package

The **defpackage** special form is the preferred way to create a package. A **defpackage** form is treated as a *definition* form by the editor, hence the Edit Definition (m-.) command can find package definitions.

Typically you put a **defpackage** form in its own file, separate from the rest of a program's source code. The reason to use a separate file is that a package must be defined before it can be used. In order to compile, load, or edit your program, the package in which its symbols are to be read must already be defined. Typically the file containing the **defpackage** will be read in the **user** package, while all the rest of the files of your program will be read in your own private package.

When a large program consisting of multiple source files is maintained with the system system, one source file typically contains nothing but a **defpackage** form and a **defsystem** form. (Occasionally a few other housekeeping forms will be present.) This file is called the *system declaration file*. The packages and systems built into the initial Lisp system are defined in two files: sys:sys;pkgdcl defines all the packages while sys:sys;sysdcl defines all the systems. See the document *Maintaining Large Systems*.

In the simplest cases, where no nontrivial **defpackage** options are required, the **defpackage** form may be omitted and no separate file is required. All the information required to create your package is contained in the file attribute list of the file containing your program. See the section "Packages and Writing Programs".

The **make-package** function is available as the primitive way to create package objects.

defpackage *name options...* *Special Form*

Define a package named *name*; the name must be a symbol so that the source file name of the package can be recorded. If no package by that name already exists, a new package is created according to the specified options. If a package by that name already exists, its characteristics are altered according to the options specified. If any characteristic cannot be altered, an error will be signalled. If the existing package was defined by a different file, you will be queried before it is changed, as with any other type of definition.

Each *option* is a keyword or a list of a keyword and arguments. A keyword by itself is equivalent to a list of that keyword and one argument, **t**; this syntax really only makes sense for the **:external-only** and **:hash-inherited-symbols** keywords.

Wherever an argument is said to be a name or a package, it may be either a symbol or a string. Usually symbols are preferred, because the reader will standardize their alphabetic case and because readability is increased by not cluttering up the **defpackage** form with string quote (") characters.

None of the arguments are evaluated. The keywords allowed are:

(:nicknames *name name...***)**
The package is given these nicknames, in addition to its primary name.

(:prefix-name *name***)**
This name is used when printing a qualified name for a symbol in this package. The specified name should be one of the nicknames of the package or its primary name. If **:prefix-name** is not specified, it defaults to the shortest of the package's names (the primary name plus the nicknames).

(:use *package package...***)**
External symbols and relative name mappings of the specified packages are inherited. If this option is not specified, it defaults to **(:use global)**. To inherit nothing, specify **(:use)**.

(:shadow *name name...***)**
Symbols with the specified names are created in this package and declared to be shadowing.

(:export *name name...***)**
Symbols with the specified names are created in this package, or inherited from the packages it uses, and declared to be external.

(:import *symbol symbol...***)**
The specified symbols are imported into the package. Note that unlike **:export**, **:import** requires symbols, not names; it matters in which package this argument is read.

(:shadowing-import *symbol symbol...***)**
The same as **:import** but no name conflicts are possible; the symbols are declared to be shadowing.

(:import-from *package name name...***)**
The specified symbols are imported into the package. The symbols to be imported are obtained by looking up each *name* in *package*. This option exists primarily for system bootstrapping, since the same thing can normally be done by **:import**. The difference between **:import** and **:import-from** may be visible if the file containing a **defpackage** is compiled; when **:import** is used the symbols will be looked up at compile time, but when **:import-from** is used the symbols will be looked up at load time. If the package structure has been changed between the time the file was compiled and the time it is loaded, there may be a difference.

(:relative-names (*name package***) (***name package***)...)**
Declare relative names by which this package can refer to other packages. The package being created cannot be one of the *packages*, since it has not been created yet.

(:relative-names-for-me (*package name***) (***package name***)...)**
Declare relative names by which other packages can refer to this package. It is valid to use the name of the package being created as a *package* here; this is useful when a package has a relative name for itself.

(:size *number***)**
The number of symbols expected to be present in the package. This controls the initial size of the package's hash table. If the **:size** specification is an underestimate, there is no problem; the hash table will be expanded as necessary.

(:hash-inherited-symbols *boolean***)**
If true, inherited symbols are entered into the package's hash table to speed up symbol lookup. If false (the default), looking up a symbol in this package will search the hash table of each package it uses.

(:external-only *boolean***)**
If true, all symbols in this package will be external and the package will be locked. This feature is only used to simulate the old package system that was used before Release 5.0. See the section "External-only Packages and Locking".

(:include *package package...***)**
Any package that uses this package will also use the specified packages. Note that if the **:include** list is changed, the change will not be propagated to users of this package. This feature is only used to simulate the old package system that was used before Release 5.

(:new-symbol-function *function***)**
function is called when a new symbol is to be made present in the package. The default is **si:pkg-new-symbol** unless **:external-only** is specified. Do not specify this option unless you understand the internal details of the package system.

(:colon-mode *mode***)**
If *mode* is **:external**, qualified names mentioning this package behave differently depending on whether "**:**" or "**::**" is used, as in Common Lisp. "**:**" names access only external symbols. If *mode* is **:internal**, "**:**" names access all symbols. **:internal** is the default currently. See the section "Avoiding the Internal/External Distinction".

(:prefix-intern-function *function***)**
The function to call to convert a qualified name referencing this package with "**:**" (rather than "**::**") to a symbol. The default is **intern** unless **(:colon-mode :external)** is specified. Do not specify this option unless you understand the internal details of the package system.

make-package *name* &key *Function*

make-package is the primitive subroutine called by **defpackage**. **make-package** makes a new package and returns it. An error is signalled if the package name or nickname conflicts with an existing package. **make-package** takes the same arguments as **defpackage** except that standard **&key** syntax is used, and there is one additional keyword, **:invisible**.

When an argument is called a *name*, it may be either a symbol or a string. When an argument is called a *package*, it may be the name of the package as a symbol or a string, or the package itself.

The keyword arguments are:

:nicknames '(*name name...***)**
The package is given these nicknames, in addition to its primary name.

:prefix-name *name*
This name is used when printing a qualified name for a symbol in this package. The specified name should be one of the nicknames of the package or its primary name. If **:prefix-name** is not specified, it defaults to the shortest of the package's names (the primary name plus the nicknames).

:invisible *boolean*
If true, the package is not entered into the system's table of packages, and therefore cannot be referenced via a qualified name. This is useful if you simply want a package to use as a data structure, rather than as the package in which to write a program.

:use '(*package package...*)
External symbols and relative name mappings of the specified packages are inherited. If only a single package is to be used, the name rather than a list of the name may be passed. If no package is to be used, specify **nil**. The default value for **:use** is **global**.

:shadow '(*name name...*)
Symbols with the specified names are created in this package and declared to be shadowing.

:export '(*name name...*)
Symbols with the specified names are created in this package, or inherited from the packages it uses, and declared to be external.

:import '(*symbol symbol...*)
The specified symbols are imported into the package. Note that unlike **:export**, **:import** requires symbols, not names; it matters in which package this argument is read.

:shadowing-import '(*symbol symbol...*)
The same as **:import** but no name conflicts are possible; the symbols are declared to be shadowing.

:import-from '(*package name name...*)
The specified symbols are imported into the package. The symbols to be imported are obtained by looking up each *name* in *package*.

:relative-names '((*name package*) (*name package*)...)
Declare relative names by which this package can refer to other packages. The package being created cannot be one of the *packages*, since it has not been created yet.

:relative-names-for-me '((*package name*) (*package name*)...)
Declare relative names by which other packages can refer to this package.

:size *number*
The number of symbols expected to be present in the package. This controls the initial size of the package's hash table. If the **:size** specification is an underestimate, there is no problem; the hash table will be expanded as necessary.

:hash-inherited-symbols *boolean*
If true, inherited symbols are entered into the package's hash table to speed up symbol lookup. If false (the default), looking up a symbol in this package will search the hash table of each package it uses.

:external-only *boolean*
If true, all symbols in this package will be external and the package will be locked. This feature is only used to simulate the old package system that was used before Release 5.0. See the section "External-only Packages and Locking".

:include **'(***package package...***)**
: Any package that uses this package will also use the specified packages. Note that if the **:include** list is changed, the change will not be propagated to users of this package. This feature is only used to simulate the old package system that was used before Release 5.

:new-symbol-function *function*
: *function* is called when a new symbol is to be made present in the package. The default is **si:pkg-new-symbol** unless **:external-only** is specified. Do not specify this option unless you understand the internal details of the package system.

:colon-mode *mode*
: If *mode* is **:external**, qualified names mentioning this package behave differently depending on whether ":" or "::" is used, as in Common Lisp. ":" names access only external symbols. If *mode* is **:internal**, ":" names access all symbols. **:internal** is the default currently. See the section "Avoiding the Internal/External Distinction".

:prefix-intern-function *function*
: The function to call to convert a qualified name referencing this package with ":" (rather than "::") to a symbol. The default is **intern** unless **(:colon-mode :external)** is specified. Do not specify this option unless you understand the internal details of the package system.

pkg-kill *package* *Function*

Kill *package* by removing it from all package system data structures. The name and nicknames of *package* cease to be recognized package names. If *package* is used by other packages, it is un-used, causing its external symbols to stop being accessible to those packages. If other packages have relative names for *package*, the names are deleted.

Any symbols in *package* will still exist and their home package will not be changed. If this is undesirable, evaluate **(mapatoms #'remob** *package* **nil)** first.

package may be a package or the name of a package.

9.3 Mapping Names to Symbols

The name of a symbol is a string, corresponding to the printed representation of that symbol with quoting characters removed. Mapping the name of a symbol into the symbol itself is called *interning*, for historical reasons. Interning is only meaningful with respect to a particular package, since packages are name-to-symbol mappings. Unless a package is explicitly specified, the current package is assumed.

There are four functions for interning, named **intern, intern-soft, intern-local,** and **intern-local-soft**. Each of these functions takes two arguments and returns two values. The arguments are a name and a package. The name may be a string or a symbol. The package argument may be a package, the name of a package as a string or a symbol, or **nil** or unsupplied, in which case the current package (the value of **package**) is used by default.

The **-soft** functions will not create new symbols, but will only find existing symbols. The other two functions will add a new symbol to the package if no existing symbol with the specified name is found. When adding a new symbol, if the name argument is a string, a new symbol is created and its home package is made to be the specified package. If the name argument is a symbol, that symbol is used as the new symbol. If it has a home package, it is not changed, but if it does not have a home package its home package is set to the package to which it was just added.

The **-local** functions only look for symbols present in the package; they do not search through inherited symbols. The other two functions see all accessible symbols.

The first value is the symbol that was found or created, or **nil** if no symbol was found and a **-soft** function was called. The second value is a flag that takes on one of the following values:

nil	No preexisting symbol was found. If the function called was not a **-soft** version, a new internal symbol was added to the package.
:internal	An existing internal symbol was found to be present in the package.
:external	An existing external symbol was found to be present in the package.
:inherited	An existing symbol was found to be inherited by the package. This symbol is necessarily external in the package from which it was inherited, and cannot be external in the package being searched.

Note that the first value should not be used as a flag to detect whether or not a symbol was found, since the *false* value, **nil**, is a symbol. The second value must be used for this purpose. The **-soft** functions return both values **nil** if they do not find a symbol.

Note: interning is sensitive to case; that is, it will consider two character strings different even if the only difference is one of uppercase versus lowercase (unlike most string comparisons elsewhere in the Lisp Machine system). Symbols are converted to uppercase when you type them in because the reader converts the case of characters in the printed representation of symbols; the characters are converted to uppercase before **intern** is ever called. So if you call **intern** with a lowercase **"foo"** and then with an uppercase **"FOO"**, you will not get the same symbol.

intern *string* &optional *(pkg* **package***)* *Function*

Find or create a symbol named *string* accessible to *pkg*, either directly present in *pkg* or inherited from a package it uses.

If *string* is not a string but a symbol, **intern** searches for a symbol with the same name. If it doesn't find one, it interns *string* — rather than a newly created symbol — in *pkg* (even if it is also interned in some other package) and returns it.

intern-local *string* &optional *(pkg* **package***)* *Function*

Find or create a symbol named *string* directly present in *pkg*. Symbols inherited by *pkg* from packages it uses are not considered, thus **intern-local** can cause a name conflict. **intern-local** is considered to be a low-level primitive and indiscriminate use of it can cause undetected name conflicts. Use **import**, **shadow**, or **shadowing-import** for normal purposes.

If *string* is not a string but a symbol, and no symbol with that print name is already interned in *pkg*, **intern-local** interns *string* — rather than a newly created symbol — in *pkg* (even if it is also interned in some other package) and returns it.

intern-soft *string* &optional *(pkg* **package***)* *Function*

Find a symbol named *string* accessible to *pkg*, either directly present in *pkg* or inherited from a package it uses. If no symbol is found, the two values **nil nil** are returned.

intern-local-soft *string* &optional *(pkg* **package***)* *Function*

Find a symbol named *string* directly present in *pkg*. Symbols inherited by *pkg* from packages it uses are not considered. If no symbol is found, the two values **nil nil** are returned.

intern-local-soft is a good low-level primitive for when you want complete control of what packages to search and when to add new symbols.

find-all-symbols *string* *Function*

Search all packages for symbols named *string* and return a list of them. Duplicates are removed from the list; if a symbol is present in more than one package, it only appears once in the list. The **global** package is searched first and so global symbols will appear earlier in the list than symbols that shadow them. In general packages are searched in the order that they were created.

string may be a symbol, in which case its name is used. This is primarily for user convenience when calling **find-all-symbols** directly from the read-eval-print loop.

Invisible packages are not searched.

The **where-is** function is a more user-oriented version of **find-all-symbols**;

it returns information about *string*, rather than just a list. See the function **where-is**.

remob *symbol* &optional *package* *Function*

remob removes *symbol* from *package* (the name is historical and means "REMove from OBlist"). *symbol* itself is unaffected, but **intern** will no longer find it in *package*. Removing a symbol from its home package sets its home package to **nil**; removing a symbol from a package different from its home package leaves the symbol's home package unchanged.

remob returns **t** if the symbol was found and removed, or **nil** if it was not found.

remob is always "local", in that it removes only from the specified package and not from any other packages. Thus **remob** has no effect unless the symbol is present in the specified package, even if it is accessible from that package via inheritance.

If *package* is unspecified it defaults to the symbol's home package. Note this exception well: the default value of **remob**'s *package* argument is *not* the current package.

9.4 Home Package of a Symbol

Every symbol has a *home package*. When a new symbol is created by the reader and made present in the current package, its home package is set to the current package. The home package of a symbol may be obtained with the **symbol-package** function.

Most symbols are present only in their home package; however, it is possible to make a symbol be present in any number of packages. Only one of those packages can be distinguished as the home package; normally this will be the first package in which the symbol was present. The package system makes an effort to ensure that a symbol *is* present in its home package. When a symbol is first created by the reader (actually by the **intern** function), it is guaranteed to be present in its home package. If the symbol is removed from its home package (by the **remob** function), the home package of the symbol will be set to **nil**, even if the symbol is still present in some other package.

Some symbols are not present in any package; they are said to be *uninterned*. See the section "Mapping Names to Symbols". The **make-symbol** function can be used to create such a symbol. An uninterned symbol has no home package; the **symbol-package** function will return **nil** given such a symbol.

When a symbol is printed, for example, with **prin1**, the printer produces a printed representation that the reader will turn back into the same symbol. If the symbol is not accessible to the current package, a qualified name is printed. See the section

"Qualified Names". The symbol's home package is used as the prefix in the qualified name.

9.5 Mapping Between Names and Packages

pkg-name *package* *Function*

Get the (primary) name of a package. The name is a string.

It is an error if *package* is not a package object. (The phrase "it is an error" has special significance in Common Lisp. See the Common Lisp manual, not available from Symbolics, for more information.) Note that **pkg-name** is a structure accessing function and does not check that its argument is a package object, only that it is some kind of an array with a leader.

pkg-find-package *x* &optional *(create-p* **:error***)* *(relative-to* **nil***)* *Function*

pkg-find-package tries to interpret *x* as a package. Most of the functions whose descriptions say "... may be either a package or the name of a package" call **pkg-find-package** to interpret their package argument.

If *x* is a package, **pkg-find-package** returns it.

If *x* is a symbol or a string, it is interpreted as the name of a package. If *relative-to* is specified and non-**nil**, then it must be a package or the name of a package. If *relative-to* or one of the packages it uses has a relative name of *x*, the package named by that relative name is used. If the relative name search fails, or if no relative name search is called for (that is, *relative-to* is **nil**, which is the default), then if a package with a primary name or nickname of *x* exists it is returned.

If *x* is a list, it is presumed to have come from a file attribute line. **pkg-find-package** is done on the car of the list. If that fails, a new package is created with that name, according to the specifications in the rest of the list. See the section "Packages and Writing Programs".

If no package is found, the *create-p* argument controls what happens. Note that this can only happen if *x* is a symbol or a string. The possible values for *create-p* are:

:error or **nil**	An error is signalled. The error can be continued by defining the package manually, creating it automatically with default attributes, or using a different package name instead. **:error** is the default. **nil** is accepted as a synonym for **:error** for backwards compatibility.
:find	Just return **nil**.
:ask	Ask the user whether to create it.

t Create a package with the specified name with default attributes. It will inherit from **global** but not from any other packages.

The package name search is independent of alphabetic case. However, this may be changed in the future for Common Lisp compatibility and should not be depended upon. In any event it is not considered good style to have two distinct packages whose names differ only in alphabetic case.

9.6 Package Iteration

mapatoms *function* &optional *(package* **package***)* *(inherited-p* **t***)* *Function*

function should be a function of one argument. **mapatoms** applies *function* to each of the symbols in *package*. If *inherited-p* is **t**, this is all symbols accessible to *package*, including symbols it inherits from other packages. If *inherited-p* is **nil**, *function* only sees the symbols that are directly present in *package*.

Note that when *inherited-p* is **t** symbols that are shadowed but otherwise would have been inherited will be seen; this slight blemish is for the sake of efficiency. If this is a problem, *function* can try **intern** in *package* on each symbol it gets, and ignore the symbol if it is not **eq** to the result of **intern**; this measure is rarely needed.

mapatoms-all *function* *Function*

function should be a function of one argument. **mapatoms-all** applies *function* to all of the symbols in all of the packages in existence, except for invisible packages. Note that symbols that are present in more than one package will be seen more than once.

Example:

```
(mapatoms-all
  (function
    (lambda (x)
      (and (alphalessp 'z x)
           (print x)))))
```

do-symbols *(variable [package] [result]) body...* *Special Form*

Evaluate the *body* forms repeatedly with *variable* bound to each symbol accessible in *package*. *package* may be a package object or a string or symbol that is the name of a package, or it may be omitted, in which case the value of **package** is used by default.

When the iteration terminates, *result* is evaluated and its values are returned. The value of *variable* is **nil** during the evaluation of *result*. If *result* is not specified, the value returned is **nil**.

The **return** special form may be used to cause a premature exit from the iteration.

do-local-symbols *(variable [package] [result]) body...* *Special Form*

Evaluate the *body* forms repeatedly with *variable* bound to each symbol present in *package*. *package* may be a package object or a string or symbol that is the name of a package, or it may be omitted, in which case the value of **package** is used by default.

When the iteration terminates, *result* is evaluated and its values are returned. The value of *variable* is **nil** during the evaluation of *result*. If *result* is not specified, the value returned is **nil**.

The **return** special form may be used to cause a premature exit from the iteration.

do-external-symbols *(variable [package] [result]) body...* *Special Form*

Evaluate the *body* forms repeatedly with *variable* bound to each external symbol exported by *package*. *package* may be a package object or a string or symbol that is the name of a package, or it may be omitted, in which case the value of **package** is used by default.

When the iteration terminates, *result* is evaluated and its values are returned. The value of *variable* is **nil** during the evaluation of *result*. If *result* is not specified, the value returned is **nil**.

The **return** special form may be used to cause a premature exit from the iteration.

do-all-symbols *(variable [result]) body...* *Special Form*

Evaluate the *body* forms repeatedly with *variable* bound to each symbol present in any package (excluding invisible packages).

When the iteration terminates, *result* is evaluated and its values are returned. The value of *variable* is **nil** during the evaluation of *result*. If *result* is not specified, the value returned is **nil**.

The **return** special form may be used to cause a premature exit from the iteration.

See the section "Iteration Paths". This section contains a discussion of the **interned-symbols** and **local-interned-symbols loop** iteration paths.

9.7 Interpackage Relations

pkg-add-relative-name *from-package name to-package* *Function*

Add a relative name named *name*, a string or a symbol, that refers to *to-package*. From now on, qualified names using *name* as a prefix, when the current package is *from-package* or a package that uses *from-package*, will refer to *to-package*.

from-package and *to-package* may be packages or names of packages.

It is an error if *from-package* already defines *name* as a relative name for a package different from *to-package*.

pkg-delete-relative-name *from-package name* *Function*

If *from-package* defines *name* as a relative name, it is removed. *from-package* may be a package or the name of a package. *name* may be a symbol or a string. It is not an error if *from-package* does not define *name* as a relative name.

package-use-list *package* *Function*

The list of other packages used by the argument package. *package* may be a package object or the name of a package (a symbol or a string). The elements of the list returned are package objects.

package-used-by-list *package* *Function*

The list of other packages that use the argument package. *package* may be a package object or the name of a package (a symbol or a string). The elements of the list returned are package objects.

use-package *packages-to-use* &optional *package* *Function*

The *packages-to-use* argument should be a list of packages or package names, or a single package or package name. These packages are added to the use-list of *package* if they are not there already. All external symbols in the packages to use become accessible in *package*. *package* may be a package object or the name of a package (a symbol or a string). If unspecified, *package* defaults to the value of **package**. Returns **t**.

unuse-package *packages-to-unuse* &optional *package* *Function*

The *packages-to-unuse* argument should be a list of packages or package names, or a single package or package name. These packages are removed from the use-list of *package* and their external symbols are no longer accessible, unless they are accessible through another path. *package* may be a package object or the name of a package (a symbol or a string). If unspecified, *package* defaults to the value of **package**. Returns **t**.

9.8 Import, Export, and Shadow

(export 'bit 'horse)

symbols must already exist

export *symbols* &optional *package* *Function*

The *symbols* argument should be a list of symbols or a single symbol. These symbols become available as external symbols in *package*. *package* may be a package object or the name of a package (a symbol or a string). If unspecified, *package* defaults to the value of **package**. Returns **t**. The **:export** option to **defpackage** and **make-package** is equivalent.

unexport *symbols* &optional *package* *Function*

The argument should be a list of symbols or a single symbol. These symbols become internal symbols in *package*. *package* may be a package object or the name of a package (a symbol or a string). If unspecified, *package* defaults to the value of **package**. Returns **t**.

package-external-symbols *package* *Function*

A list of all the external symbols exported by *package*. *package* may be a package object or the name of a package (a symbol or a string).

import *symbols* &optional *package* *Function*

The argument should be a list of symbols or a single symbol. These symbols become internal symbols in *package*, and can therefore be referred to without a colon qualifier. **import** signals a correctable error if any of the imported symbols has the same name as some distinct symbol already available in the package.

package may be a package object or the name of a package (a symbol or a string). If unspecified, *package* defaults to the value of **package**. Returns **t**.

shadowing-import *symbols* &optional *package* *Function*

This is like **import**, but it does not signal an error even if the importation of a symbol would shadow some symbol already available in the package. If a distinct symbol with the same name is already present in the package, it is removed (using **remob**).

The imported symbol is placed on the shadowing-symbols list of *package*.

shadowing-import should be used with caution. It changes the state of the package system in such a way that the consistency rules do not hold across the change. *package* may be a package object or the name of a package (a symbol or a string). If unspecified, *package* defaults to the value of **package**. Returns **t**.

shadow *symbols* &optional *package* *Function*

The argument should be a list of symbols or a single symbol. The name of each symbol is extracted, and *package* is searched for a symbol of that name. If no such symbol is present in this package (directly, not by inheritance), a new symbol is created with this name and inserted in *package* as an internal symbol. The symbol is also placed on the shadowing-symbols list of *package*.

shadow should be used with caution. It changes the state of the package system in such a way that the consistency rules do not hold across the change. *package* may be a package object or the name of a package (a symbol or a string). If unspecified, *package* defaults to the value of **package**. Returns **t**.

package-shadowing-symbols *package* *Function*

The list of symbols that have been declared as shadowing symbols in this package by **shadow** or **shadowing-import**. All symbols on this list are present in the specified package. *package* may be a package object or the name of a package (a symbol or a string).

9.9 Package "Commands"

describe-package *package* *Function*

Print a description of *package*'s attributes and the size of its hash table of symbols on **standard-output**. *package* may be a package or the name of a package. The **describe** function calls **describe-package** when its argument is a package.

where-is *pname* *Function*

Find all symbols named *pname* and print on **standard-output** a description of each symbol. The symbol's home package and name are printed. If the symbol is present in a different package than its home package (that is, it has been imported), that fact is printed. A list of the packages from which the symbol is accessible is printed, in alphabetical order. **where-is** searches all packages that exist, except for invisible packages.

If *pname* is a string it is converted to uppercase, since most symbols' names use uppercase letters. If *pname* is a symbol, its exact name is used.

where-is returns a list of the symbols it found.

The **find-all-symbols** function is the primitive that does what **where-is** does without printing anything.

globalize *name* &optional *package* *Function*

Export a symbol named *name* from *package*. If this causes any name conflicts with symbols with the same name in packages that use *package*, instead of signalling an error make an attempt to resolve the name conflict automatically. Print an explanation of what is being done on **error-output**.

globalize is useful for patching up an existing package structure. For example, if a new function is added to the Lisp language **globalize** can be used to add its name to the **global** package and hence make it accessible to all packages. There might already be symbols with the desired name in existence, either by coincidence or because the function was already defined or already called. **globalize** will make all such symbols have the new function as their definition.

package may be a package or the name of a package, as a symbol or a string. It defaults to the **global** package. **globalize** is the only function that does not care whether *package* is locked.

name may be a symbol or a string. If *package* already contains a symbol by that name, that symbol is chosen. Otherwise, if *name* is a symbol, it is chosen. If *name* is a string and any of the packages that use *package* contains a nonshadowing symbol by that name, one such symbol is chosen. Otherwise, a new symbol named *name* is created. Whichever symbol is chosen this way is made present in *package* and exported from it. If the home package of the chosen symbol is a package that uses *package*, then the home package is set to *package*; in other words, the symbol is "promoted" to a "higher" package. If the home package of the chosen symbol is some other package, it is not changed. This case typically occurs when the chosen symbol is inherited by *package* from some package it uses.

The above rules for choosing a symbol to export ensure that there will be no name conflict if at all possible. If any nonshadowing symbols exist named *name* but that are distinct from the chosen symbol present in the packages that use *package*, then a name conflict occurs. **globalize** does its best to resolve the name conflict by merging together the values, function definitions, and properties of all the symbols involved. After merging, all the symbols have the same value, the same function definition, and the same properties. The value cells, function cells, and property list cells of all the symbols are forwarded to the corresponding cells of the chosen symbol, using **dtp-one-q-forward**. This ensures that any future change to one of the symbols is reflected by all of the symbols.

The merging operation simply consists of making sure that there are no conflicts. If more than one of the symbols has a value (is **boundp**), all the values must be **eql** or an error is signalled. Similarly, all the function definitions of symbols that are **fboundp** must be **eql** and all the properties with any particular indicator must be **eql**. If an error occurs the user must manually resolve it by removing the unwanted value, definition, or property (using **makunbound**, **fmakunbound**, or **remprop**) then try again.

Note that if *name* is a symbol, **globalize** attempts to use that symbol, but there is no guarantee that it will not use some other symbol. If *name* is in a package that does not use *package*, and **globalize** does not use *name* as the symbol (because there is already another symbol by that name in *package* or in some package that uses *package*), then *name* will not be merged with the chosen symbol. It is generally more predictable to use a string, rather than a symbol, for *name*.

Of course, **globalize** cannot cause two distinct symbols to become **eq**. Its conflict resolution techniques are only useful for symbols that are used as names for things like functions and variables, not for symbols that are used for their own sake. It is sometimes possible to get the desired effect by using one of the conflicting symbols as the first argument to **globalize**, rather than using a string.

For example, suppose a program in the **color** package deals with colors by symbolic names, perhaps using **selectq** to test for such symbols as **red**, **green**, and **yellow**. Suppose there is also a function named **red** in the **math** package and someone decides that this function is generally useful and should be made global. Doing **(globalize 'color:red)** ensures that the exported symbol is the one that the color program is looking for; this means that every package except the **math** package will see the right symbol to use if it wants to call the color program. Programs that call the **red** function do not care which of the two symbols they use as the name of the function, since both symbols have the same definition. Usually the situation described in this example would not arise, because standard programming style dictates that the color program should have been using keywords for this application.

globalize returns two values. The first is the chosen symbol and the second is a (possibly empty) list of all the symbols whose value, function, and property cells were forwarded to the cells of the chosen symbol.

To disable the messages printed by **globalize**, bind **error-output** to a null stream (one that throws away all output). For example:

```
(let ((error-output 'si:null-stream))
  (globalize 'rumpelstiltskin))
```

There is a subtle pitfall in the interaction between **globalize** and the binary files output by the compiler. Because of this it is best to use a string, rather than a symbol, as the argument to **globalize** in files that are to be compiled. Suppose a file contains the following form at top level:

```
(eval-when (compile load eval)
  (globalize 'si:rumpelstiltskin))
```

If the file is loaded without being compiled, the form is read and evaluated in the obvious fashion. **rumpelstiltskin** is read as the symbol by that name in the **si** package, that symbol is passed to the **globalize** function, and the symbol is moved to the **global** package. Now suppose the file is compiled. Again **rumpelstiltskin** is read as the symbol by that name in the **si** package. The **eval-when** causes the compiler first to evaluate the **globalize** form, and then to place a representation of the form into its output file. But at the time the output file is being generated, the symbol **rumpelstiltskin** is global; the compiler no longer has any way to know that it came from the **si** package. When the binary file is loaded, it will globalize the symbol **rumpelstiltskin** in the current package, not the one in the **si** package as the programmer intended. Furthermore, if at compile time there was a **rumpelstiltskin** symbol in the current package, the compile-time **globalize** will turn that symbol into a shadowing symbol. When the binary file is loaded, it will try to refer to the symbol **rumpelstiltskin** in the **global** package, which will get an error since the **global** package is locked. The same pitfall can arise without the use of **eval-when** if the file being compiled was previously loaded into the Lisp that compiled it, perhaps for test purposes.

9.10 System Packages

The following are some of the packages initially present in the Lisp world. New packages will be added to this list from time to time. The list is presented in "logical" order, with the most important or interesting packages first. A number of packages that are not of general interest have been omitted from the list for the sake of brevity.

global	Contains the global symbols of the Lisp language, including function names, variable names, special form names, and so on. All symbols in **global** are supposed to be documented. **global** does not inherit symbols from any other package.
keyword	Contains keyword symbols. **keyword** has a blank nickname so that keywords will print as **:foo** rather than **keyword:foo**. **keyword** does not inherit symbols from any other package.
user	The default package for user programs that do not have their own package. When first booted the Lisp Machine uses the **user** package to read expressions typed in by the user.
sys or **system**	Contains symbols shared among various system programs. **system** is for symbols global to the Lisp Machine "operating system", while **global** is for symbols global to the Lisp language.
si or **system-internals**	Most of the programs that implement the Lisp language and operating system are in the **system-internals** package. **system-internals** is one of the packages that uses **system**. The externally advertised symbols of these programs are in **system** or **global**. **system-internals** would not exist as a separate package from **system** if the system took advantage of the distinction between internal symbols and external symbols, but it does not yet do so.
compiler	Contains the compiler. **compiler** is one of the packages that use **system**.

dbg or **debugger**	Contains the condition system and the debugger. **debugger** is one of the packages that use **system**.
zwei	Contains the editor and Zmail.
tv	Contains the window system. **tv** is one of the packages that use **system**.
fs or **file-system**	Contains pathnames and the generic file access system. **file-system** is one of the packages that use **system**.
lmfs	Contains the Lisp Machine file storage system. **lmfs** is one of the packages that use **system**.
format	Contains the function **format** and its associated subfunctions.
net or **network**	Contains the external interfaces to the generic network system. **network** is one of the packages that uses **system**. Each network implementation and network-related program has its own package, which uses **network**.
neti or **network-internals**	Contains the programs that implement the generic network system. **network-internals** is one of the packages that use **network** and **system**.
chaos	Contains the Chaosnet control program. **chaos** is one of the packages that use **network** and **system**.
cl or **common-lisp-global**	Contains the global symbols of the Common Lisp Compatibility Package. Inside of Common Lisp this package is called **lisp**. The CLCP is not described in this documentation set. **common-lisp-global** does not use **global**.
fonts	Contains the names of all fonts. **fonts** does not inherit symbols from any other package.

The following variables have the most important packages as their values.

pkg-global-package *Variable*
The **global** package.

pkg-keyword-package *Variable*
The **keyword** package.

pkg-system-package *Variable*
The **system** package.

10. Package-related Conditions

This section documents the most basic package-related conditions. There are other conditions built on these, but most programmers should not need to deal with them.

sys:package-error *Flavor*
All package-related error conditions are built on **sys:package-error**.

sys:package-not-found *Flavor*
A package-name lookup did not find any package by the specified name.

The **:name** message returns the name. The **:relative-to** message returns **nil** if only absolute names are being searched, or else the package whose relative names are also searched.

The **:no-action** proceed type may be used to try again. The **:new-name** proceed type may be used to specify a different name or package. The **:create-package** proceed type creates the package with default characteristics.

sys:external-symbol-not-found *Flavor*
A "**:**" qualified name referenced a name that had not been exported from the specified package.

The **:string** message returns the name being referenced (no symbol by this name exists yet). The **:package** message returns the package.

The **:export** proceed type exports a symbol by that name and uses it.

sys:package-locked *Flavor*
There was an attempt to intern a symbol in a locked package.

The **:symbol** message returns the symbol. The **:package** message returns the package.

The **:no-action** proceed type interns the symbol just as if the package had not been locked. Other proceed types are also available when interning the symbol would cause a name conflict.

sys:name-conflict *Flavor*
Any sort of name conflict occurred (there are specific flavors, built on **sys:name-conflict**, for each possible type of name conflict.) The following proceed types may be available, depending on the particular error:

The **:skip** proceed type skips the operation that would cause a name conflict.

The **:shadow** proceed type prefers the symbols already present in a package to conflicting symbols that would be inherited. The preferred symbols are added to the package's shadowing-symbols list.

The **:export** proceed type prefers the symbols being exported (or being inherited due to a **use-package**) to other symbols. The conflicting symbols are **remob**'ed if they are directly present, or shadowed if they are inherited.

The **:unintern** proceed type removes the conflicting symbol (with **remob**).

The **:shadowing-import** proceed type imports one of the conflicting symbols and makes it shadow the others. The symbol to be imported is an optional argument.

The **:share** proceed type causes the conflicting symbols to share value, function, and property cells. It as if **globalize** were called.

The **:choose** proceed type pops up a window in which the user may choose between the above proceed types individually for each conflict.

11. Multipackage Programs

Usually, each independent program occupies one package. But large programs, such as Macsyma, are usually made up of a number of subprograms, and each subprogram may be maintained by a different person or group of people. We would like each subprogram to have its own name space, since the program as a whole has too many names for anyone to remember. The package system can provide the same benefits to programs that are part of the same superprogram as it does to programs that are completely independent.

Putting each subprogram into its own package is easy enough, but it is likely that there will be a fair number of functions and symbols that should be shared by all of Macsyma's subprograms. These would be internal interfaces between the different subprograms.

A package named **macsyma** can be defined and each of the internal interface symbols can be exported from it. Each subprogram of Macsyma has its own package, which uses the **macsyma** package in addition to any other packages it uses. Thus the interface symbols are accessible to all subprograms, through package inheritance. These interface symbols typically get their function definitions, variable values, and other properties from various subprograms read into the various internal Macsyma packages, although there is nothing wrong with also putting a subprogram directly into the **macsyma** package. This is similar to the way the Lisp system works; the **global** package exports a large number of symbols, which get their values, definitions, and so on from programs residing in other packages that use **global**, such as **system-internals** or **compiler**.

It is also often convenient for the **macsyma** package to supply relative names that can be used by the various subprograms to refer to each other's packages. This allows package name abbreviations to be used internally to Macsyma without contaminating the external environment.

The system declaration file for Macsyma would then look something like the following:

```
;Contains the interfaces between the various subprograms
(defpackage macsyma
            (:export meval mprint ptimes ...)
            (:colon-mode :external))    ;error-checking in qualified name

;The integration package based on the Risch algorithm
(defpackage risch
            (:use macsyma global))

;The integration package based on pattern matching
(defpackage sin
            (:use macsyma global))

;Interface to the operating system.  This uses the SYSTEM package
;because it needs many system-dependent functions and constants.
;This package also has a local nickname because its primary name
;is so long.
(defpackage macsyma-system-interface
            (:relative-names-for-me (macsyma sysi))
            (:use macsyma system global))
```

It is possible to break the interface symbols down into separate categories. For instance, it might be desirable to separate internal symbols only used inside of Macsyma from symbols that are also useful to the outside world. The latter symbols clearly should be externals of the **macsyma** package. You could create an additional package named **macsyma-internals** that exports all the symbols that are interfaces between different subprograms of Macsyma but are not for use by the outside world. In this case we would have:

```
(defpackage risch
            (:use macsyma-internals macsyma global))
```

A program in the outside world that needed to use parts of Macsyma would either use qualified names such as **macsyma:solve** or would include **macsyma** in the **:use** option in its package definition.

The interface symbols can be broken down into even more categories. Each subpackage can have its own list of exported symbols, and can use whichever other subpackages it depends on. The subset of these exported symbols that are also useful to the outside world can be exported from the **macsyma** package as well. In this case our example system declaration file would look something like:

```
;Contains the interfaces between the various subprograms
(defpackage macsyma
            (:export solve integrate ...)
            (:colon-mode :external))    ;error-checking in qualified name

;The rational function package
(defpackage rat
            (:export ptimes ...)
            (:use macsyma global))

;The integration package
(defpackage risch
            (:export integrate)
            (:use rat macsyma global))

;The Macsyma interpreter
(defpackage meval
            (:export meval mprint ...))
```

The symbol **integrate** exported by the **macsyma** package and the symbol **integrate** exported by the **risch** package are the same symbol, because **risch** inherits it from **macsyma**.

Sometimes one can get involved in forward references when setting up this sort of package structure. In the above example, **risch** needs to use **rat**, hence **rat** was defined first. If **rat** also needed to use **risch**, there would be no way to write the package definitions using only **defpackage**. In this case one can explicitly call **use-package** after both packages have been defined. For example:

```
;The rational function package
(defpackage rat
            (:export ptimes ...)
            (:use macsyma global))      ;also uses risch

;The integration package
(defpackage risch
            (:export integrate)
            (:use rat macsyma global))

;Now complete the forward references
(use-package 'risch 'rat)
```

An analogous issue arises when using **:import**.

Now, the **risch** program and the **sin** program both do integration, and so it would be natural for each to have a function called **integrate**. From inside **sin**, **sin**'s **integrate** would be referred to as **integrate** (no prefix needed), while **risch**'s would be referred to as **risch::integrate** or as **risch:integrate** if **risch** exported it (which is likely). Similarly, from inside **risch, risch**'s own **integrate** would be called **integrate**, whereas **sin**'s would be referred to as **sin::integrate** or **sin:integrate**.

If **sin**'s **integrate** were a recursive function, you would refer to it from within **sin** itself, and would not have to type **sin:integrate** every time; you would just say **integrate**.

If the names **sin** and **risch** are considered to be too short to use up in the general space of package names, they can be made local abbreviations within Macsyma's family of package through local names. The package definitions would be

```
;Contains the interfaces between the various subprograms
(defpackage macsyma
            (:export meval mprint ptimes ...)
            (:colon-mode :external))    ;error-checking in qualified name

;The integration package based on the Risch algorithm
(defpackage macsyma-risch-integration
            (:relative-names-for-me (macsyma risch))
            (:use macsyma global))

;The integration package based on pattern matching
(defpackage macsyma-pattern-integration
            (:relative-names-for-me (macsyma sin))
            (:use macsyma global))
```

From inside the **macsyma** package or any package that uses it the two integration functions would be referred to as **sin:integrate** and as **risch:integrate**. From anywhere else in the hierarchy, they could be called **macsyma:sin:integrate** and **macsyma:risch:integrate**, or **macsyma-pattern-integration:integrate** and **macsyma-risch-integration:integrate**.

12. Compatibility with the Old Package System

The package system used before Release 5.0 used a hierarchical arrangement of packages and used **package-declare** rather than **defpackage** to create packages. Most users will not see any change between the old and new package systems, since the same function names continue to work and most of the old functionality can be simulated. All programs do need to be recompiled, however, because old assumptions built into the compiled code — such as where keyword symbols reside and what the indices of fields in the package structure are — are no longer valid.

If **pack1** was a subpackage of **pack2** in the hierarchical package system, then in the current system **pack1** should use **pack2** and **pack2** should be declared external-only so that all of its symbols will be inherited by **pack1**. Relative names follow the package use relations just as refnames used to follow the subpackage relations.

package-declare *name superior size [file-alist] clause...* *Special Form*

This special form exists only for compatibility with the pre-Release-5 package system. **defpackage** should be used instead.

name is the name of the package to be created. It must be a string or a symbol (a list is no longer acceptable).

superior is used as the **:use** option to **defpackage**.

size is used as the **:size** option to **defpackage**.

file-alist must be **nil**; this feature has been obsolete for several years. It may be omitted if there are no clauses.

Each *clause* is a list whose first element is one of the following symbols and whose remaining elements are "arguments". It makes no difference in what package the symbols are read, since only their names are used.

borrow	Used as the **:import-from** option to **defpackage**.
intern	Used as the **:export** option to **defpackage**.
shadow	Used as the **:shadow** option to **defpackage**.
refname	Used as an element of the **:relative-names** option to **defpackage**. Note that this clause is usually unnecessary in the current package system, since package naming works more rationally.
myrefname	Used as an element of the **:relative-names-for-me** option to **defpackage**, unless the first argument is **global**, in which case it is used as an element of the **:nicknames** option. Note that this clause is usually unnecessary in the current package system, since package naming works more rationally.

The **use, external, advertise, forward, forward-alias, indirect, indirect-alias, keyword,** and **subpackage** clauses that **package-declare** used to accept cannot be simulated and are no longer allowed. None of these were documented and some of them did not work.

pkg-create-package *name* &optional *superior size* *Function*

This function exists only for compatibility with the pre-Release-5 package system. **make-package** should be used instead.

name must be a symbol or a string; lists are no longer accepted. *superior* is used as the **:use** argument to **make-package**. If *superior* is **nil** then **:invisible t** is specified. *size* is used as the **:size** argument to **make-package**.

The *dont-lock-superior* argument no longer exists. Package locking is now controlled explicitly by the **:external-only** option to **defpackage** and **make-package**.

The global functions **pkg-contained-in, pkg-debug-copy, pkg-load, pkg-refname-alist,** and **pkg-super-package** no longer exist. The first three of these were not documented.

The functions **intern, intern-local, intern-soft,** and **intern-local-soft** no longer return three values. Now only two values are returned. The second value is different but upward-compatible.

The functions **mapatoms-all** and **where-is** no longer take an optional argument defaulting to the **global** package. They now always process all packages that are not invisible. The function **package-used-by-list** can help if you need to process only the subset of all packages that use some particular package.

12.1 External-only Packages and Locking

The facilities described in this section are primarily for compatibility with the old, hierarchical package system used before Release 5. Full use of these facilities requires knowing about functions that are in the **si** package and not described in this document.

A package can be *locked*, which means that any attempt to add a new symbol to it will signal an error. Continuing from the error will add the symbol.

When reading from an interactive stream, such as a window, the error for adding a new symbol to a locked package does not go into the debugger. Instead it asks you to correct your input, using the input editor. You cannot add a new symbol to a locked package just by typing its name; you must explicitly call **intern, export,** or **globalize**.

A package can be declared *external-only*. This causes any symbol added to the package to be exported automatically. Since exporting of symbols should be a conscious decision, when you create an external-only package it is automatically locked. Any attempt to add a new symbol to an external-only package signals an error because it is locked. If adding the symbol would cause a name conflict in some package that uses the package to which the symbol is being added, the error message mentions that fact. Continuing from the error will add the symbol anyway. In the event of name conflicts, appropriate proceed types for resolving name conflicts are offered.

To set up an external-only package, it can be temporarily unlocked and then the desired set of symbols can be interned in it. Unlocking an external-only package disables name-conflict checking, since the system (perhaps erroneously) assumes you know what you are doing. The **global** package is external-only and locked. Its contents are initialized when the system is built by reading files containing the desired symbols with **package** bound to the **global** package object, which is temporarily unlocked. The **system** package is external-only, locked, and initialized the same way.

Index