DAVID MOON
RICHARD M. STALLMAN
DANIEL WEINREB

LISP
CHINE
NUAL

MA
M

Lisp Machine Manual

Sixth Edition, System Version 99

June 1984

Richard Stallman
Daniel Weinreb
David Moon

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research Contract number N00014-80-C-0505.

Preface

The Lisp Machine manual describes both the language and the operating system of the Lisp Machine. The language, a dialect of Lisp called Zetalisp, is completely documented by this manual. The software environment and operating-system-like parts of the system contain many things which are still in a state of flux. This manual confines itself primarily to the stabler parts of the system. It describes how to program, but not for the most part how to operate the machine. The window system is documented separately in the Lisp Machine Window System manual.

Any comments, suggestions, or criticisms will be welcomed. Please send Arpa network mail to BUG-LMMAN@MIT-MC.

Those not on the Arpanet may send U.S. mail to
Richard M. Stallman
Artificial Intelligence Lab
545 Technology Square
Cambridge, Mass. 02139

Portions of this manual were written by Mike McMahon and Alan Bawden. The chapter on the LOOP iteration macro is mostly a reprint of Laboratory for Computer Science memo TM-169, by Glenn Burke. Sarah Smith, Meryl Cohen and Richard Ingria of LMI, and Richard Mlynarik of MIT, helped to correct the manual.

Personal Note from Richard Stallman

The Lisp Machine is a product of the efforts of many people too numerous to list here and of the former unique unbureaucratic, free-wheeling and cooperative environment of the M.I.T. Artificial Intelligence Laboratory. I believe that the commercialization of computer software has harmed the spirit which enabled such systems to be developed. Now I am attempting to build a software-sharing movement to revive that spirit from near oblivion.

Since January 1984 I have been working primarily on the development of GNU, a complete Unix-compatible software system for standard hardware architectures, to be shared freely with everyone just like EMACS. This will enable people to use computers and be good neighbors legally (a good neighbor allows his neighbors to copy any generally useful software he has a copy of). This project has inspired a growing movement of enthusiastic supporters. Just recently the first free portable C compiler compiled itself. If you would like to contribute to GNU, write to me at the address above. Restrain social decay—help get programmers sharing again.

Summary Table of Contents

Table of Contents

1. Introduction

1.1 General Information

The Lisp Machine is a new computer system designed to provide a high-performance and economical implementation of the Lisp language. It is a personal computation system, which means that processors and main memories are not time-multiplexed: when using a Lisp Machine, you get your own processor and memory system for the duration of the session. It is designed this way to relieve the problems of running large Lisp programs on time-sharing systems. Everything on the Lisp Machine is written in Lisp, including all system programs; there is never any need to program in machine language. The system is highly interactive.

The Lisp Machine executes a new dialect of Lisp called Zetalisp, developed at the M.I.T. Artificial Intelligence Laboratory for use in artificial intelligence research and related fields. It was originally based on the Maclisp dialect, and attempts to maintain a good degree of compatibility with Maclisp, while also providing many improvements and new features. Maclisp, in turn, was based on Lisp 1.5.

Common Lisp is a Lisp dialect designed to standardize all the various Lisp systems derived from Maclisp. Zetalisp today is nearly a superset of Common Lisp, but there are a few important incompatibilities between them, in places where Common Lisp involves an incompatible change which is deemed to severe to impose on traditional Zetalisp users. There is a special mode which provides strict Common Lisp compatibility. See section 1.4, page 7 for more information.

This document is the reference manual for the Zetalisp language. This document is not a tutorial, and it sometimes refers to functions and concepts that are not explained until later in the manual. It is assumed that you have a basic working knowledge of some Lisp dialect; you will be able to figure out the rest of the language from this manual.

There are also facilities explained in this manual that are not really part of the Lisp language. Some of these are subroutine packages of general use, and others are tools used in writing programs. The Lisp Machine window system and the major utility programs are, or ought to be, documented in other manuals.

1.2 Structure of the Manual

The manual starts out with an explanation of the language. Chapter 2 explains the different primitive types of Lisp object and presents some basic *predicate* functions for testing types. Chapter 3 explains the process of evaluation, which is the heart of the Lisp language. Chapter 4 introduces the basic Lisp control structures.

The next several chapters explain the details of the various primitive data-types of the language and the functions that deal with them. Chapter 5 deals with conses and the higher-level structures that can be built out of them, such as trees, lists, association lists, and property lists. Chapter 6 deals with symbols, chapter 7 with the various kinds of numbers, and chapter 8 with arrays. Chapter 10 explains character strings, which are a special kind of array.

After this there are some chapters that explain more about functions, function-calling, and related matters. Chapter 11 presents all the kinds of functions in the language, explains function-specs, and tells how to manipulate definitions of functions. Chapters 12 and 13 discuss closures and stack-groups, two facilities useful for creating coroutines and other advanced control and access structures.

Next, a few lower-level issues are dealt with. Chapter 14 explains locatives, which are a kind of pointer to memory cells. Chapter 15 explains the "subprimitive" functions, which are primarily useful for implementation of the Lisp language itself and the Lisp Machine's operating system. Chapter 16 discusses areas, which give you control over storage allocation and locality of reference.

Chapter 17 discusses the Lisp compiler, which converts Lisp programs into "machine language" or "macrocode". Chapter 18 explains the Lisp macro facility, which allows users to write their own extensions to Lisp, extending both the interpreter and the compiler. The next two chapters go into detail about two such extensions, one that provides a powerful iteration control structure (chapter 19), and one that provides a powerful data structure facility (chapter 20).

Chapter 21 documents flavors, a language facility to provide generic functions using the paradigm used in Smalltalk and related languages, called "object-oriented programming" or "message passing". Flavors are widely used by the system programs of the Lisp Machine, as well as being available to the user as a language feature.

The next few chapters discuss I/O: chapter 22 explains I/O streams and character and line level operations; chapter 23 explains reading and printing symbolic expressions; chapter 24 explains naming of files; chapter 25 explains input and output to files. Chapter 26 describes the use of the Chaosnet.

Chapter 27 describes the *package* system, which allows many name spaces within a single Lisp environment. Chapter 28 documents the "system" facility that helps you create and maintain systems, which are programs that reside in many files.

Chapter 29 discusses the facilities for multiple processes and how to write programs that use concurrent computation. Chapter 30 explains how exceptional conditions (errors) can be handled by programs, handled by users, and debugged. Chapter 31 explains the instruction set of the Lisp Machine and tells you how to examine the output of the compiler. Chapter 32 documents some functions for querying the user, chapter 34 explains some functions for manipulating dates and times, and chapter 35 contains other miscellaneous functions and facilities.

1.3 Notational Conventions and Helpful Notes

There are several conventions of notation and various points that should be understood before reading the manual. This section explains those conventions.

The symbol '=>' is used to indicate evaluation in examples. Thus, when you see '**foo** => **nil**', this means that "the result of evaluating **foo** is (or would have been) **nil**".

The symbol '==>' is used to indicate macro expansion in examples. This, when you see '**(foo bar)** ==> **(aref bar 0)**', this means that "the result of macro-expanding **(foo bar)** is (or would have been) **(aref bar 0)**".

A typical description of a Lisp function looks like this:

function-name *arg1 arg2* &optional *arg3* (*arg4 arg2*)

The **function-name** function adds together *arg1* and *arg2*, and then multiplies the result by *arg3*. If *arg3* is not provided, the multiplication isn't done. **function-name** then returns a list whose first element is this result and whose second element is *arg4*. Examples:

```
(function-name 3 4) => (7 4)
(function-name 1 2 2 'bar) => (6 bar)
```

Note the use of fonts (typefaces). The name of the function is in bold-face in the first line of the description, and the arguments are in italics. Within the text, printed representations of Lisp objects are in a different bold-face font, as in **(+ foo 56)**, and argument references are italicized, as in *arg1* and *arg2*. A different, fixed-width font, as in `function-name`, is used for Lisp examples that are set off from the text. Other font conventions are that filenames are in bold-face, all upper case (as in **SYS: SYS; SYSDCL LISP**) while keys on the keyboard are in bold-face and capitalized (as in **Help**, **Return** and **Meta**).

'Car', 'cdr' and 'cons' are in bold-face when the actual Lisp objects are being mentioned, but in the normal text font when used as words.

The word '&optional' in the list of arguments tells you that all of the arguments past this point are optional. The default value can be specified explicitly, as with *arg4* whose default value is the result of evaluating the form **(foo 3)**. If no default value is specified, it is the symbol **nil**. This syntax is used in lambda-lists in the language, which are explained in section 3.3, page 38. Argument lists may also contain '&rest' and '&key' to indicate rest and keyword arguments.

The descriptions of special forms and macros look like this:

do-three-times *form* *Special form*

This evaluates *form* three times and returns the result of the third evaluation.

with-foo-bound-to-nil *form...* *Macro*

This evaluates the *forms* with the symbol **foo** bound to **nil**. It expands as follows:

```
(with-foo-bound-to-nil
    form1
    form2 ...) ==>
(let ((foo nil))
    form1
    form2 ...)
```

Since special forms and macros are the mechanism by which the syntax of Lisp is extended, their descriptions must describe both their syntax and their semantics; functions follow a simple consistent set of rules, but each special form is idiosyncratic. The syntax is displayed on the first line of the description using the following conventions. Italicized words are names of parts of the form which are referred to in the descriptive text. They are not arguments, even though they resemble the italicized words in the first line of a function description. Parentheses ('(' and ')') stand for themselves. Square brackets ('[' and ']') indicate that what they enclose is optional. Ellipses ('...') indicate that the subform (italicized word or parenthesized list) that precedes them may be repeated any number of times (possibly no times at all). Curly brackets followed by ellipses ('{' and '}...') indicate that what they enclose may be repeated any number of times. Thus the first line of the description of a special form is a "template" for what an instance of that special form would look like, with the surrounding parentheses removed. The syntax of some special forms is sufficiently complicated that it does not fit comfortably into this style; the first line of the description of such a special form contains only the name, and the syntax is given by example in the body of the description.

The semantics of a special form includes not only what it "does for a living", but also which subforms are evaluated and what the returned value is. Usually this will be clarified with one or more examples.

A convention used by many special forms is that all of their subforms after the first few are described as '*body...*'. This means that the remaining subforms constitute the "body" of this special form; they are Lisp forms that are evaluated one after another in some environment established by the special form.

This ridiculous special form exhibits all of the syntactic features:

twiddle-frob [(*frob option...*)] {*parameter value*}... *Special form*

This twiddles the parameters of *frob*, which defaults to **default-frob** if not specified. Each *parameter* is the name of one of the adjustable parameters of a frob; each *value* is what value to set that parameter to. Any number of *parameter/value* pairs may be specified. If any *options* are specified, they are keywords that select which safety checks to override while twiddling the parameters. If neither *frob* nor any *options* are specified, the list of them may be omitted and the form may begin directly with the first *parameter* name.

frob and the *values* are evaluated; the *parameters* and *options* are syntactic keywords and not evaluated. The returned value is the frob whose parameters were adjusted. An error is signaled if any safety checks are violated.

Operations, the message-passing equivalent of ordinary Lisp's functions, are described in this style:

:operation-name *arg1 arg2* &optional *arg3* *Operation on* flavor-name
> This is the documentation of the effect of performing operation :operation-name (or, sending a message named :operation-name), with arguments *arg1*, *arg2*, and *arg3*, on an instance of flavor **flavor-name**.

Descriptions of variables ("globally special" variables) look like this:

typical-variable *Variable*
> The variable typical-variable has a typical value....

If the description says 'Constant' rather than 'Variable', it means that the value is never set by the system and should not be set by you. In some cases the value is an array or other structure whose *contents* may be changed by the system or by you.

Most numbers in this manual are decimal; octal numbers are labelled as such, using #o if they appear in examples. Currently the default radix for the Lisp Machine system is eight, but this will be changed in the near future. If you wish to change to base ten now, see the documentation on the variables *read-base* and *print-base* (page 517).

All uses of the phrase 'Lisp reader', unless further qualified, refer to the part of Lisp that reads characters from I/O streams (the read function), and not the person reading this manual.

There are several terms that are used widely in other references on Lisp, but are not used much in this document since they have become largely obsolete and misleading. For the benefit of those who may have seen them before, they are: 's-expression', which means a Lisp object; 'dotted pair', which means a cons; and 'atom', which means, roughly, symbols and numbers and sometimes other things, but not conses. The terms 'list' and 'tree' are defined in chapter 5, page 86.

The characters acute accent (') (also called "single quote") and semicolon (';') have special meanings when typed to Lisp; they are examples of what are called *macro characters*. Though the mechanism of macro characters is not of immediate interest to the new user, it is important to understand the effect of these two, which are used in the examples.

When the Lisp reader encounters a " ' ", it reads in the next Lisp object and encloses it in a quote special form. That is, 'foo-symbol turns into (quote foo-symbol), and '(cons 'a 'b) turns into (quote (cons (quote a) (quote b))). The reason for this is that quote would otherwise have to be typed in very frequently, and would look ugly.

The semicolon is used as a commenting character. When the Lisp reader sees one, the remainder of the line is discarded.

The character '/' is used for quoting strange characters so that they are not interpreted in their usual way by the Lisp reader, but rather are treated the way normal alphabetic characters are treated. So, for example, in order to give a '/' to the reader, you must type '//', the first '/' quoting the second one. When a character is preceded by a '/' it is said to be *escaped*. Escaping also turns off the effects of macro characters such as " ' " and ';'.

If you select Common Lisp syntax, escaping is done with '\' instead, and '/' has no special syntactic significance. The manual uses traditional syntax throughout, however.

The following characters also have special meanings and may not be used in symbols without escaping. These characters are explained in detail in the section on printed representation (section 23.3, page 516).

"	Double-quote delimits character strings.
#	Sharp-sign introduces miscellaneous reader macros.
`	Backquote is used to construct list structure.
,	Comma is used in conjunction with backquote.
:	Colon is the package prefix.
|	Characters between pairs of vertical-bars are escaped.
⊗	Circle-cross lets you type in characters using their octal codes.

All Lisp code in this manual is written in lower case. In fact, the reader turns all symbols into upper case, and consequently everything prints out in upper case. You may write programs in whichever case you prefer.

You will see various symbols that have the colon (:) character in their names. The colon and the characters preceding it are not actually part of the symbol name, but in early stages of learning the system you can pretend that they are. Actually they are a package prefix. See chapter 27 for an explanation of packages and what package prefixes really do.

Symbols whose names start with si: are internal to the system. These functions and variables are documented here because they are things you sometimes need to know about. However, they are subject to change with little concern for compatibility for users.

Zetalisp is descended from Maclisp, and a good deal of effort was expended to try to allow Maclisp programs to run in Zetalisp. Throughout the manual, there are notes about differences between the dialects. For the new user, it is important to note that many functions herein exist solely for Maclisp compatibility; they should *not* be used in new programs. Such functions are clearly marked in the text.

The Lisp Machine character set is not quite the same as that used on I.T.S. nor on Multics; it is described in full detail in section 10.1.1, page 205. The important thing to note for now is that the character "newline" is the same as Return, and is represented by the number 215 octal. (This number should *not* be built into any programs.)

When the text speaks of "typing Control-Q" (for example), this means to hold down the Control key on the keyboard (either of the two keys labeled 'CTRL'), and, while holding it down, to strike the Q key. Similarly, to type Meta-P, hold down either of the Meta keys and strike P. To type Control-Meta-T hold down both Control and Meta. Unlike ASCII, the Lisp machine character set does not simply label a few of the characters as "control characters"; Control and Meta (and Super and Hyper) are modifiers that can be attached to any character and are represented as separate bits. These modifier bits are not present in characters in strings or files.

Many of the functions refer to "areas". The *area* feature is of interest only to writers of large systems and can be safely disregarded by the casual user. It is described in chapter 16.

1.4 Common Lisp Support

Common Lisp is the name of a standardization project whose goal was to establish a compatible subset for Lisp systems descended from Maclisp.

Originally it was hoped that Zetalisp and the Lisp Machine system could be changed to become a superset of Common Lisp; but this proved impossible because the final Common Lisp design includes several incompatible changes to widely used functions, which, while of no fundamental importance, would make most user programs fail to work. Therefore it was necessary to make Common Lisp a separate mode of operation. The incompatibilities fall into two classes:

* Read syntax: Common Lisp specifies '\' as the single-character escape character rather than the traditional '/'. A few other constructs, such as character objects and complex numbers, are also written incompatibly.
* Specific functions: many Lisp functions of ancient pedigree, including **member**, **assoc**, **subst**, **union**, **terpri**, **close** and **//** are specified to be incompatible with their traditional behavior.

The read syntax incompatibilities have been dealt with by having separate readtables for traditional and Common Lisp syntax. The incompatibilities in functions have been dealt with by means of *reader symbol substitutions*. For each function changed incompatibly, such as **member**, a new, distinct symbol exists in a package called **cli** ("Common Lisp Incompatible"); for example, **cli:member**. The function definition of the symbol **member** is the traditional definition, while that of **cli:member** is the Common Lisp definition. In Common Lisp programs, the reader is directed to replace **member** with **cli:member** wherever it is seen. So traditional and Common Lisp programs both get the **member** functions they expect. Programs written in traditional syntax can refer to the new **cli** functions with explicit **cli:** package prefixes. Programs written in Common Lisp syntax can refer to the traditional symbols with explicit **global:** package prefixes, but this is not expected to be necessary in code.

The symbol replacements are under control of the current readtable, so that the Common Lisp readtable is responsible for causing **cli:close** to replace **close** and so on.

In this manual, the incompatible Common Lisp functions are documented under names starting with **cli:**, the names by which a traditional program could refer to them. Keep in mind that, in Common Lisp programs, the **cli:** would be omitted. A list of symbols which have incompatible Common Lisp substitutes can be found by looking up **cli:** in the function and variable indices.

Traditional read syntax is used nearly everywhere in the manual. This includes the use of '/' as an escape character, the escaping of '/' itself, and not escaping the character '\', which in traditional syntax is not special. It is up to the user to make appropriate modifications to express the same Lisp object in Common Lisp syntax when necessary.

The majority of Common Lisp changes, those that are upward compatible, have been incorporated directly into Zetalisp and are documented in this manual with no special notice.

Common Lisp read syntax and function definitions may be used either in files or interactively.

For listen loops, including Lisp Listener windows, break loops and the debugger, the choice of syntax and function semantics is made by setting the variable **readtable** to the appropriate readtable (see page 536) or most simply by calling the function **common-lisp**.

common-lisp *flag*

If *flag* is **t**, selects Common Lisp syntax and function definitions. If *flag* is **nil**, selects traditional syntax and function definitions.

In either case, this controls the reading of the following expressions that you type in the same process. It works by setting **readtable**.

In a file, Common Lisp is requested by writing the attribute **Readtable: Common-Lisp;** in the `-*-` file's line. This controls both loading or compiling the file and evaluation or compilation in the editor while visiting the file. **Readtable: Traditional;** specifies the use of traditional syntax and function definitions. If neither attribute is present, the file is processed using whatever syntax is selected in the process that loads it. See section 25.5, page 594.

Reading and printing done by programs are controlled by the same things that control reading of programs. They can also be controlled explicitly by binding the variable **readtable**.

2. Primitive Object Types

2.1 Data Types

This section enumerates some of the various different primitive types of objects in Zetalisp. The types explained below include symbols, conses, various types of numbers, two kinds of compiled code objects, locatives, arrays, stack groups, and closures.

A *symbol* (these are sometimes called "atoms" or "atomic symbols" by other texts) has a *print name*, a *value*, a *definition*, a *property list*, and a *package*.

The print name is a string, which may be obtained by the function symbol-name (page 132). This string serves as the *printed representation* (see section 23.1, page 506) of the symbol.

Each symbol has a *value*, which may be any Lisp object. This is the value of the symbol when regarded as a dynamic variable. It is also referred to sometimes as the "contents of the value cell", since internally every symbol has a cell called the *value cell*, which holds the value. It is accessed by the symeval function (page 129), and updated by the set function (page 129). (That is, given a symbol, you use symeval to find out what its value is, and use set to change its value.)

Each symbol has a *definition*, which may also be any Lisp object. It is also referred to as the "contents of the function cell", since internally every symbol has a cell called the *function cell*, which holds the definition. The definition can be accessed by the fsymeval function (page 130), and updated with fset (page 130), although usually the functions fdefinition and fdefine are employed (page 239).

The property list is a list of an even number of elements; it can be accessed directly by plist (page 131), and updated directly by setplist (page 131), although usually the functions get, putprop, and remprop (page 114) are used. The property list is used to associate any number of additional attributes with a symbol—attributes not used frequently enough to deserve their own cells as the value and definition do.

Symbols also have a package cell, which indicates which package of names the symbol belongs to. This is explained further in the section on packages (chapter 27) and can be disregarded by the casual user.

The primitive function for creating symbols is make-symbol (page 133), although most symbols are created by read, intern, or fasload (which call make-symbol themselves.)

A *cons* is an object that cares about two other objects, arbitrarily named the *car* and the *cdr*. These objects can be accessed with car and cdr (page 87), and updated with rplaca and rplacd (page 89). The primitive function for creating conses is cons (page 87).

There are several kinds of numbers in Zetalisp. *Fixnums* represent integers in the range of -2↑24 to 2↑24-1. *Bignums* represent integers of arbitrary size, but they are more expensive to use than fixnums because they occupy storage and are slower. The system automatically converts between fixnums and bignums as required. *Floats* are floating-point numbers. *Short floats* are another kind of floating-point numbers, with less range and precision, but less computational overhead. *Ratios* are exact rational numbers that are represented with a numerator and a denominator, which are integers. *Complexnums* are numbers that have explicitly represented real and imaginary parts, which can be any real numbers of the same type. See chapter 7, page 135 for full details of these types and the conversions between them.

A *character object* is much like a fixnum except that its type is distinguishable. Common Lisp programs use character objects to represent characters. Traditional programs usually use fixnums to represent characters, although they can create an manipulate character objects when they desire. Character objects behave like fixnums when used in arithmetic; only a few operations make any distinction. They do, however, print distinctively. See section 10.1, page 204 for more information.

The usual form of compiled, executable code is a Lisp object, called a "Function Entry Frame" or "FEF" for historical reasons. A FEF contains the code for one function. This is analogous to what Maclisp calls a "subr pointer". FEFs are produced by the Lisp Compiler (chapter 17, page 301), and are usually found as the definitions of symbols. The printed representation of a FEF includes its name so that it can be identified.

Another kind of Lisp object that represents executable code is a "microcode entry". These are the microcoded primitive functions of the Lisp system, and any user functions compiled into microcode.

About the only useful thing to do with any of these compiled code objects is to *apply* it to arguments. However, some functions are provided for examining such objects, for user convenience. See **arglist** (page 242), **args-info** (page 243), **describe** (page 791), and **disassemble** (page 792).

A *locative* (see chapter 14, page 267) is a kind of a pointer to a single memory cell anywhere in the system. The contents of this cell can be accessed by **cdr** (see page 87) and updated by **rplacd** (see page 89).

An *array* (see chapter 8, page 162) is a set of cells indexed by a tuple of integer subscripts. The contents of the cells may be accessed and changed individually. There are several types of arrays. Some have cells that may contain any object, while others (numeric arrays) may only contain small positive numbers. Strings are a type of array; the elements are character objects.

A *list* is not a primitive data type, but rather a data structure made up out of conses and the symbol **nil**. See chapter 5, page 86.

2.2 Data Type Predicates

A *predicate* is a function that tests for some condition involving its arguments and returns the symbol t if the condition is true, or the symbol nil if it is not true. The following predicates are for testing what data type an object has.

By convention, the names of predicates usually end in the letter 'p' (which stands for 'predicate').

The following predicates are for testing data types. These predicates return t if the argument is of the type indicated by the name of the function, nil if it is of some other type.

symbolp *object*
t if *object* is a symbol, otherwise nil.

nsymbolp *object*
nil if *object* is a symbol, otherwise t.

listp *object*
t if *object* is a cons, otherwise nil. Note that this means (listp nil) is nil even though nil is the empty list.

[This may be changed in the future to work like cli:listp. Since the current definition of listp is identical to that of consp, all uses of listp should be changed to consp unless the treatment of nil is not of concern.]

cli:listp *object*
The Common Lisp version of listp returns t if *object* is nil or a cons.

nlistp *object*
t if *object* is anything besides a cons, otherwise nil. (nlistp nil) returns t.

[This may be changed in the future, if and when listp is changed. Since the current definition of nlistp is identical to that of atom, all uses of nlistp should be changed to atom unless the treatment of nil is not of concern.]

atom *object*
t if *object* is not a cons, otherwise nil. This is the same as (not (consp *object*)).

consp *object*
t if *object* is a cons, otherwise nil. At the moment, this is the same as listp; but while listp may be changed, consp will *never* be true of nil.

numberp *object*
t if *object* is any kind of number, otherwise nil.

integerp *object*
fixp *object*
Return **t** if *object* is a representation of an integer, i.e. a fixnum or a bignum, otherwise **nil**.

floatp *object*
t if *object* is a floating-point number, i.e. a full-size or short float, otherwise **nil**.

fixnump *object*
t if *object* is a fixnum, otherwise **nil**.

bigp *object*
t if *object* is a bignum, otherwise **nil**.

flonump *object*
t if *object* is a full-size float, otherwise **nil**.

small-floatp *object*
t if *object* is a short float, otherwise **nil**.

rationalp *object*
t if *object* is an exact representation of a rational number; that is, if it is a fixnum, a bignum or a ratio. Otherwise **nil**.

complexp *object*
t if *object* is a complexnum, a number explicitly represented as complex. Otherwise **nil**.

realp *object*
t if *object* is a number whose value is real, otherwise **nil**. Any fixnum, bignum, float (of either format) or ratio satisfies this predicate. So does a complexnum whose imaginary part is zero.

characterp *object*
t if *object* is a character object, otherwise **nil**.

stringp *object*
t if *object* is a string, otherwise **nil**.

arrayp *object*
t if *object* is an array, otherwise **nil**. Note that strings are arrays.

vectorp *object*
t if *object* is an array of rank 1.

bit-vector-p *object*
t if *object* is an array of rank 1 that allows only 0 and 1 as elements.

simple-vector-p *object*

t if *object* is an array of rank 1, with no fill pointer and not displaced, that can have any Lisp object as an element.

simple-bit-vector-p *object*

t if *object* is an array of rank 1, with no fill pointer and not displaced, that allows only 0 and 1 as elements.

simple-string-p *object*

t if *object* is a string with no fill pointer and not displaced.

functionp *object* &optional *allow-special-forms*

t if *object* is a function (essentially, something that is acceptable as the first argument to apply), otherwise nil. In addition to interpreted, compiled, and microcoded functions, functionp is true of closures, select-methods (see page 232), and symbols whose function definition is functionp.

functionp is not true of objects that can be called as functions but are not normally thought of as functions: arrays, stack groups, entities, and instances. As a special case, functionp of a symbol whose function definition is an array returns t, because in this case the array is being used as a function rather than as an object.

If *allow-special-forms* is specified and non-nil, then functionp will be true of macros and special-form functions (those with quoted arguments). Normally functionp returns nil for these since they do not behave like functions.

compiled-function-p *object*
subrp *object*

t if *object* is any compiled code object, otherwise nil. The name subrp is for Maclisp compatibility.

special-form-p *symbol*

t if *symbol* is defined as a function that takes some unevaluated args. Macros do not count as special forms.

macro-function can be used to test whether a symbol is defined as a macro, but you must be careful because it also returns a non-nil value for certain special forms. See the definition macro-function (page 344) to find out how to do this properly.

closurep *object*

t if *object* is a closure, otherwise nil.

entityp *object*

t if *object* is an entity, otherwise nil. See section 12.4, page 255 for information about entities.

locativep *object*
t if *object* is a locative, otherwise **nil**.

commonp *object*
t if *object* is of a type that Common Lisp defines operations on. See the type specifier **common** (page 18).

Other standard type predicates include **packagep** (see page 656), **random-state-p** (see page 157), **hash-table-p** (page 119), **pathnamep** (page 545), **streamp** (page 459) and **readtablep** (page 536). **defstruct** can define additional type predicates automatically (page 378).

2.3 Type Specifiers

Data types can be represented symbolically by Lisp objects called *type specifiers*. A type specifier describes a class of possible Lisp objects; the function **typep** tells whether a given object matches a given type specifier.

Built-in type specifiers exist for the actual Lisp Machine data types. The user can define additional type specifiers to represent arbitrary classifications of data. Type specifiers can also be combined into specifiers for more complex types.

Some type specifiers are symbols: for example, **number**, **cons**, **symbol**, **integer**, **character**, **compiled-function**, **array**, **vector**. Their meanings are mostly obvious, but a table follows below. Type specifiers that are symbols are called *simple* type specifiers.

Lists can also be type specifiers. They are usually combinations or restrictions of other type specifiers. The car of the list is the key to understanding what it means. An example of a combination is **(or array symbol)**, which matches any array or any symbol. An example of a restriction type is **(integer 0 6)**, which matches only integers between 0 and 6 (inclusive).

2.3.1 Standard Type Specifiers

Basic Data Types

cons	non-**nil** lists.
symbol	symbols.
array	all arrays, including strings.
number	numbers of all kinds.
instance	all instances of any flavor.
structure	named structures of any structure type.
locative	locatives.
closure	closures.
entity	entities.

stack-group	stack groups.
compiled-function	macrocode functions such as the compiler makes.
microcode-function	built-in functions implemented by the microcode.
select	select-method functions (defined by **defselect** or **defselect-incremental**).
character	character objects.

Other Useful Simple Types

t	all Lisp objects belongs to this type.
nil	nothing belongs to this type.
string-char	characters that can go in strings.
standard-char	characters defined by Common Lisp. These are the 95 ASCII printing characters (including **Space**), together with **Return**.
null	**nil** is the only object that belongs to type **null**.
list	lists, including **nil**. This type is the union of the types **null** and **cons**.
sequence	lists and vectors. Many Common Lisp functions accept either a list or a vector as a way of describing a sequence of elements.
keyword	keywords (symbols belonging to package **keyword**).
atom	anything but conses.

Simple Number Types

integer	fixnums and bignums.
ratio	explicit rational numbers, such as **1\2** (**1/2** in Common Lisp syntax).
rational	integers and ratios.
fixnum	small integers, whose **%data-type** is **dtp-fix** and which occupy no storage.
bignum	larger integers, which occupy storage.
bit	very small integers—only **0** and **1** belong to this type.
float	any floating point number regardless of format.
short-float	short floats
single-float	full-size floats
double-float **long-float**	defined by Common Lisp, but on the Lisp Machine synonymous with **single-float**.
real	any number whose value is real.

complex	a number explicitly stored as complex. It is possible for such a number to have zero as an imaginary part but only if it is a floating point zero.
noncomplex	a number which is not explicitly stored as complex. This is a subtype of **real**.

Restriction Types for Numbers

(complex *type-spec***)**
: complex numbers whose components match *type-spec*. Thus, **(complex rational)** is the type of complex numbers with rational components. **(complex t)** is equivalent to **complex**.

(integer *low high***)**
: integers between *low* and *high*. *low* can be:

integer	*integer* is an inclusive lower limit
(*integer*)	*integer* is an exclusive lower limit.
*	There is no lower limit.

high has the same sorts of possibilities. If *high* is omitted, it defaults to *. If both *low* and *high* are omitted, you have **(integer)**, which is equivalent to plain **integer**. Examples:

```
(integer 0 *)        matches any nonnegative integer.
(integer 0)          matches any nonnegative integer.
(integer -4 3)       matches any integer between -4 and 3, inclusive.
(integer -4 (4))     matches any integer between -4 and 3, inclusive.
```

bit is equivalent to **(integer 0 1)**.

(rational *low high***)**
(float *low high***)**
(short-float *low high***)**
(single-float *low high***)**
(double-float *low high***)**
(long-float *low high***)**
(noncomplex *low high***)**
: These specify restrictive bounds for the types **rational**, **float** and so on. The bounds work on these types just the way they do on **integer**. Exclusive and inclusive bounds make a useful difference here:

```
(float (-4) (3))      matches any float between -4 and 3, exclusive.
```

No possible inclusive bounds could provide the same effect.

(mod *high***)** nonnegative integers less than *high*. *high* should be an integer. **(mod)**, **(mod *)** and plain **mod** are allowed, but are equivalent to **(integer 0)**.

(signed-byte *size***)**
: integers that fit into a byte of *size* bits, of which one bit is the sign bit. **(signed-byte 4)** is equivalent to **(integer -8 7)**. **(signed-byte *)** and plain **signed-byte** are equivalent to **integer**.

(unsigned-byte *size***)**
: nonnegative integers that fit into a byte of *size* bits, with no sign bit. **(unsigned-byte 3)** is equivalent to **(integer 0 7)**. **(unsigned-byte *)** and plain **unsigned-**

byte are equivalent to **(integer 0)**.

Simple Types for Arrays

array	all arrays.
simple-array	arrays that are not displaced and have no fill pointers. (Displaced arrays are defined in section 8.2.1, page 166 and fill pointers on page 166).
vector	arrays of rank one.
bit-vector	**art-1b** arrays of rank one.
string	strings; **art-string** and **art-fat-string** arrays of rank one.
simple-bit-vector	bit vectors that are simple arrays.
simple-string	strings that are simple arrays.
simple-vector	simple-arrays of rank one, whose elements' types are unrestricted. This is not the same as **(and vector simple-array)**!

Restriction Types for Arrays

(array *element-type dimensions***)**

arrays whose rank and dimensions fit the restrictions described by *dimensions* and whose nature restricts possible elements to match *element-type*.

The array elements condition has nothing to do with the actual values of the elements. Rather, it is a question of whether the array's own type permits exactly such elements as would match *element-type*. If anything could be stored in the array that would not match *element-type*, then the array does not match. If anything that would match *element-type* could not be stored in the array, then the array does not match.

For example, if *element-type* is **(signed-byte 4)**, the array must be an **art-4b** array. An **art-1b** array will not do, even though its elements all do match **(signed-byte 4)**, because some objects such as the number 12 match **(signed-byte 4)** but could not be stored in an **art-1b** array. Likewise an **art-q** array whose elements all happen to match **(signed-byte 4)** will not do, since new elements such as **nil** or 231 which fail to match could potentially be stored in the array.

If *element-type* is **t**, the type to which all objects belong, then the array must be one in which any object can be stored: **art-q** or **art-q-list**.

***** as *element-type* means "no restriction". Any type of array is then allowed, whether it restricts its elements or not.

dimensions can be *****, an integer or a list. If it is *****, the rank and dimensions are not restricted. If it is an integer, it specifies the rank of the array. Then any array of that rank matches.

If *dimensions* is a list, its length specifies the rank, and each element of *dimensions* restricts one dimension. If the element is an integer, that dimension's length must equal it. If the element is *, that dimension's length is not restricted.

(simple-array *element-type dimensions*)
: the restrictions work as in (array *element-type dimensions*), but in addition the array must be a simple array.

(vector *element-type size*)
: *element-type* works as above. The array must be a vector. *size* must be an integer or *; if it is an integer, the array's length must equal *size*.

(bit-vector *size*)
(simple-vector *size*)
(simple-bit-vector *size*)
(string *size*)
(simple-string *size*)
: These require the array to match type **bit-vector**, **simple-vector**, etc. This implicitly restricts the element type, so there is no point in allowing an *element-type* to be given in the type specifier. *size* works as in **vector**.

More Obscure Types

package	packages, such as find-package might return.
readtable	structures such as can be the value of **readtable**.
pathname	pathnames (instances of the flavor **pathname**).
hash-table	hash-tables (instances of the flavor **hash-table**).
flavor-name	instances of that flavor, or of any flavor that contains it.
defstruct-name	named structures of that type, or of any structure that includes that one using **:include**.

Common Lisp Compatibility Types

random-state	random-states. See **random** (page 157). This is actually a special case of using a defstruct name as a type specifier, but it is mentioned specifically because Common Lisp defines this type.
common	All objects of types defined by Common Lisp. This is all Lisp objects except closures, entities, stack groups, locatives, instances, select-methods, and compiled and microcode functions. (A few kinds of instances, such as pathnames, are **common**, because Common Lisp does define how to manipulate pathnames, and it is considered irrelevant that the Lisp Machine happens to implement pathnames using instances.)
stream	Anything that looks like it might be a valid I/O stream. It is impossible to tell for certain whether an object is a stream, since any function with proper behavior may be used as a stream. Therefore, use of this type specifier is discouraged. It exists for the sake of Common Lisp.

Combination Type Specifiers

(member *objects*)
: any one of *objects*, as compared with eql. Thus, (member t nil x) is matched only by t, nil or x.

(satisfies *predicate*)
: objects on which the function *predicate* returns a non-nil value. Thus, (satisfies numberp) is equivalent as a type specifier to number (though the system could not tell that this is so). *predicate* must be a symbol, not a lambda-expression.

(and *type-specs...*)
: objecs that match all of the *type-specs* individually. Thus, (and integer (satisfies oddp)) is the type of odd integers.

(or *type-specs...*)
: objects that match at least one of the *type-specs* individually. Thus, (or number array) includes all numbers and all arrays.

(not *type-spec*) objects that do not match *type-spec*.

2.3.2 User-Defined Type Specifiers

deftype *type-name lambda-list body...* *Macro*

Defines *type-name* as a type specifier by providing code to expand it into another type specifier—a sort of type specifier macro.

When a list starting with *type-name* is encountered as a type specifier, the *lambda-list* is matched against the cdr of the type specifier just as the lambda-list of an ordinary defmacro-defined macro is matched against the cdr of a form. Then the *body* is executed and should return a new type specifier to be used instead of the original one.

If there are optional arguments in *lambda-list* for which no default value is specified, they get * as a default value.

If *type-name* by itself is encountered as a type specifier, it is treated as if it were (*type-name*); that is to say, the *lambda-list* is matched against no arguments and then the *body* is executed. So each argument in the *lambda-list* gets its default value, and there is an error if they are not all optional.

Example:

```
(deftype vector (element-type size)
  `(array ,element-type (,size)))
```

could have been used to define **vector**.

```
(deftype odd-natural-number-below (n)
  `(and (integer 0 (,n)) (satisfies oddp)))

(typep 5 '(odd-natural-number-below 6)) => t
(typep 7 '(odd-natural-number-below 6)) => nil
```

2.3.3 Testing Types with Type Specifiers

type-of *object*

Returns a type specifier which *object* matches. Any given *object* matches many different type specifiers, including **t**, so you should not attempt to rely on knowing which type specifier would be returned for any particular object. The one actually returned is chosen so as to be informative for a human. Programs should generally use **typep** rather than **type-of**.

See also **data-type**, page 270.

typep *object type-spec*

t if *object* matches *type-spec*. The fundamental purpose of type specifiers is to be used in **typep** or other functions and constructs that use **typep**. Examples:

```
(typep 5 'number) => t
(typep 5 '(integer 0 7)) => t
(typep 5 'bit) => nil
(typep 5 'array) => nil
(typep "foo" 'array) => t
(typep nil 'list) => t
(typep '(a b) 'list) => t
(typep 'lose 'list) => nil
(typep 'x '(or symbol number)) => t
(typep 5 '(or symbol number)) => t
```

If the value of *type-spec* is known at compile time, the compiler optimizes **typep** so that it does not decode the argument at run time.

In Maclisp, **typep** is used with one argument. It returns a symbol describing the type of the object it is given. This is somewhat like what **type-of** does, except in Maclisp the intention was to compare the result with **eq** to test the type of an object. The Lisp Machine supports this usage of **typep** for compatibility, but the returned symbol is a keyword (such as **:list**, for conses) which makes it actually incompatible. This usage is considered obsolete and should be removed from programs.

typecase *key-form clauses...* *Macro*

Computes the value of *key-form* and then executes one (or none) of the *clauses* according to the type of the value (call it *key*).

Each clause starts with a type specifier, not evaluated, which could be the second argument to **typep**. In fact, that is how it is used. The rest of the clause is composed of forms. The type specifiers of the clauses are matched sequentially against *key*. If there is a match, the rest of that clause is executed and the values of the last form in it are returned from the **typecase** form. If no clause matches, the **typecase** form returns **nil**.

typecase, like **typep** is optimized carefully by the compiler.

Note that t, the type specifier that matches all objects, is useful in the last clause of a **typecase**. **otherwise** is also permitted instead of **t** by special dispensation, with the same meaning.

Example:

```
(typecase foo
  (symbol (get-pname foo))
  (string foo)
  (list (apply 'string-append (mapcar 'hack foo)))
  ((integer 0) (hack-positive-integer foo))
  (t (princ-to-string foo)))
```

etypecase *key-form clauses...* *Macro*

Like **typecase** except that an uncorrectable error is signaled if every clause fails. **t** or **otherwise** clauses are not allowed.

ctypecase *place clauses...* *Macro*

Like **etypecase** except that the error is correctable. The first argument is called *place* because it must be **setf**'able (see page 36). If the user proceeds from the error, a new value is read and stored into *place*; then the clauses are tested again using the new value. Errors repeat until a value is specified that makes some clause succeed.

2.3.4 Coercion with Type Specifiers

coerce *object type-spec*

Converts *object* to an "equivalent" object that matches *type-spec*. Common Lisp specifies exactly which types can be converted to which other types. In general, a conversion that would lose information, such as turning a float into an integer, is not allowed as a coercion. Here is a complete list of types you can coerce to.

complex
(complex *type***)** Real numbers can be coerced to complex. If a rational is coerced to type **complex**, the result equals the rational, and is not complex at all. This is because complex numbers with rational components are canonicalized to real if possible. However, if a rational is coerced to **(complex float)** or **(complex single-float)** then an actual complex number does result.

It is permissible of course to coerce a complex number to a complex type. The real and imaginary parts are coerced individually to *type* if *type* is specified.

short-float
single-float Rational numbers can be coerced to floating point numbers and any kind of floating point number can be coerced to any other floating point format.

float Rational numbers are converted to **single-float**'s; floats of all kinds are left alone.

character	Strings of length one can be coerced to characters. Symbols whose print-names have length one can also be. An integer can be coerced to a character: this results in a character whose character code is the specified integer.
list	Any vector can be coerced to type list. The resulting list has the same elements as the vector.
vector or array or any restricted array type.	Any sequence (list or vector) can be coerced to any array or vector type. The new array has rank one and the same elements as the original sequence. If you specify a type of array with restricted element type, you may actually get an array which can hold other kinds of things as well. For example, the Lisp Machine does not provide anything of type (array symbol), but if you specify that, you will get an array which at least can hold symbols (but can hold other things as well). If an element of the original sequence does not fit in the new array, an error is signaled.
t	Any object can be coerced to type t, without change to the object.

If the value of *type-spec* is known at compile time, the compiler optimizes coerce so that it does not decode the argument at run time.

2.3.5 Comparing Type Specifiers

Since a type describes a set of possible objects, it is possible to ask whether one type is contained in another type. Another way to say this is, is one type a *subtype* of another?

subtypep *type1 type2*

t if *type1* is a subtype of *type2*.

The system cannot always tell whether *type1* is a subtype of *type2*. When satisfies type specifiers are in use, this question is mathematically undecidable. Because of this, it has not been considered worthwhile to make the system able to answer obscure subtype questions even when that is theoretically possible. If the answer is not known, subtypep returns nil.

Thus, nil could mean that *type1* is certainly not a subtype of *type2*, or it could mean that there is no way to tell whether it is a subtype. subtypep returns a second value to distinguish these two situations: the second value is t if subtypep's first value is definitive, nil if the system does not know the answer.

Examples:

```
(subtypep 'cons 'list) => t t
(subtypep 'null 'list) => t t
(subtypep 'symbol 'list) => nil t
```

```
(subtypep 'list 'number) => nil t
```
because not all lists are numbers (in fact, no lists are numbers).

```
(subtypep 'number 'rational) => nil t
```
because not all numbers are rational.

```
(subtypep '(satisfies foo) '(satisfies bar)) => nil nil
```
because the system does not attempt to figure out your code.

3. Evaluation

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

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

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

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

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

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

In all remaining cases, *form* is a list. The evaluator examines the car of the list to figure out what to do next. There are three possibilities: this form may be a *special form*, a *macro form*, or a plain old *function form*. If the car is an explicit function such as a list starting with **lambda**, the form is a function form. If it is a symbol, things depend on the symbol's function definition, which may be a special form definition (see page 233), a macro definition, or an ordinary function.

If *form* is a special form, then it is handled accordingly; each special form works differently. All of them are documented in this manual. The internal workings of special forms are explained in more detail on page 233, but this hardly ever affects you.

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

If *form* is a function form, it calls for the *application* of a function to *arguments*. The car of *form* is a function or the name of a function. The cdr of *form* is a list of subforms. The subforms are evaluated, sequentially, and each produces one argument for the function. The function is then applied to those arguments. Whatever results the function returns are the values of the original *form*.

There is a lot more to be said about evaluation. The way variables work and the ways in which they are manipulated, including the binding of arguments, is explained in section 3.1, page 25. A basic explanation of functions is in section 3.3, page 38. The way functions can return more than one value is explained in section 3.7, page 55. The description of all of the kinds of functions, and the means by which they are manipulated, is in chapter 11. Macros are explained in chapter 18. The **evalhook** facility, which lets you do something arbitrary whenever the evaluator is invoked, is explained in section 30.12, page 748. Special forms are described all over the manual; each special form is in the section on the facility it is part of.

3.1 Variables

In Zetalisp, variables are implemented using symbols. Symbols are used for many things in the language, such as naming functions, naming special forms, and being keywords; they are also useful to programs written in Lisp, as parts of data structures. But when a symbol is evaluated, its value as a variable is taken.

3.1.1 Variables and Bindings

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
(defvar a)

(defun foo () (print a))

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

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

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

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

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

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

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

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

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

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

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

Here is an example where lexical bindings are desirable:

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

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

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

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

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

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

(setq mc (mycons 4 t))

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

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

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

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

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

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

3.1.2 Setting Variables

Here are the constructs used for setting variables.

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

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

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

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

psetq {*variable value*}... *Macro*

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

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

variable-location *symbol* *Special form*

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

For a special variable, this is equivalent to

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

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

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

variable-boundp *symbol* *Special form*

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

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

symbol is not evaluated.

variable-makunbound *symbol* *Special form*

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

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

symbol is not evaluated.

3.1.3 Variable Binding Constructs

Here are the constructs used for binding variables.

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

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

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

When this form is evaluated, first the *vforms* (the values) are evaluated. Then the *vars* are bound to the values returned by the corresponding *vforms*. Thus the bindings happen in parallel; all the *vforms* are evaluated before any of the *vars* are bound. Finally, the *bforms* (the body) are evaluated sequentially, the old values of the variables are restored, and the result of the last *bform* is returned.

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

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

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

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

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

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

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

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

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

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

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

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

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

let-globally is similar in form to let (see page 31). The difference is that let-globally does not *bind* the variables; instead, it saves the old values and *sets* the variables, and sets up an unwind-protect (see page 82) to set them back. The important consequence is that, with let-globally, when the current stack group (see chapter 13, page 256) co-calls some other stack group, the old values of the variables are *not* restored. Thus let-globally makes the new values visible in all stack groups and processes that don't bind the variables themselves, not just in the current stack group. Therefore, let-globally can be used for communication between stack groups and between processes.

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

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

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

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

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

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

After the symbols have been bound to the values, the *body* forms are evaluated, and finally the symbols' bindings are undone. The result returned is the value of the last form in the body. Assuming that the variables a, b, foo and bar are globally special, we can do:

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

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

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

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

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

First, *vars-and-val-forms-form* is evaluated. Its value should be a list that looks like the first subform of a **let**:

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

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

This is a very unusual special form because of the way the evaluator is called on the result of an evaluation. **progw** is useful mainly for implementing special forms and for functions part of whose contract is that they call the interpreter. For an example of the latter, see **sys:*break-bindings*** (page 797); **break** implements this by using **progw**.

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

3.1.4 Defining Global Variables

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

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

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

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

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

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

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

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

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

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

If defvar is used in a patch file (see section 28.8, page 672) or is a single form (not a region) evaluated with the editor's compile/evaluate from buffer commands, if there is an initial-value the variable is always set to it regardless of whether it is void.

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

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

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

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

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

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

3.1.5 The Global Binding

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

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

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

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

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

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

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

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

3.2 Generalized Variables

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

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

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

3.2.1 setf

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

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

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

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

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

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

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

Examples:

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

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

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

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

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

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

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

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

interchanges the car and cdr of x.

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

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

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

shiftf *place...* *Macro*

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

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

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

rotatef *place...* *Macro*

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

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

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

is equivalent to the psetf example above.

swapf *place1 place2* *Macro*

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

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

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

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

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

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

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

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

3.2.2 locf

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

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

locf *place* *Macro*

locf takes a form that *accesses* some cell, and produces a corresponding form to create a locative pointer to that cell.
Examples:
```
(locf (array-leader foo 3)) ==> (ap-leader foo 3)
(locf a) ==> (value-cell-location 'a)
(locf (plist 'a)) ==> (property-cell-location 'a)
(locf (aref q 2)) ==> (aloc q 2)
```

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

3.3 Functions

In the description of evaluation on page 24, we said that evaluation of a function form works by applying the function to the results of evaluating the argument subforms. What is a function, and what does it mean to apply it? In Zetalisp there are many kinds of functions, and applying them may do many different kinds of things. For full details, see chapter 11, page 223. Here we explain the most basic kinds of functions and how they work. In particular, this section explains *lambda lists* and all their important features.

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

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

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

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

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

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

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

In general, a function in Zetalisp has zero or more *positional* parameters, followed if desired by a single *rest* parameter, followed by zero or more *keyword* parameters. The positional parameters may be *required* or *optional*, but all the optional parameters must follow all the required ones. The required/optional distinction does not apply to the rest parameter; all keyword parameters are optional.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`(&optional a b c)`
a, b, and c are all optional and positional. The function may be passed zero, one, two or three arguments.

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

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

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

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

or

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

or

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

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

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

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

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

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

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

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

`(a &optional (b 3))`
: The default-form for b is 3. a is a required parameter, and so it doesn't have a default form.

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

Occasionally it is important to know whether a certain optional parameter was defaulted or not. Just by looking at the value one cannot distinguish between omitting it and passing the default value explicitly as an argument. The way to tell for sure is to put a third element into the list: the third element should be a variable (a symbol), and that variable is bound to nil if the parameter was not passed by the caller (and so was defaulted), or t if the parameter was passed. The new variable is called a "supplied-p" variable; it is bound to t if the parameter is supplied. For example:

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

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

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

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

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

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

```
(a &optional b &rest c &aux d (e 5) (f (cons a e)))
```

d, e, and f are bound, when the function is called, to **nil**, **5**, and a cons of the first argument and 5.

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

It is important to realize that the list of arguments to which a rest-parameter is bound is set up in whatever way is most efficiently implemented, rather than in the way that is most convenient for the function receiving the arguments. It is not guaranteed to be a "real" list. Sometimes the rest-args list is a stack list (see section 5.9, page 112) stored in the function-calling stack, and loses its validity when the function returns. If a rest-argument is to be returned or made part of permanent list-structure, it must first be copied (see **copylist**, page 94), as you must always assume that it is one of these special lists. The system does not detect the error of omitting to copy a rest-argument; you will simply find that you have a value which seems to change behind your back.

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

lambda-parameters-limit *Constant*

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

3.3.1 Lambda-List Keywords

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

lambda-list-keywords *Constant*

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

&optional	Separates the required arguments of a function from the optional arguments. See section 3.3, page 38.
&rest	Separates the required and optional arguments of a function from the rest argument. There may be only one rest argument. See page 41 for full information about rest arguments. See section 3.3, page 38.
&key	Separates the positional arguments and rest argument of a function from the keyword arguments. See section 3.3, page 38.
&allow-other-keys	In a function that accepts keyword arguments, says that keywords that are not recognized are allowed. They and the corresponding values are ignored, as far as keyword arguments are concerned, but they do become part of the rest argument, if there is one.
&aux	Separates the arguments of a function from the auxiliary variables. Following **&aux** you can put entries of the form (*variable initial-value-form*) or just *variable* if you want it initialized to **nil** or don't care what the initial value is.
&special	Declares the following arguments and/or auxiliary variables to be special within the scope of this function.
&local	Turns off a preceding **&special** for the variables that follow.
&quote	Declares that the following arguments are not to be evaluated. This is how you create a special function. See the caveats about special forms on page 233.
&eval	Turns off a preceding **&quote** for the arguments which follow.
&list-of	This is for macros defined by **defmacro** only. Refer to page 324.
&body	This is for macros defined by **defmacro** only. It is similar to **&rest**, but declares to **grindef** and the code-formatting module of the editor that the body forms of a special form follow and should be indented accordingly. Refer to page 324.
&whole	This is for macros defined by **defmacro** only. It means that the following argument is bound to the entire macro call form being expanded. Refer to page 324.
&environment	This is for macros defined by **defmacro** only. It means that the following argument is bound to an environment structure which records the local **macrolet** macro definitions in effect for subforms of the macro call form. Refer to page 324.

3.3.2 Local Functions

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

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

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

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

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

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

Example:

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

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

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

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

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

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

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

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

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

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

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

3.4 Some Functions and Special Forms

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

eval *form* &optional *nohook*

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

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

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

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

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

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

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

si:evall *form* &optional *nohook*

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

funcall *f* &rest *args*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

call-arguments-limit *Constant*

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

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

call *function* &rest *argument-specifications*

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

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

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

Example:

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

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

quote *object* *Special form*

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

Examples:

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

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

For example,

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

is converted by **read** into

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

function *f* *Special form*

function has two distinct, though related, meanings.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

or

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

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

false

Takes no arguments and returns **nil**.

true

Takes no arguments and returns **t**.

ignore &rest *ignore*

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

comment *Special form*

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

Example:

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

;; prevents this definition of **foo** from being used.

3.5 Declarations

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

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

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

A **local-declare** form looks like

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

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

declare *declaration...* *Special form*

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

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

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

is equivalent to

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

Note that

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

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

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

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

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

is equivalent to

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

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

Elsewhere, declare's are ignored.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(:self-flavor *flavorname***)**
: Instance variables of the flavor *flavorname*, in **self**, will be accessible in the function.

locally &body *body* *Macro*

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

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

proclaim &rest *declarations*

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

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

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

special *variables...* *Special form*

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

unspecial *variables...* *Special form*

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

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

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

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

3.6 Tail Recursion

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

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

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

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

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

tail-recursion-flag *Variable*

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

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

3.7 Multiple Values

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

The primitive for producing multiple values is values, which takes any number of arguments and returns that many values. If the last form in the body of a function is a values with three arguments, then a call to that function returns three values. Many system functions produce multiple values, but they all do it via values.

values &rest *args*

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

values-list *list*

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

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

Here are the special forms for receiving multiple values.

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

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

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

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

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

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

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

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

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

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

Evaluates the argforms, saving all of their values, and then calls *function* with all those values as arguments. This differs from
```
(funcall function argforms...)
```
because that would get only one argument for *function* from each *argform*, whereas multiple-value-call gets as many args from each *argform* as the *argform* cares to return. This works by consing a list of all the values returned, and applying *function* to it. Example:
```
(multiple-value-call 'append
     (values '(a b) '(c d))
     '(e f))
   =>  (a b c d e f)
```

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

Evaluates *form*, saves its values, evaluates the *forms*, discards their values, then returns whatever values *form* produced. This does not cons. Example:
```
(multiple-value-prog1 (values 1 2)
     (print 'foo))
   =>  1 2
```

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

multiple-value-list evaluates *form*, and returns a list of the values it returned. This is useful for when you don't know how many values to expect.
Example:
```
(setq a (multiple-value-list (intern "goo")))
a => (goo nil #<Package USER 10112147>)
```
This is similar to the example of multiple-value above; a is set to a list of three elements, the three values returned by intern.

nth-value *n form* *Special form*

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

When one form finished by tail recursively evaluating a subform (see section 3.6, page 55), all of the subform's multiple values are passed back by the outer form. For example, the value of a cond is the value of the last form in the selected clause. If the last form in that clause produces multiple values, so does the cond. This *passing-back* of multiple values of course has no effect unless eventually one of the special forms for receiving multiple values is reached.

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

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

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

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

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

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

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

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

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

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

prog behaves like **block** if it is exited with **return** (or **return-from**). If control falls through the end of a **prog**, it returns the single value **nil**. **do** also behaves like **block** with respect to **return**, but if it is exited through the exit test, all the values of the last *exit-form* are returned.

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

catch passes back multiple values from the last form in its body, if it exits normally. If a throw is done, multiple values are passed back from the value form in the **throw**.

multiple-values-limit *Constant*

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

3.8 Evaluation and Function Calling Errors

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

sys:invalid-function (error) *Condition*

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

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

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

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

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

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

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

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

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

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

This is signaled when a function that takes keyword arguments is given a keyword that it does not accept, if &allow-other-keys was not used in the function's definition and :allow-other-keys was not specified by the caller (see page 40). The :keyword operation on the condition instance returns the extraneous keyword, and the :value operation returns the value supplied with it.

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

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

This condition name categorizes all the errors signaled because of references to void memory locations. It includes "unbound" variables, "undefined" functions, and other things.

:address	A locative pointer to the referenced cell.
:current-address	A locative pointer to the cell which currently contains the contents that were found in the referenced cell when the error happened. This can be different from the original address in the case of dynamic variable bindings, which move between special PDLs and symbol value cells.
:cell-type	A keyword saying what type of cell was referred to: :function, :value, :closure, or nil for a cell that is not one of those.
:containing-structure	The object (list, array, symbol) inside which the referenced memory cell is found.
:data-type :pointer	The data type and pointer fields of the contents of the memory cell, at the time of the error. Both are fixnums.

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

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

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

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

This condition name categorizes all errors of variables whose values are void.

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

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

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

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

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

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

The condition instance supports these extra operations:

:arg-name	The name of the erroneous argument. This may be nil if there is no name, or if the system no longer remembers which argument it was.
:old-value	The value that was supplied for the argument.
:function	The function which received and rejected the argument.
:description	A type specifier which says what sort of object was expected for this argument.

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

4. Flow of Control

Lisp provides a variety of structures for flow of control.

Function application is the basic method for construction of programs. Operations are written as the application of a function to its arguments. Usually, Lisp programs are written as a large collection of small functions, each of which implements a simple operation. These functions operate by calling one another, and so larger operations are defined in terms of smaller ones.

A function may always call itself in Lisp. The calling of a function by itself is known as *recursion*; it is analogous to mathematical induction.

The performing of an action repeatedly (usually with some changes between repetitions) is called *iteration*, and is provided as a basic control structure in most languages. The *do* statement of PL/I, the *for* statement of ALGOL/60, and so on are examples of iteration primitives. Lisp provides two general iteration facilities: **do** and **loop**, as well as a variety of special-purpose iteration facilities. (**loop** is sufficiently complex that it is explained in its own chapter later in the manual; see page 350.)

A *conditional* construct is one which allows a program to make a decision, and do one thing or another based on some logical condition. Lisp provides the simple one-way conditionals **and** and **or**, the simple two-way conditional **if**, and more general multi-way conditionals such as **cond** and **selectq**. The choice of which form to use in any particular situation is a matter of personal taste and style.

There are some *non-local exit* control structures, analogous to the *leave*, *exit*, and *escape* constructs in many modern languages. Zetalisp provides for both static (lexical) non-local exits with **block** and **return-from** and dynamic non-local exits with **catch** and **throw**. Another kind of non-local exit is the goto, provided by the **tagbody** and **go** constructs.

Zetalisp also provides a coroutine capability, explained in the section on *stack-groups* (chapter 13, page 256), and a multiple-process facility (see chapter 29, page 682). There is also a facility for generic function calling using message passing; see chapter 21, page 401.

4.1 Compound Statements

progn *body...* *Special form*

The *body* forms are evaluated in order from left to right and the value of the last one is returned. **progn** is the primitive control structure construct for "compound statements". Example:

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

Lambda-expressions, **cond** forms, **do** forms, and many other control structure forms use **progn** implicitly, that is, they allow multiple forms in their bodies.

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

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

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

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

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

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

4.2 Conditionals

if *Special form*

if is the simplest conditional form. The "if-then" form looks like:

(if *predicate-form then-form*)

predicate-form is evaluated, and if the result is non-**nil**, the *then-form* is evaluated and its result is returned. Otherwise, **nil** is returned.

In the "if-then-else" form, it looks like

(if *predicate-form then-form else-form*)

predicate-form is evaluated, and if the result is non-**nil**, the *then-form* is evaluated and its result is returned. Otherwise, the *else-form* is evaluated and its result is returned.

If there are more than three subforms, **if** assumes you want more than one *else-form*; if the predicate returns **nil**, they are evaluated sequentially and the result of the last one is returned.

when *condition body...* *Macro*

If *condition* evaluates to something non-**nil**, the *body* is executed and its value(s) returned. Otherwise, the value of the **when** is **nil** and the *body* is not executed.

unless *condition body...* *Macro*

If *condition* evaluates to **nil**, the *body* is executed and its value(s) returned. Otherwise, the value of the **unless** is **nil** and the *body* is not executed.

cond *Special form*

The **cond** special form consists of the symbol **cond** followed by several *clauses*. Each clause consists of a predicate form, called the *condition*, followed by zero or more *body* forms.

```
(cond (condition body body...)
      (condition)
      (condition body ...)
      ... )
```

The idea is that each clause represents a case which is selected if its condition is satisfied and the conditions of all preceding clauses were not satisfied. When a clause is selected, its body forms are evaluated.

cond processes its clauses in order from left to right. First, the condition of the current clause is evaluated. If the result is **nil**, **cond** advances to the next clause. Otherwise, the cdr of the clause is treated as a list of body forms which are evaluated in order from left to right. After evaluating the body forms, **cond** returns without inspecting any remaining clauses. The value of the **cond** form is the value of the last body form evaluated, or the value of the condition if there were no body forms in the clause. If **cond** runs out of clauses, that is, if every condition evaluates to **nil**, and thus no case is selected, the value of the **cond** is **nil**.
Example:

```
(cond ((zerop x)        ;First clause:
       (+ y 3))         ; (zerop x) is the condition.
                        ; (+ y 3) is the body.
      ((null y)         ;A clause with 2 body forms:
       (setq y 4)       ; this
       (cons x z))      ; and this.
      (z)               ;A clause with no body forms: the condition is
                        ; just z. If z is non-nil, it is returned.
      (t                ;A condition of t
       105)             ; is always satisfied.
   )                    ;This is the end of the cond.
```

cond-every *Macro*

cond-every has the same syntax as **cond**, but executes every clause whose condition is satisfied, not just the first. If a condition is the symbol **otherwise**, it is satisfied if and only if no preceding condition is satisfied. The value returned is the value of the last body form in the last clause whose condition is satisfied. Multiple values are not returned.

and *form...* *Special form*

and evaluates the *forms* one at a time, from left to right. If any *form* evaluates to **nil**, **and** immediately returns **nil** without evaluating the remaining *forms*. If all the *forms* evaluate to non-nil values, **and** returns the value of the last *form*.

and can be used in two different ways. You can use it as a logical **and** function, because it returns a true value only if all of its arguments are true:

```
(if (and socrates-is-a-person
         all-people-are-mortal)
    (setq socrates-is-mortal t))
```

Because the order of evaluation is well-defined, you can do

```
(if (and (boundp 'x)
         (eq x 'foo))
    (setq y 'bar))
```

knowing that the x in the **eq** form will not be evaluated if x is found to be void.

You can also use **and** as a simple conditional form:

```
(and (setq temp (assq x y))
     (rplacd temp z))
(and bright-day
     glorious-day
     (princ "It is a bright and glorious day."))
```

but **when** is usually preferable.

Note: **(and)** => **t**, which is the identity for the **and** operation.

or *form...* *Special form*

or evaluates the *forms* one by one from left to right. If a *form* evaluates to **nil**, **or** proceeds to evaluate the next *form*. If there are no more *forms*, **or** returns **nil**. But if a *form* evaluates to a non-**nil** value, **or** immediately returns that value without evaluating any remaining *forms*.

As with **and**, **or** can be used either as a logical **or** function, or as a conditional:

```
(or it-is-fish it-is-fowl)

(or it-is-fish it-is-fowl
   (print "It is neither fish nor fowl."))
```

but it is often possible and cleaner to use **unless** in the latter case.

Note: **(or)** => **nil**, the identity for this operation.

selectq *Macro*
case *Macro*
caseq *Macro*

selectq is a conditional which chooses one of its clauses to execute by comparing the value of a form against various constants using **eql**. Its form is as follows:

```
(selectq key-form
  (test body...)
  (test body...)
  (test body...)
  ...)
```

The first thing **selectq** does is to evaluate *key-form*; call the resulting value *key*. Then **selectq** considers each of the clauses in turn. If *key* matches the clause's *test*, the body of the clause is evaluated, and **selectq** returns the value of the last body form. If there are no matches, **selectq** returns **nil**.

A *test* may be any of:

1) A symbol	If the *key* is **eql** to the symbol, it matches.
2) A number	If the *key* is **eql** to the number, it matches. *key* must have the same type and the same value as the number.
3) A list	If the *key* is **eql** to one of the elements of the list, then it matches. The elements of the list should be symbols or numbers.
4) **t** or **otherwise**	The symbols **t** and **otherwise** are special tests which match anything. Either symbol may be used, it makes no difference; **t** is mainly for compatibility with Maclisp's **caseq** construct. To be useful, this should be the last clause in the **selectq**.

Example:
```
       (selectq x
         (foo (do-this))
         (bar (do-that))
         ((baz quux mum) (do-the-other-thing))
         (otherwise (ferror nil "Never heard of ~S" x)))
```
is equivalent to
```
       (cond ((eq x 'foo) (do-this))
             ((eq x 'bar) (do-that))
             ((memq x '(baz quux mum)) (do-the-other-thing))
             (t (ferror nil "Never heard of ~S" x)))
```

Note that the *tests* are *not* evaluated; if you want them to be evaluated use **select** rather than **selectq**.

case is the Common Lisp name for this construct. **caseq** is the Maclisp name; it identical to **selectq**, which is not totally compatible with Maclisp, because **selectq** accepts **otherwise** as well as **t** where **caseq** would not accept **otherwise**, and because Maclisp does some error-checking that **selectq** does not. Maclisp programs that use **caseq** work correctly so long as they don't use the symbol **otherwise** as a key.

ecase *key-form clauses...* *Macro*
Like **case** except that an uncorrectable error is signaled if every clause fails. **t** or **otherwise** clauses are not allowed.

ccase *place clauses...* *Macro*
Like **ecase** except that the error is correctable. The first argument is called *place* because it must be **setf**'able. If the user proceeds from the error, a new value is read and stored into *place*; then the clauses are tested again using the new value. Errors repeat until a value is specified which makes some clause succeed.

Also see **defselect** (page 236), a special form for defining a function whose body is like a **selectq**.

select *Macro*

select is like selectq, except that the elements of the *tests* are evaluated before they are used.

This creates a syntactic ambiguity: if (bar baz) is seen the first element of a clause, is it a list of two forms, or is it one form? select interprets it as a list of two forms. If you want to have a clause whose test is a single form, and that form is a list, you have to write it as a list of one form.
Example:

```
(select (frob x)
   (foo 1)
   ((bar baz) 2)
   (((current-frob)) 4)
   (otherwise 3))
```

is equivalent to

```
(let ((var (frob x)))
  (cond ((eq var foo) 1)
        ((or (eq var bar) (eq var baz)) 2)
        ((eq var (current-frob)) 4)
        (t 3)))
```

selector *Macro*

selector is like select, except that you get to specify the function used for the comparison instead of eq. For example,

```
(selector (frob x) equal
   (('(one . two)) (frob-one x))
   (('(three . four)) (frob-three x))
   (otherwise (frob-any x)))
```

is equivalent to

```
(let ((var (frob x)))
  (cond ((equal var '(one . two)) (frob-one x))
        ((equal var '(three . four)) (frob-three x))
        (t (frob-any x))))
```

select-match *Macro*

select-match is like select but each clause can specify a pattern to match the key against. The general form of use looks like

(select-match *key-form*
  (*pattern condition body*...)
  (*pattern condition body*...)
  ...
  (otherwise *body*...))

The value of *key-form* is matched against the *patterns* one at a time until a match succeeds and the accompanying *condition* evaluates to something non-nil. At this point the *body* of that clause is executed and its value(s) returned. If all the patterns/conditions fail, the *body* of the otherwise clause (if any) is executed. A pattern can test the shape of the key object, and set variables which the *condition* form can refer to. All the variables set by the patterns are bound locally to the select-match form.

The patterns are matched using list-match-p (page 92).

Example:
```
        (select-match '(a b c)
          ('(,x b ,x) t (vector x))
          ('((,x ,y) b . ,ignore) t (list x y))
          ('(,x b ,y) (symbolp x) (cons x y))
          (otherwise 'lose-big))
```
returns (a . c), having checked (symbolp 'a). The first clause matches only if the there are three elements, the first and third elements are equal and the second element is b. The second matches only if the first element is a list of length two and the second element is b. The third clause accepts any list of length three whose second element is b. The fourth clause accepts anything that did not match the previous clauses.

select-match generates highly optimized code using special instructions.

dispatch *Macro*

(dispatch *byte-specifier number clauses...*) is the same as select (not selectq), but the key is obtained by evaluating (ldb *byte-specifier number*). *byte-specifier* and *number* are both evaluated. Byte specifiers and ldb are explained on page 155.
Example:
```
        (princ (dispatch (byte 2 13) cat-type
                   (0 "Siamese.")
                   (1 "Persian.")
                   (2 "Alley.")
                   (3 (ferror nil
                              "~S is not a known cat type."
                              cat-type))))
```

It is not necessary to include all possible values of the byte which is dispatched on.

selectq-every *Macro*

selectq-every has the same syntax as selectq, but, like cond-every, executes every selected clause instead of just the first one. If an otherwise clause is present, it is selected if and only if no preceding clause is selected. The value returned is the value of the last form in the last selected clause. Multiple values are not returned. Example:
```
        (selectq-every animal
          ((cat dog) (setq legs 4))
          ((bird man) (setq legs 2))
          ((cat bird) (put-in-oven animal))
          ((cat dog man) (beware-of animal)))
```

4.2.1 Comparison Predicates

eq *x y*

(eq *x y*) => t if and only if *x* and *y* are the same object. It should be noted that things that print the same are not necessarily eq to each other. In particular, numbers with the same value need not be eq, and two similar lists are usually not eq.
Examples:

```
(eq 'a 'b) => nil
(eq 'a 'a) => t
(eq (cons 'a 'b) (cons 'a 'b)) => nil
(setq x (cons 'a 'b)) (eq x x) => t
```

Note that in Zetalisp equal fixnums are eq; this is not true in Maclisp. Equality does not imply eq-ness for other types of numbers. To compare numbers, use =; see page 139.

neq *x y*

(neq *x y*) = (not (eq *x y*)). This is provided simply as an abbreviation for typing convenience.

eql *x y*

eql is the same as eq except that if *x* and *y* are numbers of the same type they are eql if they are =.

equal *x y*

The equal predicate returns t if its arguments are similar (isomorphic) objects. Two numbers are equal if they have the same value and type (for example, a float is never equal to a fixnum, even if = is true of them). For conses, equal is defined recursively as the two cars being equal and the two cdrs being equal. Two strings are equal if they have the same length, and the characters composing them are the same; see string=, page 214. Alphabetic case is significant. All other objects are equal if and only if they are eq. Thus equal could have been defined by:

```
(defun equal (x y)
  (cond ((eq x y) t)
        ((and (numberp x) (numberp y))
         (= x y))
        ((and (stringp x) (stringp y))
         (string-equal x y))
        ((and (consp x) (consp y))
         (and (equal (car x) (car y))
              (equal (cdr x) (cdr y))))))
```

As a consequence of the above definition, it can be seen that equal may compute forever when applied to looped list structure. In addition, eq always implies equal; that is, if (eq a b) then (equal a b). An intuitive definition of equal (which is not quite correct) is that two objects are equal if they look the same when printed out. For example:

```
(setq a '(1 2 3))
(setq b '(1 2 3))
(eq a b) => nil
(equal a b) => t
(equal "Foo" "foo") => nil
```

equalp *x y*

equalp is a broader kind of equality than equal. Two objects that are equal are always equalp. In addition, numbers of different types are equalp if they are =. Two character objects are equalp if they are char-equal (that is, they are compared ignoring font, case and meta bits).

Two arrays of any sort are equalp if they have the same dimensions and corresponding elements are equalp. In particular, this means that two strings are equalp if they match ignoring case and font information.

```
(equalp "Foo" "foo") => t
(equalp '1 '1.0) => t
(equalp '(1 "Foo") '(1.0 "foo")) => t
```

Because equalp is a Common Lisp function, it regards a string as having character objects as its elements:

```
(equalp "Foo" #(#≠/F #≠/o #≠/o)) => t
(equalp "Foo" #(#/F #/o #/o)) => nil
```

not *x*
null *x*

not returns t if *x* is nil, else nil. null is the same as not; both functions are included for the sake of clarity. Use null to check whether something is nil; use not to invert the sense of a logical value. Some people prefer to distinguish between nil as falsehood and nil as the empty list by writing:

```
(cond ((not (null lst)) ... )
      ( ... ))
rather than
(cond (lst ... )
      ( ... ))
```

There is no loss of efficiency, since these compile into exactly the same instructions.

4.3 Iteration

do *Special form*

The do special form provides a simple generalized iteration facility, with an arbitrary number of "index variables" whose values are saved when the do is entered and restored when it is left, i.e. they are bound by the do. The index variables are used in the iteration performed by do. At the beginning, they are initialized to specified values, and then at the end of each trip around the loop the values of the index variables are changed according to specified rules. do allows the programmer to specify a predicate which determines when the iteration will terminate. The value to be returned as the result of the form may, optionally, be specified.

do comes in two varieties, new-style and old-style. The old-style do is obsolete and exists for Maclisp compatibility only. The more general, "new-style" do looks like:

```
(do ((var init repeat) ...)
    (end-test exit-form ...)
    body...)
```

The first item in the form is a list of zero or more index variable specifiers. Each index variable specifier is a list of the name of a variable *var*, an initial value form *init*, which defaults to nil if it is omitted, and a repeat value form *repeat*. If *repeat* is omitted, the *var* is not changed between repetitions. If *init* is omitted, the *var* is initialized to nil.

An index variable specifier can also be just the name of a variable, rather than a list. In this case, the variable has an initial value of nil, and is not changed between repetitions.

All assignment to the index variables is done in parallel. At the beginning of the first iteration, all the *init* forms are evaluated, then the *vars* are bound to the values of the *init* forms, their old values being saved in the usual way. Note that the *init* forms are evaluated *before* the *vars* are bound, i.e. lexically *outside* of the do. At the beginning of each succeeding iteration those *vars* that have *repeat* forms get set to the values of their respective *repeat* forms. Note that all the *repeat* forms are evaluated before any of the *vars* is set.

The second element of the do-form is a list of an end-testing predicate form *end-test*, and zero or more forms, called the *exit-forms*. This resembles a cond clause. At the beginning of each iteration, after processing of the variable specifiers, the *end-test* is evaluated. If the result is nil, execution proceeds with the body of the do. If the result is not nil, the *exit-forms* are evaluated from left to right and then do returns. The value of the do is the value of the last *exit-form*, or nil if there were no *exit-forms* (*not* the value of the *end-test*, as you might expect by analogy with cond).

Note that the *end-test* gets evaluated before the first time the body is evaluated. do first initializes the variables from the *init* forms, then it checks the *end-test*, then it processes the body, then it deals with the *repeat* forms, then it tests the *end-test* again, and so on. If the end-test returns a non-nil value the first time, then the body is not executed.

If the second element of the form is nil, there is no *end-test* nor *exit-forms*, and the *body* of the do is executed only once. In this type of do it is an error to have *repeats*. This type of do is no more powerful than let; it is obsolete and provided only for Maclisp compatibility.

If the second element of the form is (nil), the *end-test* is never true and there are no *exit-forms*. The *body* of the do is executed over and over. The resulting infinite loop can be terminated by use of return or throw.

do implicitly creates a block with name nil, so return can be used lexically within a do to exit it immediately. This unbinds the do variables and the do form returns whatever values were specified in the return form. See section 4.4, page 75 for more information

on these matters. The *body* of the do is actually treated as a tagbody, so that it may contain go tags (see section 4.5, page 78), but this usage is discouraged as it is often unclear.
Examples of the new form of do:

```
(do ((i 0 (1+ i))           ; This is just the same as the above example,
     (n (array-length foo-array)))
    ((= i n))               ; but written as a new-style do.
  (aset 0 foo-array i))     ; Note how the setq is avoided.

(do ((z list (cdr z))  ; z starts as list and is cdr'd each time.
     (y other-list)    ; y starts as other-list, and is unchanged by the do.
     (x)               ; x starts as nil and is not changed by the do.
     w)                ; w starts as nil and is not changed by the do.
    (nil)              ; The end-test is nil, so this is an infinite loop.
  body)        ; Presumably the body uses return somewhere.
```

The construction

```
(do ((x e (cdr x))
     (oldx x x))
    ((null x))
  body)
```

exploits parallel assignment to index variables. On the first iteration, the value of oldx is whatever value x had before the do was entered. On succeeding iterations, oldx contains the value that x had on the previous iteration.

The *body* of a do may contains no forms at all. Very often an iterative algorithm can be most clearly expressed entirely in the *repeats* and *exit-forms* of a new-style do, and the *body* is empty. For example,

```
(do ((x x (cdr x))
     (y y (cdr y))
     (z nil (cons (f x y) z)))  ;exploits parallel assignment.
    ((or (null x) (null y))
     (nreverse z))              ;typical use of nreverse.
    )                           ;no do-body required.
```

is like (maplist 'f x y) (see page 84).

The old-style do exists only for Maclisp compatibility. It looks like:

```
(do var init repeat end-test body...)
```

The first time through the loop *var* gets the value of the *init* form; the remaining times through the loop it gets the value of the *repeat* form, which is re-evaluated each time. Note that the *init* form is evaluated before *var* is bound, i.e. lexically *outside* of the do. Each time around the loop, after *var* is set, *end-test* is evaluated. If it is non-nil, the do finishes and returns nil. If the *end-test* evaluated to nil, the *body* of the loop is executed. As with the new-style do, return and go may be used in the body, and they have the same meaning.

Also see loop (page 350), a general iteration facility based on a keyword syntax rather than a list-structure syntax.

do* *Special form*

In a word, do* is to do as let* is to let.

do* works like do but binds and steps the variables sequentially instead of in parallel. This means that the *init* form for one variable can use the values of previous variables. The *repeat* forms refer to the new values of previous variables instead of their old values. Here is an example:

```
(do* ((x xlist (cdr x))
      (y (car x) (car x)))
   (print (list x y)))
```

On each iteration, *y*'s value is the car of *x*. The same construction with do might get an error on entry since *x* might not be bound yet.

do-named *Special form*

do-named is like do but defines a block with a name explicitly specified by the programmer in addition to the block named nil which every do defines. This makes it possible to use return-from to return from this do-named even from within an inner do. An ordinary return there would return from the inner do instead. do-named is obsolete now that block, which is more general and more coherent, exists. See section 4.4, page 75 for more information on block and return-from.

The syntax of do-named is like do except that the symbol do-named is immediately followed by the block name, which should be a symbol.
Example:

```
(do-named george ((a 1 (1+ a))
                  (d 'foo))
                 ((> a 4) 7)
  (do ((c b (cdr c)))
      ((null c))
    ...
    (return-from george (cons b d))
    ...))
```

is equivalent to

```
(block george
  (do ((a 1 (1+ a))
       (d 'foo))
      ((> a 4) 7)
    (do ((c b (cdr c)))
        ((null c))
      ...
      (return-from george (cons b d))
      ...)))
```

t as the name of a do-named behaves somewhat peculiarly, and therefore should be avoided.

do*-named *Special form*

This special form offers a combination of the features of do* and those of do-named. It is obsolete, as is do-named, since it is cleaner to use block.

dotimes (*index count* [*value-expression*]) *body...* *Macro*

dotimes is a convenient abbreviation for the most common integer iteration. dotimes performs *body* the number of times given by the value of *count*, with *index* bound to 0, 1, etc. on successive iterations. When the *count* is exhausted, the value of *value-expression* is returned; or nil, if *value-expression* is missing.

Example:

```
(dotimes (i (truncate m n))
  (frob i))
```

is equivalent to:

```
(do ((i 0 (1+ i))
     (count (truncate m n)))
    ((≥ i count))
  (frob i))
```

except that the name count is not used. Note that i takes on values starting at zero rather than one, and that it stops before taking the value (truncate m n) rather than after. You can use return and go and tagbody-tags inside the body, as with do. dotimes forms return the value of *value-expression*, or nil, unless returned from explicitly with return. For example:

```
(dotimes (i 5)
  (if (eq (aref a i) 'foo)
      (return i)))
```

This form searches the array that is the value of a, looking for the symbol foo. It returns the fixnum index of the first element of a that is foo, or else nil if none of the elements are foo.

dolist (*item list* [*value-expression*]) *body...* *Macro*

dolist is a convenient abbreviation for the most common list iteration. dolist performs *body* once for each element in the list which is the value of *list*, with *item* bound to the successive elements. If the list is exhausted, the value of *value-expression* is returned; or nil, if *value-expression* is missing.

Example:

```
(dolist (item (frobs foo))
  (mung item))
```

is equivalent to:

```
(do ((lst (frobs foo) (cdr lst))
     (item))
    ((null lst))
  (setq item (car lst))
  (mung item))
```

except that the name lst is not used. You can use return and go and tagbody-tags inside the body, as with do.

do-forever *body...* *Macro*
Executes the forms in the body over and over, or until a non-local exit (such as **return**).

4.4 Static Non-Local Exits

The static non-local exit allows code deep within a construct to jump to the end of that construct instantly, not executing anything except **unwind-protect**'s on the way. The construct which defines a static level that can be exited non-locally is called **block** and the construct which exits it is called **return-from**. The **block** being exited must be lexically visible from the **return-from** which says to exit it; this is what 'static' means. By contrast, **catch** and **throw** provide for dynamic non-local exits; refer to the following section. Here is an example of using a static non-local exit:

```
(block top
  (let ((v1 (do-1)))
    (when (all-done v1) (return-from top v1))
    (do-2))
  (do-3)
  ...
  (do-last))
```

If **(all-done v1)** returns non-nil, the entire **block** immediately returns the value of **v1**. Otherwise, the rest of the body of the block is executed sequentially, and ultimately the value or values of **(do-last)** are returned.

Note that the **return-from** form is very unusual: it does not ever return a value itself, in the conventional sense. It isn't useful to write **(setq a (return-from foo 3))**, because when the **return-from** form is evaluated, the containing **block** is immediately exited, and the **setq** never happens.

The fact that **block**'s and **return-from**'s are matched up lexically means you cannot do this:

```
(defun foo (a)
  (block foo1
    (bar a)))

(defun bar (x)
  (return-from foo1 x))
```

The **(return-from foo1 x)** gets an error because there is no lexically visible **block** named **foo1**. The suitable **block** in the caller, **foo**, is not even noticed.

Static handling allows the compiler to produce good code for **return-from**. It is also useful with functional arguments:

```
(defun first-symbol (list)
  (block done
    (mapc #'(lambda (elt)
              (if (symbolp elt) (return-from done elt)))
          list)))
```

The return-from done sees the block done lexically. Even if mapc had a block in it named done it would have no effect on the execution of first-symbol.

When a function is defined with defun with a name which is a symbol, a block whose name is the function name is automatically placed around the body of the function definition. For example,

```
(defun foo (a)
  (if (evenp a)
      (return-from foo (list a)))
  (1+ a))

(foo 4) => (4)
(foo 5) => 6
```

A function written explicitly with lambda does not have a block unless you write one yourself.

A named prog, or a do-named, implicitly defines a block with the specified name. So you can exit those constructs with return-from. In fact, the ability to name prog's was the original way to define a place for return-from to exit, before block was invented.

Every prog, do or loop, whether named or not, implicitly defines a block named nil. Thus, named prog's define *two* block's, one named nil and one named whatever name you specify. As a result, you can use return (an abbreviation for return-from nil) to return from the innermost lexically containing prog, do or loop (or from a block nil if you happen to write one). This function is like assq, but it returns an additional value which is the index in the table of the entry it found. For example,

```
(defun assqn (x table)
  (do ((l table (cdr l))
       (n 0 (1+ n)))
      ((null l) nil)
    (if (eq (caar l) x)
        (return (values (car l) n)))))
```

There is one exception to this: a prog, do or loop with name t defines only the block named t, no block named nil. The compiler used to make use of this feature in expanding certain built-in constructs into others.

block *name body...* *Special form*

Executes *body*, returning the values of the last form in *body*, but permitting non-local exit using **return-from** forms present lexically within *body*. *name* is not evaluated, and is used to match up **return-from** forms with their **block**'ss.

```
        (block foo
          (return-from foo 24) t) => 24
        (block foo t) => t
```

return-from *name values* *Special form*

Performs a non-local exit from the innermost lexically containing **block** whose name is *name*. *name* is not evaluated. When the compiler is used, **return-from**'s are matched up with **block**'s at compile time.

values is evaluated and its values become the values of the exited **block** form.

A **return-from** form may appear as or inside an argument to a regular function, but if the **return-from** is executed then the function will never actually be called. For example,

```
        (block done
          (foo (if a (return-from done t) nil)))
```

foo is actually called only if **a**'s value is **nil**. This style of coding is not recommended when **foo** is actually a function.

return-from can also be used with zero value forms, or with several value forms. Then *one* value is returned from each value form. Originally **return-from** always returned only one value from each value form, even when there was only one value form. Passing back all the values when there is a single *values* form is a later change, which is also the Common Lisp standard. In fact, the single value form case is much more powerful and subsumes all the others. For example,

```
        (return-from foo 1 2)
```

is equivalent to

```
        (return-from foo (values 1 2))
```

and

```
        (return-from foo)
```

is equivalent to

```
        (return-from foo (values))
```

It is unfortunate that the case of one value form is treated differently from all other cases, but the power of being able to propagate any number of values from a single form is worth it.

To return precisely one value, use **(return-from foo (values** *form***))**. It is legal to write simply **(return-from foo)**, which returns no values from the **block**. See section 3.7, page 55 for more information.

return *values* *Special form*

Is equivalent to **(return-from nil** *values***)**. It returns from a **block** whose name is **nil**.

In addition, **break** (see page 795) recognizes the typed-in form **(return** *value***)** specially. **break** evaluates *value* and returns it.

return-list *list* *Special form*

This function is like **return** except that each element of *list* is returned as a separate value from the **block** that is exited.

return-list is obsolete, since (return (values-list *list*)) does the same thing.

4.5 Tags and Gotos

Jumping to a label or *tag* is another kind of static non-local exit. Compared with **return-from**, it allows more flexibility in choosing where to send control to, but does not allow values to be sent along. This is because the tag does not have any way of saying what to *do* with any values.

To define a tag, the **tagbody** special form is used. In the body of a **tagbody**, all lists are treated as forms to be evaluated (called *statements* when they occur in this context). If no goto happens, all the forms are evaluated in sequence and then the **tagbody** form returns **nil**. Thus, the statements are evaluated only for effect.

An element of the **tagbody**'s body which is a symbol is not a statement but a tag instead. It identifies a place in the sequence of statements which you can go to. Going to a tag is accomplished by the form (**go** *tag*), executed at any point lexically within the **tagbody**.

go transfers control immediately to the first statement following *tag* in its **tagbody**, pausing only to deal with any **unwind-protect**s that are being exited as a result. If there are no more statements after *tag* in its **tagbody**, then that **tagbody** returns **nil** immediately.

All lexically containing **tagbody**'s are eligible to contain the specified tag, with the innermost **tagbody** taking priority. If no suitable tag is found, an error is signaled. The compiler matches **go**'s with tags at compile time and issues a compiler warning if no tag is found. Example:

```
(block nil
  (tagbody
    (setq x somefrob)
  loop
    do something
    (if somepredicate (go endtag))
    do something more
    (if (minusp x) (go loop))
  endtag
    (return z)))
```

is a kind of iteration made out of go-to's. This **tagbody** can never exit normally because the **return** in the last statement takes control away from it. This use of a **return** and **block** is how one encapsulates a **tagbody** to produce a non-**nil** value.

It works to **go** from an internal **lambda** function to a tag in a lexically containing function, as in

```
(defun foo (a)
  (tagbody
   t1
    (bar #'(lambda () (go t1)))))
```

If **bar** ever invokes its argument, control goes to **t1** and **bar** is invoked anew. Not very useful, but it illustrates the technique.

tagbody *statements-and-tags...* *Special form*

Executes all the elements of *statements-and-tags* which are lists (the statements), and then returns **nil**. But meanwhile, all elements of *statements-and-tags* which are symbols (the tags) are available for use with **go** in any of the statements. Atoms other than symbols are meaningless in a **tagbody**.

The reason that **tagbody** returns **nil** rather than the value of the last statement is that the designers of Common Lisp decided that one could not reliably return a value from the **tagbody** by writing it as the last statement since some of the time the expression for the desired value would be a symbol rather than a list, and then it would be taken as a tag rather than the last statement and it would not work.

go *tag* *Special form*

The **go** special form is used to "go-to" a tag defined in a lexically containing **tagbody** form (or other form which implicitly expands into a **tagbody**, such as **prog**, **do** or **loop**). *tag* must be a symbol. It is not evaluated.

prog *Special form*

prog is an archaic special form which provides temporary variables, static non-local exits, and tags for **go**. These aspects of **prog** were individually abstracted out to inspire **let**, **block** and **tagbody**. Now **prog** is obsolete, as it is much cleaner to use **let**, **block**, **tagbody** or all three of them, or **do** or **loop**. But **prog** appears in so many programs that it cannot be eliminated.

A typical **prog** looks like **(prog (***variables...***)** *body...***)**, which is equivalent to

```
(block nil
  (let (variables...)
    (tagbody body...)))
```

If the first subform of a **prog** is a non-**nil** symbol (rather than a list of variables), it is the name of the **prog**, and **return-from** (see page 77) can be used to return from it. A *named prog* looks like

```
(prog name (variables...) body...)
```

and is equivalent to

```
(block name
  (block nil
    (let (variables...)
      (tagbody body...))))
```

prog* *Special form*

The **prog*** special form is almost the same as **prog**. The only difference is that the binding and initialization of the temporary variables is done *sequentially*, so each one can depend on the previous ones. Thus, the equivalent code would use **let*** rather than **let**.

4.6 Dynamic Non-Local Exits

catch *tag body...* *Special form*

catch is a special form used with the **throw** function to do non-local exits. First *tag* is evaluated; the result is called the *tag* of the **catch**. Then the *body* forms are evaluated sequentially, and the values of the last form are returned. However, if, during the evaluation of the body, the function **throw** is called with the same tag as the tag of the catch, then the evaluation of the body is aborted, and the **catch** form immediately returns the values of the second argument to **throw** without further evaluating the current *body* form or the rest of the body.

The *tag*'s are used to match up **throw**'s with **catch**'s. (catch 'foo *form*) catches a (throw 'foo *form*) but not a (throw 'bar *form*). It is an error if **throw** is done when there is no suitable **catch** (or **catch-all**; see below).

Any Lisp object may be used as a **catch** tag. The values **t** and **nil** for *tag* are special: a **catch** whose tag is one of these values catches throws regardless of tag. These are only for internal use by **unwind-protect** and **catch-all** respectively. The only difference between **t** and **nil** is in the error checking; **t** implies that after a "cleanup handler" is executed control will be thrown again to the same tag, therefore it is an error if a specific catch for this tag does not exist higher up in the stack. With **nil**, the error check isn't done. Example:

```
(catch 'negative
  (values
    (mapcar #'(lambda (x)
                (cond ((minusp x)
                       (throw 'negative
                         (values x :negative)))
                      (t (f x)) )))
            y)
    :positive))
```

returns a list of f of each element of y, and :positive, if they are all positive, otherwise the first negative member of y, and :negative.

catch-continuation *tag throw-cont non-throw-cont body...* *Macro*
catch-continuation-if *cond-form tag throw-cont non-throw-cont body...* *Macro*

The catch-continuation special form makes it convenient to discriminate whether exit is normal or due to a throw.

The *body* is executed inside a **catch** on *tag* (which is evaluated). If *body* returns normally, the function *non-throw-cont* is called, passing all the values returned by the last form in *body* as arguments. This function's values are returned from the **catch-continuation**.

If on the other hand a throw to *tag* occurs, the values thrown are passed to the function *throw-cont*, and its values are returned.

If a continuation is explicitly written as **nil**, it is not called at all. The arguments that would have been passed to it are returned instead. This is equivalent to using **values** as the function; but explicit **nil** is optimized, so use that.

catch-continuation-if differs only in that the catch is not done if the value of the *cond-form* is **nil**. In this case, the non-throw continuation if any is always called.

In the general case, consing is necessary to record the multiple values, but if a continuation is an explicit **#'(lambda ...)** with a fixed number of arguments, or if a continuation is **nil**, it is open coded and the consing is avoided.

throw *tag values-form* *Special form*

throw is the primitive for exiting from a surrounding **catch**. *tag* is evaluated, and the result is matched (with **eq**) against the tags of all active **catch**'es; the innermost matching one is exited. If no matching **catch** is dynamically active, an error is signaled.

All the values of *values-form* are returned from the exited **catch**.

catch'es with tag **nil** always match any **throw**. They are really **catch-all**'s. So do **catch**'es with tag **t**, which are **unwind-protect**'s, but if the only matching **catch**'es are these then an error is signaled anyway. This is because an **unwind-protect** always throws again after its cleanup forms are finished; if there is nothing to catch after the last **unwind-protect**, an error will happen then, and it is better to detect the error sooner.

The values **t**, **nil**, and **0** for *tag* are reserved and used for internal purposes. **nil** may not be used, because it would cause confusion in handling of **unwind-protect**'s. **t** may only be used with ***unwind-stack**. **0** and **nil** are used internally when returning out of an **unwind-protect**.

***catch** *form tag* *Macro*
***throw** *form tag* *Macro*

Old, obsolete names for **catch** and **throw**.

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

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

:tag	The tag being thrown to.
:value	The value being thrown (the second argument to **throw**).
:count :action	The additional two arguments given to ***unwind-stack**, if that was used.

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

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

***unwind-stack** *tag value active-frame-count action*

This is a generalization of **throw** provided for program-manipulating programs such as the debugger.

tag and *value* are the same as the corresponding arguments to **throw**.

A *tag* of **t** invokes a special feature whereby the entire stack is unwound, and then the function *action* is called (see below). During this process **unwind-protect**'s receive control, but **catch-all**'s do not. This feature is provided for the benefit of system programs which want to unwind a stack completely.

active-frame-count, if non-**nil**, is the number of frames to be unwound. The definition of a frame is implementation-dependent. If this counts down to zero before a suitable **catch** is found, the ***unwind-stack** terminates and *that frame* returns *value* to whoever called it. This is similar to Maclisp's **freturn** function.

If *action* is non-nil, whenever the ***unwind-stack** would be ready to terminate (either due to *active-frame-count* or due to *tag* being caught as in **throw**), instead *action* is called with one argument, *value*. If *tag* is **t**, meaning throw out the whole way, then the function *action* is not allowed to return. Otherwise the function *action* may return and its value will be returned instead of *value* from the **catch**—or from an arbitrary function if *active-frame-count* is in use. In this case the **catch** does not return multiple values as it normally does when thrown to. Note that it is often useful for *action* to be a stack-group.

Note that if both *active-frame-count* and *action* are **nil**, ***unwind-stack** is identical to **throw**.

unwind-protect *protected-form cleanup-form...* *Special form*

Sometimes it is necessary to evaluate a form and make sure that certain side-effects take place after the form is evaluated; a typical example is:

```
(progn
   (turn-on-water-faucet)
   (hairy-function 3 nil 'foo)
   (turn-off-water-faucet))
```

The non-local exit facilities of Lisp create situations in which the above code won't work, however: if **hairy-function** should use **throw**, **return** or **go** to transfer control outside of the **progn** form, then **(turn-off-water-faucet)** will never be evaluated (and the faucet will presumably be left running). This is particularly likely if **hairy-function** gets an error and the user tells the debugger to give up and flush the computation.

In order to allow the above program to work, it can be rewritten using **unwind-protect** as follows:

```
(unwind-protect
    (progn (turn-on-water-faucet)
           (hairy-function 3 nil 'foo))
  (turn-off-water-faucet))
```

If **hairy-function** transfers control out of the evaluation of the **unwind-protect**, the **(turn-off-water-faucet)** form is evaluated during the transfer of control, before control arrives at the **catch**, **block** or **go** tag to which it is being transferred.

If the **progn** returns normally, then the **(turn-off-water-faucet)** is evaluated, and the **unwind-protect** returns the result of the **progn**.

The general form of **unwind-protect** looks like

```
(unwind-protect
    protected-form
  cleanup-form1
  cleanup-form2
  ...)
```

protected-form is evaluated, and when it returns or when it attempts to transfer control out of the **unwind-protect**, the *cleanup-forms* are evaluated. The value of the **unwind-protect** is the value of *protected-form*. Multiple values returned by the *protected-form* are propagated back through the **unwind-protect**.

The cleanup forms are run in the variable-binding environment that you would expect: that is, variables bound outside the scope of the **unwind-protect** special form can be accessed, but variables bound inside the *protected-form* can't be. In other words, the stack is unwound to the point just outside the *protected-form*, then the cleanup handler is run, and then the stack is unwound some more.

catch-all *body...* *Macro*

(catch-all *form***)** is like **(catch** *some-tag form***)** except that it catches a **throw** to any tag at all. Since the tag thrown to is one of the returned values, the caller of **catch-all** may continue throwing to that tag if he wants. The one thing that **catch-all** does not catch is a ***unwind-stack** with a tag of **t**. **catch-all** is a macro which expands into **catch** with a *tag* of **nil**.

catch-all returns all the values thrown to it, or returned by the body, plus three additional values: the tag thrown to, the active-frame-count, and the action. The tag value is **nil** if the body returned normally. The last two values are the third and fourth arguments to ***unwind-stack** (see page 82) if that was used, or **nil** if an ordinary **throw** was done or if the body returned normally.

If you think you want this, most likely you are mistaken and you really want **unwind-protect**.

4.7 Mapping

map *fcn* &rest *lists*
mapl *fcn* &rest *lists*
mapc *fcn* &rest *lists*
maplist *fcn* &rest *lists*
mapcar *fcn* &rest *lists*
mapcon *fcn* &rest *lists*
mapcan *fcn* &rest *lists*

Mapping is a type of iteration in which a function is successively applied to pieces of a list. There are several options for the way in which the pieces of the list are chosen and for what is done with the results returned by the applications of the function.

For example, mapcar operates on successive *elements* of the list. As it goes down the list, it calls the function giving it an element of the list as its one argument: first the car, then the cadr, then the caddr, etc., continuing until the end of the list is reached. The value returned by mapcar is a list of the results of the successive calls to the function. An example of the use of mapcar would be mapcar'ing the function abs over the list (1 -2 -4.5 6.0e15 -4.2), which would be written as (mapcar (function abs) '(1 -2 -4.5 6.0e15 -4.2)). The result is (1 2 4.5 6.0e15 4.2).

In general, the mapping functions take any number of arguments. For example,

```
(mapcar f x1 x2 ... xn)
```

In this case *f* must be a function of *n* arguments. mapcar proceeds down the lists *x1*, *x2*, ..., *xn* in parallel. The first argument to *f* comes from *x1*, the second from *x2*, etc. The iteration stops as soon as any of the lists is exhausted. (If there are no lists at all, then there are no lists to be exhausted, so *f* is called repeatedly without end. This is an obscure way to write an infinite loop. It is supported for consistency.) If you want to call a function of many arguments where one of the arguments successively takes on the values of the elements of a list and the other arguments are constant, you can use a circular list for the other arguments to mapcar. The function circular-list is useful for creating such lists; see page 93.

There are five other mapping functions besides mapcar. maplist is like mapcar except that the function is applied to the list and successive cdrs of that list rather than to successive elements of the list. map (or mapl) and mapc are like maplist and mapcar respectively, except that they don't return any useful value. These functions are used when the function is being called merely for its side-effects, rather than its returned values. mapcan and mapcon are like mapcar and maplist respectively, except that they combine the results of the function using nconc instead of list. That is, mapcon could have been defined by

```
(defun mapcon (f x y)
    (apply 'nconc (maplist f x y)))
```

Of course, this definition is less general than the real one.

Sometimes a do or a straightforward recursion is preferable to a map; however, the mapping functions should be used wherever they naturally apply because this increases the clarity of the code.

Often *f* is a lambda-expression, rather than a symbol; for example,
```
    (mapcar (function (lambda (x) (cons x something)))
            some-list)
```

The functional argument to a mapping function must be a function, acceptable to **apply**—it cannot be a macro or the name of a special form.

Here is a table showing the relations between the six map functions.

		applies function to	
		successive sublists	successive elements
	its own second argument	map(1)	mapc
returns	list of the function results	maplist	mapcar
	nconc of the function results	mapcon	mapcan

Note that **map** and **mapl** are synonymous. **map** is the traditional name of this function. **mapl** is the Common Lisp name. In Common Lisp, the function **map** does something different and incompatible; see **cli:map**, page 191. **mapl** works the same in traditional Zetalisp and Common Lisp.

There are also functions (**mapatoms** and **mapatoms-all**) for mapping over all symbols in certain packages. See the explanation of packages (chapter 27, page 636).

You can also do what the mapping functions do in a different way by using **loop**. See page 350.

5. Manipulating List Structure

This chapter discusses functions that manipulate conses, and higher-level structures made up of conses such as lists and trees. It also discusses hash tables and resources, which are related facilities.

A *cons* is a primitive Lisp data object that is extremely simple: it knows about two other objects, called its *car* and its *cdr*.

A list is recursively defined to be the symbol **nil**, or a cons whose cdr is a list. A typical list is a chain of conses: the cdr of each is the next cons in the chain, and the cdr of the last one is the symbol **nil**. The cars of each of these conses are called the *elements* of the list. A list has one element for each cons; the empty list, **nil**, has no elements at all. Here are the printed representations of some typical lists:

```
        (foo bar)                 ;This list has two elements.
        (a (b c d) e)             ;This list has three elements.
```

Note that the second list has three elements: **a**, **(b c d)**, and **e**. The symbols **b**, **c**, and **d** are *not* elements of the list itself. (They are elements of the list which is the second element of the original list.)

A *dotted list* is like a list except that the cdr of the last cons does not have to be **nil**. This name comes from the printed representation, which includes a "dot" character (period). Here is an example:

```
        (a b . c)
```

This dotted list is made of two conses. The car of the first cons is the symbol **a**, and the cdr of the first cons is the second cons. The car of the second cons is the symbol **b**, and the cdr of the second cons is the symbol **c**.

A tree is any data structure made up of conses whose cars and cdrs are other conses. The following are all printed representations of trees:

```
        (foo . bar)
        ((a . b) (c . d))
        ((a . b) (c d e f (g . 5) s) (7 . 4))
```

These definitions are not mutually exclusive. Consider a cons whose car is **a** and whose cdr is **(b (c d) e)**. Its printed representation is

```
        (a b (c d) e)
```

It can be thought of and treated as a cons, or as a list of four elements, or as a tree containing six conses. You can even think of it as a dotted list whose last cons just happens to have **nil** as a cdr. Thus, lists and dotted lists and trees are not fundamental data types; they are just ways of thinking about structures of conses.

A circular list is like a list except that the cdr of the last cons, instead of being **nil**, is the first cons of the list. This means that the conses are all hooked together in a ring, with the cdr of each cons being the next cons in the ring. These are legitimate Lisp objects, but dealing with them requires special techniques; straightforward tree-walking recursive functions often loop infinitely when given a circular list. The printer is is an example of both aspects of the handling of circular lists: if ***print-circle*** is non-**nil** the printer uses special techniques to detect circular structure and print it with a special encoding, but if *print-circle* is nil the printer does not check for circularity and loops infinitely unless *print-level* or *print-length* imposes a "time limit". See page 514 for more information on *print-circle* and related matters.

The Lisp Machine internally uses a storage scheme called *cdr-coding* to represent conses. This scheme is intended to reduce the amount of storage used in lists. The use of cdr-coding is invisible to programs except in terms of storage efficiency; programs work the same way whether or not lists are cdr-coded or not. Several of the functions below mention how they deal with cdr-coding. You can completely ignore all this if you want. However, if you are writing a program that allocates a lot of conses and you are concerned with storage efficiency, you may want to learn about the cdr-coded representation and how to control it. The cdr-coding scheme is discussed in section 5.4, page 100.

5.1 Conses

car *x*

Returns the car of *x*.
Example:

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

cdr *x*

Returns the cdr of *x*.
Example:

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

Officially car and cdr are only applicable to conses and locatives. However, as a matter of convenience, car and cdr of nil return nil. car or cdr of anything else is an error.

c...r *x*

All of the compositions of up to four car's and cdr's are defined as functions in their own right. The names of these functions begin with c and end with r, and in between is a sequence of a's and d's corresponding to the composition performed by the function.
Example:

`(cddadr x)` is the same as `(cdr (cdr (car (cdr x))))`

The error checking for these functions is exactly the same as for car and cdr above.

cons *x y*

cons is the primitive function to create a new cons, whose car is *x* and whose cdr is *y*.
Examples:

```
(cons 'a 'b) => (a . b)
(cons 'a (cons 'b (cons 'c nil))) => (a b c)
(cons 'a '(b c d)) => (a b c d)
```

ncons *x*

(ncons *x*) is the same as (cons *x* nil). The name of the function is from "nil-cons".

xcons *x y*

xcons ("exchanged cons") is like **cons** except that the order of the arguments is reversed.
Example:
```
        (xcons 'a 'b) => (b . a)
```

cons-in-area *x y area-number*

Creates a cons in a specific *area*. (Areas are an advanced feature of storage management, explained in chapter 16; if you aren't interested in them, you can safely skip all this stuff). The first two arguments are the same as the two arguments to **cons**, and the third is the number of the area in which to create the cons.
Example:
```
        (cons-in-area 'a 'b my-area) => (a . b)
```

ncons-in-area *x area-number*

(ncons-in-area *x area-number*) = (cons-in-area *x* nil *area-number*)

xcons-in-area *x y area-number*

(xcons-in-area *x y area-number*) = (cons-in-area *y x area-number*)

push *item place* *Macro*

Adds an element *item* to the front of a list that is stored in *place*. A new cons is allocated whose car is *item* and whose cdr is the old contents of *place*. This cons is stored into *place*.

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.

place can be any form that **setf** can store into. For example,
```
        (push x (get y z))
         ==> (putprop y (cons x (get y z)) z)
```

The returned value of **push** is not defined.

pop *place* *Macro*

Removes an element from the front of the list that is stored in *place*. It finds the cons in *place*, stores the cdr of the cons back into *place*, and returns the car of that cons. *place* can be any form that **setf** can store into.
Example:
```
        (setq x '(a b c))
        (pop x) => a
        x => (b c)
```

The backquote reader macro facility is also generally useful for creating list structure, especially mostly-constant list structure, or forms constructed by plugging variables into a template. It is documented in the chapter on macros; see chapter 18, page 320.

car-location *cons*

car-location returns a locative pointer to the cell containing the car of *cons*.

Note: there is no **cdr-location** function; it is difficult because of the cdr-coding scheme (see section 5.4, page 100). Instead, the cons itself serves as a kind of locative to its cdr (see page 267).

The functions **rplaca** and **rplacd** are used to make alterations in already-existing list structure; that is, to change the cars and cdrs of existing conses. The structure is altered rather than copied. Exercise caution when using these functions, as strange side-effects can occur if they are used to modify portions of list structure which have become shared unbeknownst to the programmer. The **nconc**, **nreverse**, **nreconc**, **nbutlast** and **delq** functions and others, described below, have the same property, because they call **rplaca** or **rplacd**.

rplaca *x y*

Changes the car of *x* to *y* and returns (the modified) *x*. *x* must be a cons or a locative. *y* may be any Lisp object.
Example:

```
(setq g '(a b c))
(rplaca (cdr g) 'd) => (d c)
Now g => (a d c)
```

rplacd *x y*

Changes the cdr of *x* to *y* and returns (the modified) *x*. *x* must be a cons or a locative. *y* may be any Lisp object.
Example:

```
(setq x '(a b c))
(rplacd x 'd) => (a . d)
Now x => (a . d)
```

(setf (car *x*) *y*) and (setf (car *x*) *y*) are much like **rplaca** and **rplacd**, but they return *y* rather than *x*.

5.2 Lists

list &rest *args*

Constructs and returns a list of its arguments.
Example:

```
(list 3 4 'a (car '(b . c)) (+ 6 -2)) => (3 4 a b 4)
```

list could have been defined by:

```
(defun list (&rest args)
    (let ((list (make-list (length args))))
      (do ((l list (cdr l))
           (a args (cdr a)))
          ((null a) list)
        (rplaca l (car a)))))
```

list* &rest *args*

list* is like list except that the last cons of the constructed list is dotted. It must be given at least one argument.
Example:
```
        (list* 'a 'b 'c 'd) => (a b c . d)
```
This is like
```
        (cons 'a (cons 'b (cons 'c 'd)))
```
More examples:
```
        (list* 'a 'b) => (a . b)
        (list* 'a) => a
```

length *list-or-array*

Returns the length of *list-or-array*. The length of a list is the number of elements in it; the number of times you can cdr it before you get a non-cons.
Examples:
```
        (length nil) => 0
        (length '(a b c d)) => 4
        (length '(a (b c) d)) => 3
        (length "foobar") => 6
```
length could have been defined by:
```
        (defun length (x)
          (if (arrayp x) (array-active-length x)
            (do ((n 0 (1+ n))
                 (y x (cdr y)))
                ((null y) n))))
```

list-length *list*

Returns the length of *list*, or nil if *list* is circular. (The function length would loop forever if given a circular list.)

first *list*
second *list*
third *list*
fourth *list*
fifth *list*
sixth *list*
seventh *list*

These functions take a list as an argument, and return the first, second, etc. element of the list. first is identical to car, second is identical to cadr, and so on. The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

rest *list*
rest1 *list*
rest2 *list*
rest3 *list*
rest4 *list*

rest*n* returns the rest of the elements of a list, starting with element *n* (counting the first element as the zeroth). Thus **rest** or **rest1** is identical to **cdr**, **rest2** is identical to **cddr**, and so on. The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

endp *list*

Returns **t** if *list* is **nil**, **nil** if *list* is a cons cell. Signals an error if *list* is not a list. This is the way Common Lisp recommends for terminating a loop which **cdr**'s down a list. However, Lisp Machine system functions generally prefer to test for the end of the list with **atom**; it is regarded as a feature that these functions do something useful for dotted lists.

nth *n list*

(**nth** *n list*) returns the *n*'th element of *list*, where the zeroth element is the car of the list. If *n* is greater than the length of the list, **nil** is returned.
Examples:

```
(nth 1 '(foo bar gack)) => bar
(nth 3 '(foo bar gack)) => nil
```

Note: this is not the same as the InterLisp function called **nth**, which is similar to but not exactly the same as the Lisp Machine function **nthcdr**. Also, some people have used macros and functions called **nth** of their own in their Maclisp programs, which may not work the same way; be careful.

nth could have been defined by:

```
(defun nth (n list)
  (do ((i n (1- i))
       (l list (cdr l)))
      ((zerop i) (car l))))
```

nthcdr *n list*

(**nthcdr** *n list*) cdrs *list* *n* times, and returns the result.
Examples:

```
(nthcdr 0 '(a b c)) => (a b c)
(nthcdr 2 '(a b c)) => (c)
```

In other words, it returns the *n*'th cdr of the list. If *n* is greater than the length of the list, **nil** is returned.

This is similar to InterLisp's function **nth**, except that the InterLisp function is one-based instead of zero-based; see the InterLisp manual for details. **nthcdr** could have been defined by:

```
(defun nthcdr (n list)
    (do ((i 0 (1+ i))
         (list list (cdr list)))
        ((= i n) list)))
```

last *list*

last returns the last cons of *list*. If *list* is nil, it returns nil. Note that last is unfortunately *not* analogous to first (first returns the first element of a list, but last doesn't return the last element of a list); this is a historical artifact.
Examples:

```
        (setq x '(a b c d))
        (last x) => (d)
        (rplacd (last x) '(e f))
        x => '(a b c d e f)
```

last could have been defined by:

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

list-match-p *object pattern* *Macro*

object is evaluated and matched against *pattern*; the value is t if it matches, nil otherwise. *pattern* is made with backquotes (section 18.2.2, page 325); whereas normally a backquote expression says how to construct list structure out of constant and variable parts, in this context it says how to match list structure against constants and variables. Constant parts of the backquote expression must match exactly; variables preceded by commas can match anything but set the variable to what was matched. (Some of the variables may be set even if there is no match.) If a variable appears more than once, it must match the same thing (equal list structures) each time. ,ignore can be used to match anything and ignore it.

For example, '(x (,y) . ,z) is a pattern that matches a list of length at least two whose first element is x and whose second element is a list of length one; if a list matches, the caadr of the list is stored into the value of *y* and the cddr of the list is stored into *z*.

Variables set during the matching remain set after the list-match-p returns; in effect, list-match-p expands into code which can setq the variables. If the match fails, some or all of the variables may already have been set.

Example:

```
        (list-match-p foo
                      '((a ,x) ,ignore . ,c))
```

is t if foo's value is a list of two or more elements, the first of which is a list of two elements; and in that case it sets x to (cadar foo) and c to (cddr foo). An equivalent expression would be

```
(let ((tem foo))
  (and (consp tem)
       (consp (car tem))
       (eq (caar tem) 'a)
       (consp (cdar tem))
       (progn (setq x (cadar tem)) t)
       (null (cddar tem))
       (consp (cdr tem))
       (setq c (cddr tem))))
```

but **list-match-p** is faster.

list-match-p generates highly optimized code using special instructions.

list-in-area *area-number* &rest *args*

list-in-area is exactly the same as list except that it takes an extra argument, an area number, and creates the list in that area.

list*-in-area *area-number* &rest *args*

list*-in-area is exactly the same as list* except that it takes an extra argument, an area number, and creates the list in that area.

make-list *length* &key *area initial-element*

Creates and returns a list containing *length* elements. *length* should be a fixnum. *area*, if specified, is the area in which to create the list (see chapter 16, page 296). If it is nil, the area used is the value of working-storage-area.

initial-element is stored in each element of the new list.

make-list always creates a cdr-coded list (see section 5.4, page 100).
Examples:

```
(make-list 3) => (nil nil nil)
(make-list 4 :initial-element 7) => (7 7 7 7)
```

The keyword :initial-value may be used in place of :initial-element.

When make-list was originally implemented, it took exactly two arguments: the area and the length. This obsolete form is still supported so that old programs can continue to work, but the new keyword-argument form is preferred.

circular-list &rest *args*

Constructs a circular list whose elements are args, repeated infinitely. circular-list is the same as list except that the list itself is used as the last cdr, instead of nil. circular-list is especially useful with mapcar, as in the expression

```
(mapcar (function +) foo (circular-list 5))
```

which adds each element of foo to 5.

circular-list could have been defined by:

```
(defun circular-list (&rest elements)
  (setq elements (copylist* elements))
  (rplacd (last elements) elements)
  elements)
```

copylist *list* &optional *area*
copy-list *list* &optional *area*

Returns a list which is equal to *list*, but not eq. copylist does not copy any elements of the list, only the conses of the list itself. The returned list is fully cdr-coded (see section 5.4, page 100) to minimize storage. If *list* is dotted, that is, if (cdr (last *list*)) is a non-nil atom, then the copy also has this property. You may optionally specify the area in which to create the new copy.

copylist* *list* &optional *area*

This is the same as copylist except that the last cons of the resulting list is never cdr-coded (see section 5.4, page 100). This makes for increased efficiency if you nconc something onto the list later.

copyalist *list* &optional *area*
copy-alist *list* &optional *area*

copyalist is for copying association lists (see section 5.5, page 102). The *list* is copied, as in copylist. In addition, each element of *list* which is a cons is replaced in the copy by a new cons with the same car and cdr. You may optionally specify the area in which to create the new copy.

append &rest *lists*

The arguments to append are lists. The result is a list which is the concatenation of the arguments. The arguments are not changed (cf. nconc).
Example:

```
(append '(a b c) '(d e f) nil '(g)) => (a b c d e f g)
```

append makes copies of the conses of all the lists it is given, except for the last one. So the new list shares the conses of the last argument to append, but all of the other conses are newly created. Only the lists are copied, not the elements of the lists.

A version of append which only accepts two arguments could have been defined by:

```
(defun append2 (x y)
    (cond ((null x) y)
          ((cons (car x) (append2 (cdr x) y)) )))
```

The generalization to any number of arguments could then be made (relying on car of nil being nil):

```
(defun append (&rest args)
  (if (< (length args) 2) (car args)
      (append2 (car args)
               (apply (function append) (cdr args)))))
```

These definitions do not express the full functionality of **append**; the real definition minimizes storage utilization by turning all the arguments that are copied into one cdr-coded list.

To copy a list, use **copylist** (see page 94); the old practice of using **append** to copy lists is unclear and obsolete.

nconc &rest *lists*

nconc takes lists as arguments. It returns a list which is the arguments concatenated together. The arguments are changed, rather than copied (cf. **append**, page 94).
Example:

```
(setq x '(a b c))
(setq y '(d e f))
(nconc x y) => (a b c d e f)
x => (a b c d e f)
```

Note that the value of x is now different, since its last cons has been rplacd'd to the value of y. If the **nconc** form were evaluated again, it would yield a piece of circular list structure, whose printed representation would be (a b c d e f d e f d e f ...), repeating forever.

nconc could have been defined by:

```
(defun nconc (x y)                  ; for simplicity, this definition
    (cond ((null x) y)              ; only works for 2 arguments.
          (t (rplacd (last x) y)    ;hook y onto x
             x)))                   ; and return the modified x.
```

revappend *x y*

(revappend *x y*) is exactly the same as (nconc (reverse *x*) *y*) except that it is more efficient. Both *x* and *y* should be lists.

revappend could have been defined by:

```
(defun revappend (x y)
    (cond ((null x) y)
          (t (revappend (cdr x) (cons (car x) y)))))
```

nreconc *x y*

(nreconc *x y*) is exactly the same as (nconc (nreverse *x*) *y*) except that it is more efficient. Both *x* and *y* should be lists.

nreconc could have been defined by:

```
(defun nreconc (x y)
    (cond ((null x) y)
          ((nreverse1 x y)) ))
```

using the same nreverse1 as above.

butlast *list* &optional (*n* 1)

This creates and returns a list with the same elements as *list*, excepting the last *n* elements.
Examples:
```
        (butlast '(a b c d)) => (a b c)
        (butlast '(a b c d) 3) => (a)
        (butlast '(a b c d) 4) => nil
        (butlast nil) => nil
```
The name is from the phrase "all elements but the last".

nbutlast *list* &optional (*n* 1)

This is the destructive version of **butlast**; it changes the cdr of the last cons but *n* of the list to **nil**. The value is *list*, as modified. If *list* does not have more than *n* elements then it is not really changed and the value is **nil**.
Examples:
```
        (setq foo '(a b c d))
        (nbutlast foo) => (a b c)
        foo => (a b c)
        (nbutlast foo 2) => (a)
        foo => (a)
        (nbutlast foo) => nil
        foo => (a)
```

firstn *n list*

Returns a list of length *n*, whose elements are the first *n* elements of **list**. If *list* is fewer than *n* elements long, the remaining elements of the returned list are **nil**.
Examples:
```
        (firstn 2 '(a b c d)) => (a b)
        (firstn 0 '(a b c d)) => nil
        (firstn 6 '(a b c d)) => (a b c d nil nil)
```

nleft *n list* &optional *tail*

Returns a "tail" of *list*, i.e. one of the conses that makes up *list*, or **nil**. (**nleft** *n list*) returns the last *n* elements of *list*. If *n* is too large, **nleft** returns *list*.

(**nleft** *n list tail*) takes cdr of *list* enough times that taking *n* more cdrs would yield *tail*, and returns that. You can see that when *tail* is **nil** this is the same as the two-argument case. If *tail* is not **eq** to any tail of *list*, **nleft** returns **nil**.
Examples:
```
        (setq x '(a b c d e f))
        (nleft 2 x) => (e f)
        (nleft 2 x (cddddr x)) => (c d e f)
```

ldiff *list tail*

list should be a list, and *tail* should be one of the conses that make up *list*. **ldiff** (meaning 'list difference') returns a new list, whose elements are those elements of *list* that appear before *tail*.

Examples:
```
        (setq x '(a b c d e))
        (setq y (cdddr x)) => (d e)
        (ldiff x y) => (a b c)
        (ldiff x nil) => (a b c d e)
        (ldiff x x) => nil
```
but
```
        (ldiff '(a b c d) '(c d)) => (a b c d)
```
since the *tail* was not eq to any part of the *list*.

car-safe *object*
cdr-safe *object*
caar-safe *object*
cadr-safe *object*
cdar-safe *object*
cddr-safe *object*
caddr-safe *object*
cdddr-safe *object*
cadddr-safe *object*
cddddr-safe *object*
nth-safe *n object*
nthcdr-safe *n object*

Return the same things as the corresponding non-safe functions, except nil if the non-safe function would get an error. These functions are about as fast as the non-safe functions. The same effect could be had by handling the sys:wrong-type-argument error, but that would be slower. Examples:
```
        (car-safe '(a . b)) => a
        (car-safe nil) => nil
        (car-safe 'a) => nil
        (car-safe "foo") => nil
        (cadr-safe '(a . b)) => nil
        (cadr-safe 3) => nil
```

5.3 Cons Cells as Trees

copytree *tree* &optional *area*
copy-tree *tree* &optional *area*

copytree copies all the conses of a tree and makes a new maximally cdr-coded tree with the same fringe. If *area* is specified, the new tree is constructed in that area.

tree-equal *x y* &key *test test-not*

Compares two trees recursively to all levels. Atoms must match under the function *test* (which defaults to eql). Conses must match recursively in both the car and the cdr.

If *test-not* is specified instead of *test*, two atoms match if *test-not* returns nil.

subst *new old tree*

(subst *new old tree*) substitutes *new* for all occurrences of *old* in *tree*, and returns the modified copy of *tree*. The original *tree* is unchanged, as subst recursively copies all of *tree* replacing elements equal to *old* as it goes.
Example:

```
(subst 'Tempest 'Hurricane
       '(Shakespeare wrote (The Hurricane)))
     => (Shakespeare wrote (The Tempest))
```

subst could have been defined by:

```
(defun subst (new old tree)
    (cond ((equal tree old) new)  ;if item equal to old, replace.
          ((atom tree) tree)      ;if no substructure, return arg.
          ((cons (subst new old (car tree))  ;otherwise recurse.
                 (subst new old (cdr tree))))))
```

Note that this function is not destructive; that is, it does not change the car or cdr of any already-existing list structure.

To copy a tree, use copytree (see page 97); the old practice of using subst to copy trees is unclear and obsolete.

cli:subst *new old tree* &key *test test-not key*

The Common Lisp version of subst replaces with *new* every atom or subtree in *tree* which matches *old*, returning a new tree. List structure is copied as necessary to avoid clobbering parts of tree. This differs from the traditional subst function, which always copies the entire tree.

test or *test-not* is used to do the matching. If *test* is specified, a match happens when *test* returns non-nil; otherwise, if *test-not* is specified, a match happens when it returns nil. If neither is specified, then eql is used for *test*.

The first argument to the *test* or *test-not* function is always *old*. The second argument is normally a leaf or subtree of *tree*. However, if *key* is non-nil, then it is called with the subtree as argument, and the result of this becomes the second argument to the *test* or *test-not* function.

Because (subst nil nil *tree*) is a widely used idiom for copying a tree, even though it is obsolete, there is no practical possibility of installing this function as the standard subst for a long time.

nsubst *new old tree* &key *test test-not key*

nsubst is a destructive version of subst. The list structure of *tree* is altered by replacing each occurrence of *old* with *new*. No new list structure is created. The keyword arguments are as in cli:subst.

A simplified version of nsubst, handling only the three required arguments, could be defined as

```
(defun nsubst (new old tree)
    (cond ((eql tree old) new)      ; If item matches old, replace.
          ((atom tree) tree)        ; If no substructure, return arg.
          (t                        ; Otherwise, recurse.
             (rplaca tree (nsubst new old (car tree)))
             (rplacd tree (nsubst new old (cdr tree)))
             tree)))
```

subst-if *new predicate tree* &key *key*

Replaces with *new* every atom or subtree in *tree* which satisfies *predicate*. List structure is copied as necessary so that the original tree is not modified. *key*, if non-nil, is a function applied to each tree node to get the object to match against. If *key* is nil or omitted, the tree node itself is used.

subst-if-not *new predicate tree* &key *key*

Similar, but replaces tree nodes which *do not* satisfy *predicate*.

nsubst-if *new predicate tree* &key *key*
nsubst-if-not *new predicate tree* &key *key*

Like subst-if and subst-if-not except that they destructively modify *tree* itself and return it, creating no new list structure.

sublis *alist tree* &key *test test-not key*

Performs multiple parallel replacements on *tree*, returning a new tree. *tree* itself is not modified because list structure is copied as necessary. If no substitutions are made, the result is *tree*. *alist* is an association list (see section 5.5, page 102). Each element of *alist* specifies one replacement; the car is what to look for, and the cdr is what to replace it with.

test, *test-not* and *key* control how matching is done between nodes of the tree (cons cells or atoms) and objects to be replaced. See cli:subst, above, for the details of how they work. The first argument to *test* or *test-not* is the car of an element of *alist*.
Example:

```
(sublis '((x . 100) (z . zprime))
        '(plus x (minus g z x p) 4))
   => (plus 100 (minus g zprime 100 p) 4)
```

A simplified sublis could be defined by:

```
(defun sublis (alist tree)
  (let ((tem (assq tree alist)))
    (cond (tem (cdr tem))
          ((atom tree) tree)
          (t
           (let ((car (sublis alist (car tree)))
                 (cdr (sublis alist (cdr tree))))
             (if (and (eq (car tree) car) (eq (cdr tree) cdr))
                 tree
               (cons car cdr)))))))
```

nsublis *alist tree* &key *test test-not key*

nsublis is like sublis but changes the original tree instead of allocating new structure.

A simplified nsublis could be defined by:

```
(defun nsublis (alist tree)
  (let ((tem (assq tree alist)))
    (cond (tem (cdr tem))
          ((atom tree) tree)
          (t (rplaca tree (nsublis alist (car tree)))
             (rplacd tree (nsublis alist (cdr tree)))
             tree))))
```

5.4 Cdr-Coding

This section explains the internal data format used to store conses inside the Lisp Machine. Casual users don't have to worry about this; you can skip this section if you want. It is only important to read this section if you require extra storage efficiency in your program.

The usual and obvious internal representation of conses in any implementation of Lisp is as a pair of pointers, contiguous in memory. If we call the amount of storage that it takes to store a Lisp pointer a 'word', then conses normally occupy two words. One word (say it's the first) holds the car, and the other word (say it's the second) holds the cdr. To get the car or cdr of a list, you just reference this memory location, and to change the car or cdr, you just store into this memory location.

Very often, conses are used to store lists. If the above representation is used, a list of *n* elements requires two times *n* words of memory: *n* to hold the pointers to the elements of the list, and *n* to point to the next cons or to nil. To optimize this particular case of using conses, the Lisp Machine uses a storage representation called *cdr-coding* to store lists. The basic goal is to allow a list of *n* elements to be stored in only *n* locations, while allowing conses that are not parts of lists to be stored in the usual way.

The way it works is that there is an extra two-bit field in every word of memory, called the *cdr-code* field. There are three meaningful values that this field can have, which are called cdr-normal, cdr-next, and cdr-nil. The regular, non-compact way to store a cons is by two contiguous words, the first of which holds the car and the second of which holds the cdr. In this case, the cdr-code of the first word is cdr-normal. (The cdr-code of the second word doesn't matter; as we will see, it is never looked at.) The cons is represented by a pointer to the first of the two words. When a list of *n* elements is stored in the most compact way, pointers to the *n* elements occupy *n* contiguous memory locations. The cdr-codes of all these locations are cdr-next, except the last location whose cdr-code is cdr-nil. The list is represented as a pointer to the first of the *n* words.

Now, how are the basic operations on conses defined to work based on this data structure? Finding the car is easy: you just read the contents of the location addressed by the pointer. Finding the cdr is more complex. First you must read the contents of the location addressed by the pointer, and inspect the cdr-code you find there. If the code is cdr-normal, then you add one to the pointer, read the location it addresses, and return the contents of that location; that is, you read the second of the two words. If the code is **cdr-next**, you add one to the pointer, and simply return that pointer without doing any more reading; that is, you return a pointer to the next word in the *n*-word block. If the code is **cdr-nil**, you simply return **nil**.

If you examine these rules, you will find that they work fine even if you mix the two kinds of storage representation within the same list.

How about changing the structure? Like **car**, **rplaca** is very easy; you just store into the location addressed by the pointer. To do **rplacd** you must read the location addressed by the pointer and examine the cdr-code. If the code is **cdr-normal**, you just store into the location one greater than that addressed by the pointer; that is, you store into the second word of the two words. But if the cdr-code is **cdr-next** or **cdr-nil**, there is a problem: there is no memory cell that is storing the cdr of the cons. That is the cell that has been optimized out; it just doesn't exist.

This problem is dealt with by the use of *invisible pointers*. An invisible pointer is a special kind of pointer, recognized by its data type (Lisp Machine pointers include a data type field as well as an address field). The way they work is that when the Lisp Machine reads a word from memory, if that word is an invisible pointer then it proceeds to read the word pointed to by the invisible pointer and use that word instead of the invisible pointer itself. Similarly, when it writes to a location, it first reads the location, and if it contains an invisible pointer then it writes to the location addressed by the invisible pointer instead. (This is a somewhat simplified explanation; actually there are several kinds of invisible pointer that are interpreted in different ways at different times, used for things other than the cdr-coding scheme.)

Here's how to do **rplacd** when the cdr-code is **cdr-next** or **cdr-nil**. Call the location addressed by the first argument to **rplacd** *l*. First, you allocate two contiguous words in the same area that *l* points to. Then you store the old contents of *l* (the car of the cons) and the second argument to **rplacd** (the new cdr of the cons) into these two words. You set the cdr-code of the first of the two words to **cdr-normal**. Then you write an invisible pointer, pointing at the first of the two words, into location *l*. (It doesn't matter what the cdr-code of this word is, since the invisible pointer data type is checked first, as we will see.)

Now, whenever any operation is done to the cons (**car**, **cdr**, **rplaca**, or **rplacd**), the initial reading of the word pointed to by the Lisp pointer that represents the cons finds an invisible pointer in the addressed cell. When the invisible pointer is seen, the address it contains is used in place of the original address. So the newly-allocated two-word cons is used for any operation done on the original object.

Why is any of this important to users? In fact, it is all invisible to you; everything works the same way whether or not compact representation is used, from the point of view of the semantics of the language. That is, the only difference that any of this makes is a difference in efficiency. The compact representation is more efficient in most cases. However, if the conses are going to get **rplacd**'ed, then invisible pointers will be created, extra memory will be allocated, and the compact representation will degrade storage efficiency rather than improve it. Also, accesses that go through invisible pointers are somewhat slower, since more memory references are needed. So if you care a lot about storage efficiency, you should be careful about which lists get stored in which representations.

You should try to use the normal representation for those data structures that will be subject to rplacd operations, including nconc and nreverse, and the compact representation for other structures. The functions cons, xcons, ncons, and their area variants make conses in the normal representation. The functions list, list*, list-in-area, make-list, and append use the compact representation. The other list-creating functions, including read, currently make normal lists, although this might get changed. Some functions, such as sort, take special care to operate efficiently on compact lists (sort effectively treats them as arrays). nreverse is rather slow on compact lists, currently, since it simple-mindedly uses rplacd, but this may be changed.

(copylist *x*) is a suitable way to copy a list, converting it into compact form (see page 94).

5.5 Tables

Zetalisp includes functions which simplify the maintenance of tabular data structures of several varieties. The simplest is a plain list of items There are functions to add (cons), remove (delete, delq, del, del-if, del-if-not, remove, remq, rem, rem-if, rem-if-not), and search for (member, memq, mem) items in a list.

Association lists are very commonly used. An association list is a list of conses. The car of each cons is a "key" and the cdr is a "datum", or a list of associated data. The functions assoc, assq, ass, memass, and rassoc may be used to retrieve the data, given the key. For example,

```
        ((tweety . bird) (sylvester . cat))
```

is an association list with two elements. Given a symbol representing the name of an animal, it can retrieve what kind of animal this is.

Structured records can be stored as association lists or as stereotyped cons-structures where each element of the structure has a certain car-cdr path associated with it. However, these are better implemented using structure macros (see chapter 20, page 372) or as flavors (chapter 21, page 401).

Simple list-structure is very convenient, but may not be efficient enough for large data bases because it takes a long time to search a long list. Zetalisp includes hash table facilities for more efficient but more complex tables (see section 5.11, page 116), and a hashing function (sxhash) to aid users in constructing their own facilities.

5.6 Lists as Tables

memq *item list*

Returns nil if *item* is not one of the elements of *list*. Otherwise, it returns the sublist of *list* beginning with the first occurrence of *item*; that is, it returns the first cons of the list whose car is *item*. The comparison is made by eq. Because memq returns nil if it doesn't find anything, and something non-nil if it finds something, it is often used as a predicate.

Examples:

```
(memq 'a '(1 2 3 4)) => nil
(memq 'a '(g (x a y) c a d e a f)) => (a d e a f)
```

Note that the value returned by memq is eq to the portion of the list beginning with a.

Thus **rplaca** on the result of **memq** may be used, if you first check to make sure **memq** did not return **nil**.
Example:

```
        (let ((sublist (memq x z)))       ;Search for x in the list z.
          (if (not (null sublist))        ;If it is found,
              (rplaca sublist y)))        ;Replace it with y.
```

memq could have been defined by:

```
        (defun memq (item list)
            (cond ((null list) nil)
                  ((eq item (car list)) list)
                  (t (memq item (cdr list)))))
```

memq is hand-coded in microcode and therefore especially fast. It is equivalent to **cli:member** with **eq** specified as the *test* argument.

member *item list*
member is like **memq**, except **equal** is used for the comparison, instead of **eq**. Note that the **member** function of Common Lisp, which is **cli:member**, is similar but thoroughly incompatible (see below).

member could have been defined by:

```
        (defun member (item list)
            (cond ((null list) nil)
                  ((equal item (car list)) list)
                  (t (member item (cdr list)))))
```

cli:member *item list* &key *test test-not key*
The Common Lisp **member** function. It is like **memq** or **member** except that there is more generality in how elements of *list* are matched against *item*—and the default is incompatible.

test, *test-not* and *key* are used in matching the elements, just as described under **cli:subst** (see page 98). If neither *test* nor *test-not* is specified, the default is to compare with **eql**, whereas **member** compares with **equal**.

Usually *test* is a commutative predicate such as **eq**, **equal**, **=**, **char-equal** or **string-equal**. It can also be a non-commutative predicate. The predicate is called with *item* as its first argument and the element of *list* as its second argument. Example:

```
        (cli:member 4 '(1.5 2.5 2 3.5 4.5 8) :test '<) => (4.5 8)
```

member-if *predicate list* &key *key*
Searches the elements of *list* for one which satisfies *predicate*. If one is found, the value is the tail of *list* whose car is that element. Otherwise the value is **nil**.

If *key* is non-**nil**, then *predicate* is applied to (**funcall** *key element*) rather than to the element itself.

member-if-not *predicate list* &key *key*

Searches for an element which does not satisfy *predicate*. Otherwise like **member-if**.

mem *predicate item list*

Is equivalent to

```
(cli:member item list :test predicate)
```

The function **mem** antedates **cli:member**.

find-position-in-list *item list*

Searches *list* for an element which is **eq** to *item*, like **memq**. However, it returns the numeric index in the list at which it found the first occurence of *item*, or **nil** if it did not find it at all. This function is sort of the complement of **nth** (see page 91); like **nth**, it is zero-based.

Examples:

```
(find-position-in-list 'a '(a b c)) => 0
(find-position-in-list 'c '(a b c)) => 2
(find-position-in-list 'e '(a b c)) => nil
```

See also the generic sequence function **position** (page 198).

find-position-in-list-equal *item list*

Is like **find-position-in-list**, except that the comparison is done with **equal** instead of **eq**.

tailp *sublist list*

Returns **t** if *sublist* is a sublist of *list* (i.e. one of the conses that makes up *list*). Otherwise returns **nil**. Another way to look at this is that **tailp** returns **t** if (**nthcdr** *n list*) is *sublist*, for some value of *n*. **tailp** could have been defined by:

```
(defun tailp (sublist list)
    (do list list (cdr list) (null list)
      (if (eq sublist list)
          (return t))))
```

delq *item list* &optional *n*

(**delq** *item list*) returns the *list* with all occurrences of *item* removed. **eq** is used for the comparison. The argument *list* is actually modified (**rplacd**'ed) when instances of *item* are spliced out. **delq** should be used for value, not for effect. That is, use

```
(setq a (delq 'b a))
```

rather than

```
(delq 'b a)
```

These two are *not* equivalent when the first element of the value of **a** is **b**.

(**delq** *item list n*) is like (**delq** *item list*) except only the first *n* instances of *item* are deleted. *n* is allowed to be zero. If *n* is greater than or equal to the number of occurrences of *item* in the list, all occurrences of *item* in the list are deleted.

Example:

```
(delq 'a '(b a c (a b) d a e)) => (b c (a b) d e)
```

delq could have been defined by:
```
(defun delq (item list &optional (n -1))
    (cond ((or (atom list) (zerop n)) list)
          ((eq item (car list))
           (delq item (cdr list) (1- n)))
          (t (rplacd list (delq item (cdr list) n)))))
```
If the third argument (*n*) is not supplied, it defaults to -1 which is effectively infinity since it can be decremented any number of times without reaching zero.

delete *item list* &optional *n*
delete is the same as delq except that equal is used for the comparison instead of eq.

Common Lisp programs have a different, incompatible function called delete; see page 195. This function may be useful in non-Common-Lisp programs as well, where it can be referred to as cli:delete.

del *predicate item list* &optional *n*
del is the same as delq except that it takes an extra argument which should be a predicate of two arguments, which is used for the comparison instead of eq. (del 'eq a b) is the same as (delq a b). See also mem, page 104.

Use of del is equivalent to
```
(cli:delete item list :test predicate)
```

remq *item list* &optional *n*
remq is similar to delq, except that the list is not altered; rather, a new list is returned. Examples:
```
(setq x '(a b c d e f))
(remq 'b x) => (a c d e f)
x => (a b c d e f)
(remq 'b '(a b c b a b) 2) => (a c a b)
```

remove *item list* &optional *n*
remove is the same as remq except that equal is used for the comparison instead of eq. Common Lisp programs have a different, incompatible function called remove; see page 195. This function may be useful in non-Common-Lisp programs as well, where it can be referred to as cli:remove.

rem *predicate item list* &optional *n*
rem is the same as remq except that it takes an extra argument which should be a predicate of two arguments, which is used for the comparison instead of eq. (rem 'eq a b) is the same as (remq a b). See also mem, page 104.

The function rem in Common Lisp programs is actually cli:rem, a remainder function. See page 144.

subset *predicate list*
rem-if-not *predicate list*

predicate should be a function of one argument. A new list is made by applying *predicate* to all of the elements of *list* and removing the ones for which the predicate returns nil. One of this function's names (rem-if-not) means "remove if this condition is not true"; i.e. it keeps the elements for which *predicate* is true. The other name (subset) refers to the function's action if *list* is considered to represent a mathematical set.
Example:

```
        (subset #'minusp '(1 2 -4 2 -3)) => (-4 -3)
```

subset-not *predicate list*
rem-if *predicate list*

predicate should be a function of one argument. A new list is made by applying *predicate* to all of the elements of *list* and removing the ones for which the predicate returns non-nil. One of this function's names (rem-if) means "remove if this condition is true". The other name (subset-not) refers to the function's action if *list* is considered to represent a mathematical set.

del-if *predicate list*

del-if is just like rem-if except that it modifies *list* rather than creating a new list.

del-if-not *predicate list*

del-if-not is just like rem-if-not except that it modifies *list* rather than creating a new list.

See also the generic sequence functions delete-if, delete-if-not, remove-if and remove-if-not (page 194).

every *list predicate* &optional *step-function*

Returns t if *predicate* returns non-nil when applied to every element of *list*, or nil if *predicate* returns nil for some element. If *step-function* is present, it replaces cdr as the function used to get to the next element of the list; cddr is a typical function to use here.

In Common Lisp programs, the name every refers to a different, incompatible function which serves a similar purpose. It is documented in the manual under the name cli:every. See page 192.

some *list predicate* &optional *step-function*

Returns a tail of *list* such that the car of the tail is the first element that the *predicate* returns non-nil when applied to, or nil if *predicate* returns nil for every element. If *step-function* is present, it replaces cdr as the function used to get to the next element of the list; cddr is a typical function to use here.

In Common Lisp programs, the name some refers to a different, incompatible function which serves a similar purpose. It is documented in the manual under the name cli:some. See page 191.

5.7 Lists as Sets

A list can be used to represent an unordered set of objects, simply by using it in a way that ignores the order of the elements. Membership in the set can be tested with **memq** or **member**, and some other functions in the previous section also make sense on lists representing sets. This section describes several functions specifically intended for lists that represent sets.

It is often desirable to avoid adding duplicate elements in the list. The set functions attempt to introduce no duplications, but do not attempt to eliminate duplications present in their arguments. If you need to make absolutely certain that a list contains no duplicates, use **remove-duplicates** or **delete-duplicates** (see page 196).

subsetp *list1 list2* &key *test test-not key*

t if every element of *list1* matches some element of *list2*.

The keyword arguments control how matching is done. Either *test* or *test-not* should be a function of two arguments. Normally it is called with an element of *list1* as the first argument and an element of *list2* as the second argument. If *test* is specified, a match happens when *test* returns non-nil; otherwise, if *test-not* is specified, a match happens when it returns nil. If neither is specified, then eql is used for *test*.

If *key* is non-nil, it should be a function of one argument. *key* is applied to each list element to get a key to be passed to *test* or *test-not* instead of the element.

adjoin *item list* &key *test test-not key*

Returns a list like *list* but with *item* as an additional element if no existing element matches item. It is done like this:

```
    (if (cli:member (if key (funcall key item) item)
                    list other-args...)
        list
      (cons item list))
```

The keyword arguments work as in **subsetp**.

pushnew *item list-place* &key *test test-not key* *Macro*

Pushes *item* onto *list-place* unless *item* matches an existing element of the value stored in that place. Equivalent to

```
    (setf list-place
          (adjoin item list-place keyword-args...))
```

except for order of evaluation. Compare with **push**, page 88.

union *list* &rest *more-lists*

Returns a list representing the set which is the union of the sets represented by the arguments. Anything which is an element of at least one of the arguments is also an element of the result.

Each element of each list is compared, using **eq**, with all elements of the other lists, to make sure that no duplications are introduced into the result. As long as no individual argument list contains duplications, the result does not either.

It is best to use **union** with only two arguments so that your code will not be sensitive to the difference between the traditional version of **union** and the Common Lisp version **cli:union**, below.

intersection *list* &rest *more-lists*

If lists are regarded as sets of their elements, **intersection** returns a list which is the intersection of the lists which are supplied as arguments. If *list* contains no duplicate elements, neither does the value returned by **intersection**. Elements are compared using **eq**.

It is best to use **intersection** with only two arguments so that your code will not be sensitive to the difference between the traditional version of **intersection** and the Common Lisp version **cli:intersection**, below.

nunion *list* &rest *more-lists*

If lists are regarded as sets of their elements, **nunion** modifies *list* to become the union of the lists which are supplied as arguments. This is done by adding on, at the end, any elements of the other lists that were not already in *list*. If none of the arguments contains any duplicate elements, neither does the value returned by **nunion**. Elements are compared using **eq**.

It is not safe to assume that *list* has been modified properly in place, as this will not be so if *list* is **nil**. Rather, you must store the value returned by **nunion** in place of *list*.

It is best to use **nunion** with only two arguments so that your code will not be sensitive to the difference between the traditional version of **nunion** and the Common Lisp version **cli:nunion**, below.

nintersection *list* &rest *more-lists*

Like **intersection** but produces the value by deleting elements from *list* until the desired result is reached, and then returning *list* as modified.

It is not safe to assume that *list* has been modified properly in place, as this will not be so if the first element was deleted. Rather, you must store the value returned by **nintersection** in place of *list*.

It is best to use **nintersection** with only two arguments so that your code will not be sensitive to the difference between the traditional version of **nintersection** and the Common Lisp version **cli:nintersection**, below.

cli:union *list1 list2* &key *test test-not key*
cli:intersection *list1 list2* &key *test test-not key*
cli:nunion *list1 list2* &key *test test-not key*
cli:nintersection *list1 list2* &key *test test-not key*

The Common Lisp versions of the above functions, which accept only two sets to operate on, but permit additional arguments to control how elements are matched. These keyword arguments work the same as in **subsetp**.

set-difference *list1 list2* &key *test test-not key*

Returns a list which has all the elements of *list1* which do not match any element of *list2*. The keyword arguments control comparison of elements just as in subsetp.

The result contains no duplicate elements as long as *list1* contains none.

set-exclusive-or *list1 list2* &key *test test-not key*

Returns a list which has all the elements of *list1* which do not match any element of *list2*, and also all the elements of *list2* which do not match any element of *list1*. The keyword arguments control comparison of elements just as in subsetp.

The result contains no duplicate elements as long as neither *list1* nor *list2* contains any.

nset-difference *list1 list2* &key *test test-not key*

Like set-difference but destructively modifies *list1* to produce the value. See the caveat in nintersection, above.

nset-exclusive-or *list1 list2* &key *test test-not key*

Like set-exclusive-or but may destructively modify both *list1* and *list2* to produce the value. See the caveat in nintersection, above.

5.8 Association Lists

In all the alist-searching functions, alist elements which are nil are ignored; they do not count as equivalent to (nil . nil). Elements which are not lists cause errors.

pairlis *cars cdrs* &optional *tail*

pairlis takes two lists and makes an association list which associates elements of the first list with corresponding elements of the second list.

Example:

```
(pairlis '(beef clams kitty) '(roast fried yu-shiang))
   => ((beef . roast) (clams . fried) (kitty . yu-shiang))
```

If *tail* is non-nil, it should be another alist. The new alist continues with *tail* following the newly constructed mappings.

pairlis is defined as:

```
(defun pairlis (cars cdrs &optional tail)
  (nconc (mapcar 'cons cars cdrs) tail))
```

acons *acar acdr tail*

Returns (cons (cons *acar acdr*) *tail*). This adds one additional mapping from *acar* to *acdr* onto the alist *tail*.

assq *item alist*

(assq *item alist*) looks up *item* in the association list (list of conses) *alist*. The value is the first cons whose car is **eq** to *x*, or **nil** if there is none such.

Examples:

```
(assq 'r '((a . b) (c . d) (r . x) (s . y) (r . z)))
        =>  (r . x)

(assq 'fooo '((foo . bar) (zoo . goo))) => nil

(assq 'b '((a b c) (b c d) (x y z))) => (b c d)
```

It is okay to **rplacd** the result of **assq** as long as it is not **nil**, if your intention is to "update" the "table" that was **assq**'s second argument.

Example:

```
(setq values '((x . 100) (y . 200) (z . 50)))
(assq 'y values) => (y . 200)
(rplacd (assq 'y values) 201)
(assq 'y values) => (y . 201)
```

A common trick is to say (cdr (assq x y)). Since the cdr of **nil** is guaranteed to be **nil**, this yields **nil** if no pair is found (or if a pair is found whose cdr is **nil**.)

assq could have been defined by:

```
(defun assq (item list)
    (cond ((null list) nil)
          ((eq item (caar list)) (car list))
          ((assq item (cdr list))) ))
```

assoc *item alist*

assoc is like **assq** except that the comparison uses **equal** instead of **eq**.

Example:

```
(assoc '(a b) '((x . y) ((a b) . 7) ((c . d) .e)))
        => ((a b) . 7)
```

assoc could have been defined by:

```
(defun assoc (item list)
    (cond ((null list) nil)
          ((equal item (caar list)) (car list))
          ((assoc item (cdr list))) ))
```

cli:assoc *item alist* &key *test test-not*

The Common Lisp version of **assoc**, this function returns the first element of *alist* which is a cons whose car matches *item*, or **nil** if there is no such element.

test and *test-not* are used in comparing elements, as in **cli:subst** (page 98), but note that there is no *key* argument in **cli:assoc**.

cli:assoc is incompatible with the traditional function **assoc** in that, like most Common Lisp functions, it uses **eql** by default rather than **equal** for the comparison.

ass *predicate item alist*

ass is the same as assq except that it takes an extra argument which should be a predicate of two arguments, which is used for the comparison instead of eq. (ass 'eq a b) is the same as (assq a b). See also mem, page 104.

This function is part of The mem, rem. del series, whose names were chosed partly because they created a situation in which this function simply had to be called ass. It's too bad that cli:assoc is so general and subsumes ass, which is equivalent to

```
(cli:assoc item alist :test predicate)
```

assoc-if *predicate alist*

Returns the first element of *alist* which is a cons whose car satisfies *predicate*, or nil if there is no such element.

assoc-if-not *predicate alist*

Returns the first element of *alist* which is a cons whose car does not satisfy *predicate*, or nil if there is no such element.

memass *predicate item alist*

memass searches *alist* just like ass, but returns the portion of the list beginning with the pair containing *item*, rather than the pair itself. (car (memass *x y z*)) = (ass *x y z*). See also mem, page 104.

rassq *item alist*
rassoc *item alist*
rass *predicate item alist*
cli:rassoc *item alist* &key *test test-not*
rassoc-if *predicate alist*
rassoc-if-not *predicate alist*

The reverse-association functions are like assq, assoc, etc. but match or test the cdrs of the alist elements instead of the cars. For example, rassq could have been defined by:

```
(defun rassq (item in-list)
    (do l in-list (cdr l) (null l)
      (and (eq item (cdar l))
           (return (car l)))))
```

sassq *item alist fcn*

(sassq *item alist fcn*) is like (assq *item alist*) except that if *item* is not found in *alist*, instead of returning nil, sassq calls the function *fcn* with no arguments. sassq could have been defined by:

```
(defun sassq (item alist fcn)
    (or (assq item alist)
        (apply fcn nil)))
```

sassq and sassoc (see below) are of limited use. These are primarily leftovers from Lisp 1.5.

sassoc *item alist fcn*

(sassoc *item alist fcn*) is like (assoc *item alist*) except that if *item* is not found in *alist*, instead of returning nil, sassoc calls the function *fcn* with no arguments. sassoc could have been defined by:

```
(defun sassoc (item alist fcn)
    (or (assoc item alist)
        (apply fcn nil)))
```

5.9 Stack Lists

When you are creating a list that will not be needed any more once the function that creates it is finished, it is possible to create the list on the stack instead of by consing it. This avoids any permanent storage allocation, as the space is reclaimed as part of exiting the function. By the same token, it is a little risky; if any pointers to the list remain after the function exits, they will become meaningless.

These lists are called *temporary lists* or *stack lists*. You can create them explicitly using the special forms with-stack-list and with-stack-list*. &rest arguments also sometimes create stack lists.

If a stack list, or a list which might be a stack list, is to be returned or made part of permanent list-structure, it must first be copied (see copylist, page 94). The system cannot detect the error of omitting to copy a stack list; you will simply find that you have a value that seems to change behind your back.

with-stack-list (*variable element...*) *body...* *Special form*
with-stack-list* (*variable element... tail*) *body...* *Special form*

These special forms create stack lists that live inside the stack frame of the function that they are used in. You should assume that the stack lists are only valid until the special form is exited.

```
(with-stack-list (foo x y)
  (mumblify foo))
```
is equivalent to
```
(let ((foo (list x y)))
  (mumblify foo))
```
except for the fact that foo's value in the first example is a stack list.

The list created by with-stack-list* looks like the one created by list*. *tail*'s value becomes the ultimate cdr rather than an element of the list.

Here is a practical example. condition-resume (see page 723) might have been defined as follows:

```
(defmacro condition-resume (handler &body body)
  `(with-stack-list* (eh:condition-resume-handlers
                        ,handler eh:condition-resume-handlers)
     . ,body))
```

It is an error to do **rplacd** on a stack list (except for the tail of one made using **with-stack-list***). **rplaca** works normally.

sys:rplacd-wrong-representation-type (error) *Condition*

This is signaled if you **rplacd** a stack list (or a list overlayed with an array or other structure).

5.10 Property Lists

From time immemorial, Lisp has had a kind of tabular data structure called a *property list* (plist for short). A property list contains zero or more entries; each entry associates from a keyword symbol (called the *property name*, or sometimes the *indicator*) to a Lisp object (called the *value* or, sometimes, the *property*). There are no duplications among the property names; a property-list can have only one property at a time with a given name.

This is very similar to an association list. The important difference is that a property list is an object with a unique identity; the operations for adding and removing property-list entries are side-effecting operations which alter the property-list rather than making a new one. An association list with no entries would be the empty list (), i.e. the symbol **nil**. There is only one empty list, so all empty association lists are the same object. Each empty property-list is a separate and distinct object.

The implementation of a property list is a memory cell containing a list with an even number (possibly zero) of elements. Each pair of elements constitutes a *property*; the first of the pair is the name and the second is the value. (It would have been possible to use an alist to hold the pairs; this format was chosen when Lisp was young.) The memory cell is there to give the property list a unique identity and to provide for side-effecting operations.

The term 'property list' is sometimes incorrectly used to refer to the list of entries inside the property list, rather than the property list itself. This is regrettable and confusing.

How do we deal with "memory cells" in Lisp? That is, what kind of Lisp object is a property list? Rather than being a distinct primitive data type, a property list can exist in one of three forms:

1. Any cons can be used as a property list. The cdr of the cons holds the list of entries (property names and values). Using the cons as a property list does not use the car of the cons; you can use that for anything else.

2. The system associates a property list with every symbol (see section 6.3, page 131). A symbol can be used where a property list is expected; the property-list primitives automatically find the symbol's property list and use it.

3. A flavor instance may have a property list. The property list functions operate on instances by sending messages to them, so the flavor can store the property list any way it likes. See page 445.

4. A named structure may have a property list. The property list functions automatically call **named-structure-invoke** when a named structure is supplied as the property list. See page 390.

5. A property list can be a memory cell in the middle of some data structure, such as a list, an array, an instance, or a defstruct. An arbitrary memory cell of this kind is named by a locative (see chapter 14, page 267). Such locatives are typically created with the **locf** special form (see page 38).

Property lists of the first kind are called *disembodied* property lists because they are not associated with a symbol or other data structure. The way to create a disembodied property list is **(ncons nil)**, or **(ncons** *data***)** to store *data* in the car of the property list.

Suppose that, inside a program which deals with blocks, the property list of the symbol **b1** contains this list (which would be the value of **(symbol-plist 'b1)**):

```
(color blue on b6 associated-with (b2 b3 b4))
```

The list has six elements, so there are three properties. The first property's name is the symbol **color**, and its value is the symbol **blue**. One says that "the value of **b1**'s **color** property is **blue**", or, informally, that "**b1**'s **color** property is **blue**." The program is probably representing the information that the block represented by **b1** is painted blue. Similarly, it is probably representing in the rest of the property list that block **b1** is on top of block **b6**, and that **b1** is associated with blocks **b2**, **b3**, and **b4**.

get *plist property-name* &optional *default-value*

get looks up *plist*'s *property-name* property. If it finds such a property, it returns the value; otherwise, it returns *default-value*. If *plist* is a symbol, the symbol's associated property list is used. For example, if the property list of **foo** is **(baz 3)**, then

```
(get 'foo 'baz) => 3
(get 'foo 'zoo) => nil
(get 'foo 'baz t) => 3
(get 'foo 'zoo t) => t
```

getl *plist property-name-list*

getl is like **get**, except that the second argument is a list of property names. **getl** searches down *plist* for any of the names in *property-name-list*, until it finds a property whose name is one of them. If *plist* is a symbol, the symbol's associated property list is used.

getl returns the portion of the list inside *plist* beginning with the first such property that it found. So the car of the returned list is a property name, and the **cadr** is the property value. If none of the property names on *property-name-list* are on the property list, **getl** returns **nil**. For example, if the property list of **foo** were

```
(bar (1 2 3) baz (3 2 1) color blue height six-two)
```

then

```
(getl 'foo '(baz height))
  => (baz (3 2 1) color blue height six-two)
```

When more than one of the names in *property-name-list* is present in *plist*, which one **getl** returns depends on the order of the properties. This is the only thing that depends on that order. The order maintained by **putprop** and **defprop** is not defined (their behavior with respect to order is not guaranteed and may be changed without notice).

putprop *plist x property-name*

This gives *plist* an *property-name*-property of *x*. After this is done, (get *plist property-name*) returns *x*. If *plist* is a symbol, the symbol's associated property list is used.
Example:

```
        (putprop 'nixon t 'crook)
```

It is more modern to write

```
        (setf (get plist property-name) x)
```

which avoids the counterintuitive order in which putprop takes its arguments.

defprop *symbol x property-name* *Special form*

defprop is a form of putprop with unevaluated arguments, which is sometimes more convenient for typing. Normally only a symbol makes sense as the first argument.
Example:

```
        (defprop foo bar next-to)
```

is the same as

```
        (putprop 'foo 'bar 'next-to)
```

remprop *plist property-name*

This removes *plist*'s *property-name* property, by splicing it out of the property list. It returns that portion of the list inside *plist* of which the former *property-name*-property was the car. car of what remprop returns is what get would have returned with the same arguments. If *plist* is a symbol, the symbol's associated property list is used. For example, if the property list of foo was

```
        (color blue height six-three near-to bar)
```

then

```
        (remprop 'foo 'height) => (six-three near-to bar)
```

and foo's property list would be

```
        (color blue near-to bar)
```

If *plist* has no *property-name*-property, then remprop has no side-effect and returns nil.

getf *place property* &optional *default* *Macro*

Equivalent to (get (locf *place*) *property default*), except that getf is defined in Common Lisp, which does not have locf or locatives of any kind.

(setf (getf *place property*) *value*) can be used to store properties into the property list at *place*.

remf *place property* *Macro*

Equivalent to (remprop (locf *place*) *property*), except that remf is defined in Common Lisp.

get-properties *place list-of-properties* *Macro*

Like (getl (locf *place*) *list-of-properties*) but returns slightly different values. Specifically, it searches the property list for a property name which is memq in *list-of-properties*, then returns three values:

propname	the property name found
value	the value of that property
cell	the property list cell found, whose car is *propname* and whose cadr is *value*.

If nothing is found, all three values are nil.

It is possible to continue searching down the property list by using cddr of the third value as the argument to another call to get-properties.

5.11 Hash Tables

A hash table is a Lisp object that works something like a property list. Each hash table has a set of *entries*, each of which associates a particular *key* with a particular *value* (or sequence of values). The basic functions that deal with hash tables can create entries, delete entries, and find the value that is associated with a given key. Finding the value is very fast even if there are many entries, because hashing is used; this is an important advantage of hash tables over property lists. Hashing is explained in section 5.11.4, page 121.

A given hash table stores a fixed number of values for each key; by default, there is only one value. Each time you specify a new value or sequence of values, the old one(s) are lost.

There are three standard kinds of hash tables, which differ in how they compare keys: with eq, with eql or with equal. In other words, there are hash tables which hash on Lisp *objects* (using eq or eql) and there are hash tables which hash on trees (using equal).

You can also create a nonstandard hash table with any comparison function you like, as long as you also provide a suitable hash function. Any two objects which would be regarded as the same by the comparison function should produce the same hash code under the hash function. See the :compare-function and :hash-function keywords under make-hash-table, below.

The following discussion refers to the eq kind of hash table; the other kinds are described later, and work analogously.

eq hash tables are created with the function make-hash-table, which takes various options. New entries are added to hash tables with the puthash function. To look up a key and find the associated value(s), the gethash function is used. To remove an entry, use remhash. Here is a simple example.

```
(setq a (make-hash-table))

(puthash 'color 'brown a)
(puthash 'name 'fred a)

(gethash 'color a) => brown
(gethash 'name a) => fred
```

In this example, the symbols **color** and **name** are being used as keys, and the symbols **brown** and **fred** are being used as the associated values. The hash table remembers one value for each key, since we did not specify otherwise, and has two items in it, one of which associates from **color** to **brown**, and the other of which associates from **name** to **fred**.

Keys do not have to be symbols; they can be any Lisp object. Likewise values can be any Lisp object. Since **eq** does not work reliably on numbers (except for fixnums), they should not be used as keys in an **eq** hash table. Use an **eql** hash table if you want to hash on numeric values.

When a hash table is first created, it has a *size*, which is the number of entries it has room for. But hash tables which are nearly full become slow to search, so if more than a certain fraction of the entries become in use, the hash table is automatically made larger, and the entries are *rehashed* (new hash values are recomputed, and everything is rearranged so that the fast hash lookup still works). This is transparent to the caller; it all happens automatically.

The **describe** function (see page 791) prints a variety of useful information when applied to a hash table.

This hash table facility is similar to the hasharray facility of Interlisp, and some of the function names are the same. However, it is *not* compatible. The exact details and the order of arguments are designed to be consistent with the rest of Zetalisp rather than with Interlisp. For instance, the order of arguments to **maphash** is different, we do not have the Interlisp "system hash table", and we do not have the Interlisp restriction that keys and values may not be **nil**. Note, however, that the order of arguments to **gethash**, **puthash**, and **remhash** is not consistent with the Zetalisp's **get**, **putprop**, and **remprop**, either. This is an unfortunate result of the haphazard historical development of Lisp.

Hash tables are implemented as instances of the flavor **hash-table**. The internals of a hash table are subject to change without notice. Hash tables should be manipulated only with the functions and operations described below.

5.11.1 Hash Table Functions

make-hash-table &rest *options*
make-equal-hash-table &rest *options*

These functions create new hash tables. **make-equal-hash-table** creates an **equal** hash table. **make-hash-table** normally creates an **eq** hash table, but this can be overridden by keywords as described below. Valid option keywords are:

:size Sets the initial size of the hash table, in entries, as a fixnum. The default is 64. The actual size is rounded up from the size you specify to the next size that is good for the hashing algorithm. The number of entries you can actually store in the hash table before it is rehashed is at least the actual size times the rehash threshold (see below).

:test This keyword is the Common Lisp way to specify the kind of hashing desired. The value must be **eq**, **eql** or **equal**. The one specified is used as the compare function and an appropriate hash function is chosen automatically to go with it.

:compare-function
: Specifies a function of two arguments which compares two keys to see if they count as the same for retrieval from this table. For reasonable results, this function should be an equivalence relation. The default is **eq**. For **make-equal-hash-table** the default is **equal**; that is the only difference between that function and **make-hash-table**.

:hash-function
: Specifies a function of one argument which, given a key, computes its hash code. The hash code may be any Lisp object. The purpose of the hash function is to map equivalent keys into identical objects: if two keys would cause the compare function to return non-**nil**, the hash function must produce identical (**eq**) hash codes for them.

  For an **eq** hash table, the key itself is a suitable hash code, so no hash function is needed. Then this option's value should be **nil** (**identity** would also work, but slower). **nil** is the default in **make-hash-table**. **make-equal-hash-table** specifies an appropriate function which uses **sxhash**.

:number-of-values
: A positive integer which specifies how many values to associate with each key. The default is one.

:area
: Specifies the area in which the hash table should be created. This is just like the :area option to **make-array** (see page 167). Defaults to **nil** (i.e. **default-cons-area**).

:rehash-function
: Specifies the function to be used for rehashing when the table becomes full. Defaults to the internal rehashing function that does the usual thing. If you want to write your own rehashing function, you must know all the internals of how hash tables work. These internals are not documented here, as the best way to learn them is to read the source code.

:rehash-size
: Specifies how much to increase the size of the hash table when it becomes full. This can be a fixnum which is the number of entries to add, or it can be a float which is the ratio of the new size to the old size. The default is **1.3**, which causes the table to be made 30% bigger each time it has to grow.

:rehash-threshold
: Sets a maximum fraction of the entries which can be in use before the hash table is made larger and rehashed. The default is **0.7s0**. Alternately, an integer may be specified. It is the exact number of filled entries at which a rehash should be done. When the rehash happens, if the threshold is an integer it is increased in the same proportion as the table has grown.

:rehash-before-cold
: If non-**nil**, this hash table should be rehashed (if that is necessary due to

garbage collection) by **disk-save**. This avoids a delay for rehashing the hash table the first time it is referenced after booting the saved band.

:actual-size
: Specifies exactly the size for the hash table. Hash tables used by the microcode for flavor method lookup must be a power of two in size. This differs from :size in that :size is rounded up to a nearly prime number, but :actual-size is used exactly as specified. :actual-size overrides :size.

hash-table-p *object*

t if *object* is a hash table.

```
(hash-table-p object)
```

is equivalent to

```
(typep object 'hash-table)
```

The following functions are equivalent to sending appropriate messages to the hash table.

gethash *key hash-table* &optional *default-value*

Finds the entry in *hash-table* whose key is *key*, and return the associated value. If there is no such entry, returns *default-value*. Returns also a second value, which is t if an entry was found or nil if there is no entry for *key* in this table.

Returns also a third value, a list which overlays the hash table entry. Its car is the key; the remaining elements are the values in the entry. This is how you can access values other than the first, if the hash table contains more than one value per entry.

puthash *key value hash-table* &rest *extra-values*

Creates an entry associating *key* to *value*; if there is already an entry for *key*, then replace the value of that entry with *value*. Returns *value*. The hash table automatically grows if necessary.

If the hash table associates more than one value with each key, the remaining values in the entry are taken from *extra-values*.

remhash *key hash-table*

Removes any entry for *key* in *hash-table*. Returns t if there was an entry or nil if there was not.

swaphash *key value hash-table* &rest *extra-values*

This specifies new value(s) for *key* like **puthash**, but returns values describing the previous state of the entry, just like **gethash**. It returns the previous (replaced) associated value as the first value, and returns t as the second value if the entry existed previously.

maphash *function hash-table* &rest *extra-args*

For each occupied entry in *hash-table*, call *function*. The arguments passed to *function* are the key of the entry, the value(s) of the entry (however many there are), and the *extra-args* (however many there are).

If the hash table has more than one value per key, all the values, in order, are supplied as successive arguments.

maphash-return *function hash-table*
Like maphash, but accumulates and returns a list of all the values returned by *function* when it is applied to the items in the hash table.

clrhash *hash-table*
Removes all the entries from *hash-table*. Returns the hash table itself.

hash-table-count *hash-table*
Returns the number of filled entries in hash-table.

5.11.2 Hash Table Operations

Hash tables are instances, and support the following operations:

:size *Operation on* hash-table
Returns the number of entries in the hash table. Note that the hash table is rehashed when only a fraction of this many (the rehash threshold) are full.

:filled-entries *Operation on* hash-table
Returns the number of entries that are currently occupied.

:get-hash *key* *Operation on* hash-table
:put-hash *key* &rest *values* *Operation on* hash-table
:swap-hash *key* &rest *values* *Operation on* hash-table
:rem-hash *key* *Operation on* hash-table
:map-hash *function* &rest *extra-args* *Operation on* hash-table
:map-hash-return *function* *Operation on* hash-table
:clear-hash *Operation on* hash-table
:filled-entries *Operation on* hash-table
Are equivalent to the functions gethash, puthash, swaphash, remhash, maphash, maphash-return, clrhash and hash-table-count except that the hash table need not be specified as an argument because it is the object that receives the message. Those functions (documented in the previous section) actually work by invoking these operations.

:modify-hash *key function* &rest *additional-args* *Operation on* hash-table
Passes the value associated with *key* in the table to *function*; whatever *function* returns is stored in the table as the new value for *key*. Thus, the hash association for *key* is both examined and updated according to *function*.

The arguments passed to *function* are *key*, the value associated with *key*, a flag (t if *key* is actually found in the hash table), and the *additional-args* that you specify.

If the hash table stores more than one value per key, only the first value is examined and updated.

5.11.3 Hash Tables and the Garbage Collector

The eq type hash tables actually hash on the address of the representation of the object. equal hash tables do so too, if given keys containing unusual objects (other than symbols, numbers, strings and lists of the above). When the copying garbage collector changes the addresses of objects, it lets the hash facility know so that the next gethash will rehash the table based on the new object addresses.

There may eventually be an option to make-hash-table which tells it to make a "non-GC-protecting" hash table. This is a special kind of hash table with the property that if one of its keys becomes garbage, i.e. is an object not known about by anything other than the hash table, then the entry for that key will be removed silently from the table. When this option exists it will be documented in this section.

5.11.4 Hash Primitive

Hashing is a technique used in algorithms to provide fast retrieval of data in large tables. A function, known as the *hash function*, takes an object that might be used as a key, and produces a number associated with that key. This number, or some function of it, can be used to specify where in a table to look for the datum associated with the key. It is always possible for two different objects to hash to the same value; that is, for the hash function to return the same number for two distinct objects. Good hash functions are designed to minimize this by evenly distributing their results over the range of possible numbers. However, hash table algorithms must still deal with this problem by providing a secondary search, sometimes known as a *rehash*. For more information, consult a textbook on computer algorithms.

sxhash *tree* &optional *ok-to-use-address*

sxhash computes a hash code of a tree, and returns it as a fixnum. A property of sxhash is that (equal *x* *y*) always implies (= (sxhash *x*) (sxhash *y*)). The number returned by sxhash is always a non-negative fixnu. sxhash tries to compute its hash code in such a way that common permutations of an object, such as interchanging two elements of a list or changing one character in a string, always change the hash code.

Here is an example of how to use sxhash in maintaining hash tables of trees:

```
(defun knownp (x &aux i bkt)       ;look up x in the table
    (setq i (abs (remainder (sxhash x) 176)))
      ;The remainder should be reasonably randomized.
    (setq bkt (aref table i))
      ;bkt is thus a list of all those expressions that
      ;hash into the same number as does x.
    (memq x bkt))
```

For an "intern" for trees, one could write:

```
(defun sintern (x &aux bkt i tem)
    (setq i (abs (remainder (sxhash x) 2n-1)))
        ;2n-1 stands for a power of 2 minus one.
        ;This is a good choice to randomize the
        ;result of the remainder operation.
    (setq bkt (aref table i))
    (cond ((setq tem (memq x bkt))
           (car tem))
          (t (aset (cons x bkt) table i)
             x)))
```

If sxhash is given a named structure or a flavor instance, or if such an object is part of a tree that is sxhash'ed, it asks the object to supply its own hash code by performing the :sxhash operation if the object supports it. This should return a suitable nonnegative hash code. The easiest way to compute one is usually by applying sxhash to one or more of the components of the structure or the instance variables of the instance.

For named structures and flavor instances that do not handle the :sxhash operation, and other unusual kinds of objects, sxhash can optionally use the object's address as its hash code, if you specify a non-nil second argument. If you use this option, you must be prepared to deal with hash codes changing due to garbage collection.

sxhash provides what is called "hashing on equal"; that is, two objects that are equal are considered to be "the same" by sxhash. If two strings differ only in alphabetic case, sxhash returns the same thing for both of them, making it suitable for equalp hashing as well in some cases.

Therefore, sxhash is useful for retrieving data when two keys that are not the same object but are equal are considered the same. If you consider two such keys to be different, then you need "hashing on eq", where two different objects are always considered different. In some Lisp implementations, there is an easy way to create a hash function that hashes on eq, namely, by returning the virtual address of the storage associated with the object. But in other implementations, of which Zetalisp is one, this doesn't work, because the address associated with an object can be changed by the relocating garbage collector. The hash tables created by make-hash-table deal with this problem by using the appropriate subprimitives so that they interface correctly with the garbage collector. If you need a hash table that hashes on eq, it is already provided; if you need an eq hash function for some other reason, you must build it yourself, either using the provided eq hash table facility or carefully using subprimitives.

5.12 Resources

Storage allocation is handled differently by different computer systems. In many languages, the programmer must spend a lot of time thinking about when variables and storage units are allocated and deallocated. In Lisp, freeing of allocated storage is normally done automatically by the Lisp system; when an object is no longer accessible to the Lisp environment, the garbage collector reuses its storage for some other object. This relieves the programmer of a great burden, and makes writing programs much easier.

However, automatic freeing of storage incurs an expense: more computer resources must be devoted to the garbage collector. If a program is designed to allocate temporary storage, which is then left as garbage, more of the computer must be devoted to the collection of garbage; this expense can be high. In some cases, the programmer may decide that it is worth putting up with the inconvenience of having to free storage under program control, rather than letting the system do it automatically, in order to prevent a great deal of overhead from the garbage collector.

It usually is not worth worrying about freeing of storage when the units of storage are very small things such as conses or small arrays. Numbers are not a problem, either; fixnums and short floats do not occupy storage, and the system has a special way of garbage-collecting the other kinds of numbers with low overhead. But when a program allocates and then gives up very large objects at a high rate (or large objects at a very high rate), it can be worthwhile to keep track of that one kind of object manually. Within the Lisp Machine system, there are several programs that are in this position. The Chaosnet software allocates and frees "packets", which are moderately large, at a very high rate. The window system allocates and frees certain kinds of windows, which are very large, moderately often. Both of these programs manage their objects manually, keeping track of when they are no longer used.

When we say that a program "manually frees" storage, it does not really mean that the storage is freed in the same sense that the garbage collector frees storage. Instead, a list of unused objects is kept. When a new object is desired, the program first looks on the list to see if there is one around already, and if there is it uses it. Only if the list is empty does it actually allocate a new one. When the program is finished with the object, it returns it to this list.

The functions and special forms in this section perform the above function. The set of objects forming each such list is called a *resource*; for example, there might be a Chaosnet packet resource. defresource defines a new resource; allocate-resource allocates one of the objects; deallocate-resource frees one of the objects (putting it back on the list); and using-resource temporarily allocates an object and then frees it.

5.12.1 Defining Resources

defresource *Macro*

The defresource special form is used to define a new resource. The form looks like this:

```
(defresource name parameters
    doc-string
    keyword value
    keyword value
    ...)
```

name should be a symbol; it is the name of the resource and gets a defresource property of the internal data structure representing the resource.

parameters is a lambda-list giving names and default values (if &optional is used) of parameters to an object of this type. For example, if one had a resource of two-dimensional arrays to be used as temporary storage in a calculation, the resource would typically have two parameters, the number of rows and the number of columns. In the simplest case *parameters* is ().

The documentation string is recorded for (documentation *name* 'resource) to access. It may be omitted.

The keyword options control how the objects of the resource are made and kept track of. The following keywords are allowed:

:constructor — The *value* is either a form or the name of a function. It is responsible for making an object, and will be used when someone tries to allocate an object from the resource and no suitable free objects exist. If the *value* is a form, it may access the parameters as variables. If it is a function, it is given the internal data structure for the resource and any supplied parameters as its arguments; it will need to default any unsupplied optional parameters. This keyword is required.

:free-list-size — The *value* is the number of objects which the resource data structure should have room, initially, to remember. This is not a hard limit, since the data structure will be made bigger if necessary.

:initial-copies — The *value* is a number (or nil which means 0). This many objects will be made as part of the evaluation of the defresource; thus is useful to set up a pool of free objects during loading of a program. The default is to make no initial copies.

If initial copies are made and there are *parameters*, all the parameters must be &optional and the initial copies will have the default values of the parameters.

:initializer — The *value* is a form or a function as with :constructor. In addition to the parameters, a form here may access the variable object (in the current package). A function gets the object as its second argument, after the data structure and before the parameters. The purpose of the initializer function or form is to clean up the contents of the object before each use.

It is called or evaluated each time an object is allocated, whether just constructed or being reused.

:finder
The *value* is a form or a function as with :constructor and sees the same arguments. If this option is specified, the resource system does not keep track of the objects. Instead, the finder must do so. It will be called inside a without-interrupts and must find a usable object somehow and return it.

:matcher
The *value* is a form or a function as with :constructor. In addition to the parameters, a form here may access the variable object (in the current package). A function gets the object as its second argument, after the data structure and before the parameters. The job of the matcher is to make sure that the object matches the specified parameters. If no matcher is supplied, the system will remember the values of the parameters (including optional ones that defaulted) that were used to construct the object, and will assume that it matches those particular values for all time. The comparison is done with equal (not eq). The matcher is called inside a without-interrupts.

:checker
The job of the checker is to determine whether the object is safe to allocate. The *value* is a form or a function, as above. In addition to the parameters, a form here may access the variables object and in-use-p (in the current package). A function receives these as its second and third arguments, after the data structure and before the parameters. If no checker is supplied, the default checker looks only at in-use-p; if the object has been allocated and not freed it is not safe to allocate, otherwise it is. The checker is called inside a without-interrupts.

If these options are used with forms (rather than functions), the forms get compiled into functions as part of the expansion of defresource. The functions, whether user-provided or generated from forms, are given names like (:property *resource-name* si:resource-constructor); these names are not guaranteed not to change in the future.

Most of the options are not used in typical cases. Here is an example:

```
(defresource two-dimensional-array (rows columns)
      :constructor (make-array (list rows columns)))
```

Suppose the array was usually going to be 100 by 100, and you wanted to preallocate one during loading of the program so that the first time you needed an array you wouldn't have to spend the time to create one. You might simply put

```
(using-resource (foo two-dimensional-array 100 100)
      )
```

after your defresource, which would allocate a 100 by 100 array and then immediately free it. Alternatively you could write:

```
(defresource two-dimensional-array
                       (&optional (rows 100) (columns 100))
      :constructor (make-array (list rows columns))
      :initial-copies 1)
```

Here is an example of how you might use the :matcher option. Suppose you wanted to have a resource of two-dimensional arrays, as above, except that when you allocate one you don't care about the exact size, as long as it is big enough. Furthermore you realize that you are going to have a lot of different sizes and if you always allocated one of exactly the right size, you would allocate a lot of different arrays and would not reuse a pre-existing array very often. So you might write:

```
(defresource sloppy-two-dimensional-array (rows columns)
    :constructor (make-array (list rows columns))
    :matcher (and (≥ (array-dimension-n 1 object) rows)
                  (≥ (array-dimension-n 2 object) columns)))
```

5.12.2 Allocating Resource Objects

allocate-resource *resource-name* &rest *parameters*

Allocates an object from the resource specified by *resource-name*. The various forms and/or functions given as options to defresource, together with any *parameters* given to allocate-resource, control how a suitable object is found and whether a new one has to be constructed or an old one can be reused.

Note that the using-resource special form is usually what you want to use, rather than allocate-resource itself; see below.

deallocate-resource *resource-name resource-object*

Frees the object *resource-object*, returning it to the free-object list of the resource specified by *resource-name*.

using-resource (*variable resource parameters...*) *body...* *Macro*

The *body* forms are evaluated sequentially with *variable* bound to an object allocated from the resource named *resource*, using the given *parameters*. The *parameters* (if any) are evaluated, but *resource* is not.

using-resource is often more convenient than calling allocate-resource and deallocate-resource. Furthermore it is careful to free the object when the body is exited, whether it returns normally or via throw. This is done by using unwind-protect; see page 82.

Here is an example of the use of resources:

```
(defresource huge-16b-array (&optional (size 1000))
  :constructor (make-array size :type 'art-16b))

(defun do-complex-computation (x y)
  (using-resource (temp-array huge-16b-array)
    ...                                ;Within the body, the array can be used.
    (aset 5 temp-array i)
    ...))                              ;The array is deallocated at the end.
```

deallocate-whole-resource *resource-name*
Frees all objects in *resource-name*. This is like doing deallocate-resource on each one individually. This function is often useful in warm-boot initializations.

map-resource *function resource-name* &rest *extra-args*
Calls function on each object created in *resource-name*. Each time function is called, it receives three fixed arguments, plus whatever *extra-args* were specified. The three fixed arguments are an object of the resource; t if the object is currently allocated ("in use"); and the resource data structure itself.

clear-resource *resource-name*
Forgets all of the objects being remembered by the resource specified by *resource-name*. Future calls to allocate-resource will create new objects. This function is useful if something about the resource has been changed incompatibly, such that the old objects are no longer usable. If an object of the resource is in use when clear-resource is called, an error will be signaled when that object is deallocated.

5.12.3 Accessing the Resource Data Structure

The constructor, initializer, matcher and checker functions receive the internal resource data structure as an argument. This is a named structure array whose elements record the objects both free and allocated, and whose array leader contains sundry other information. This structure should be accessed using the following primitives:

si:resource-object *resource-structure index*
Returns the *index*'th object remembered by the resource. Both free and allocated objects are remembered.

si:resource-in-use-p *resource-structure index*
Returns t if the *index*'th object remembered by the resource has been allocated and not deallocated. Simply defined resources will not reallocate an object in this state.

si:resource-parameters *resource-structure index*
Returns the list of parameters from which the *index*'th object was originally created.

si:resource-n-objects *resource-structure*
Returns the number of objects currently remembered by the resource. This will include all objects ever constructed, unless clear-resource has been used.

si:resource-parametizer *resource-structure*
Returns a function, created by defresource, which accepts the supplied parameters as arguments, and returns a complete list of parameter values, including defaults for the optional ones.

5.12.4 Fast Pseudo-Resources

When small temporary data structures are allocated so often that they amount to a considerable drain of storage space, an ordinary resource may be unacceptably slow. Here is a simple technique that provides in such cases nearly all the benefit of a resource while costing nearly nothing. The function read uses it to allocate a buffer for reading tokens of input.

```
(defvar buffer-for-reuse nil)

(defsubst get-buffer ()
  (or (do (old)
          ((%store-conditional (locf buffer-for-reuse)
                               (setq old buffer-for-reuse)
                               nil)
           old))
      (construct-new-buffer))))

(defsubst free-buffer (buffer)
  (setq buffer-for-reuse buffer))
```

To allocate a buffer for use, do (get-buffer). To free it when you are done with it, call free-buffer. It is assumed that construct-new-buffer is the function which can create a new buffer when there is none available for reuse.

This technique keeps track of at most one buffer which has been freed and may be reused. It is not effective in this simple form when more than one buffer is needed at any given time by one application. In the case of read, only one token is being read in at any time.

It is safe for more than one process to call read because get-buffer is designed to guarantee that a request cannot get a buffer already handed out and not freed. Likewise, nothing terrible happens if there is an error inside read and read is called recursively within the debugger. The only problem is that multiple buffers will be allocated, which means that some of them will be lost. But the cost of this is minor in the cases where this technique is applicable. For example, if two processes are reading files, process switching will probably happen a few times a second, each time costing one buffer not reused. This is insignificant compared to the storage used up for other purposes by reading large amounts of data.

6. Symbols

This chapter discusses the symbol as a Lisp data type. The Lisp system uses symbols as variables and function names, but these applications of symbols are discussed in chapter 3.

6.1 The Value Cell

Each symbol has associated with it a *value cell*, which refers to one Lisp object. This object is called the symbol's *value*, since it is what you get when you evaluate the symbol as a dynamic variable in a program. Variables and how they work are described in section 3.1, page 25. We also say the the symbol *is bound to* the object which is its value. The symbols nil and t are always bound to themselves; they may not be assigned, bound, or otherwise used as variables. The same is true of symbols in the keyword package.

The value cell can also be *void*, referring to *no* Lisp object, in which case the symbol is said to be void or *unbound*. This is the initial state of a symbol when it is created. An attempt to evaluate a void symbol causes an error.

Lexical variable bindings are not stored in symbol value cells. The functions in this section have no interaction with lexical bindings.

symeval *symbol*
symbol-value *symbol*

symeval is the basic primitive for retrieving a symbol's value. (symeval *symbol*) returns *symbol*'s current binding. This is the function called by eval when it is given a symbol to evaluate. If the symbol is unbound, then symeval signals an error. symbol-value is the Common Lisp name for this function.

set *symbol value*

set is the primitive for assignment of symbols. The *symbol*'s value is changed to *value*; *value* may be any Lisp object. set returns *value*.
Example:

```
(set (cond ((eq a b) 'c)
           (t 'd))
     'foo)
```

either sets c to foo or sets d to foo.

(setf (symeval *symbol*) *value*) is a more modern way to do this.

boundp *symbol*

t if *symbol*'s value cell is not void.

makunbound *symbol*

Makes *symbol*'s value cell void.

Example:
```
        (setq a 1)
        a => 1
        (makunbound 'a)
        a => causes an error.
```
makunbound returns its argument.

value-cell-location *symbol*

Returns a locative pointer to *symbol*'s value cell. See the section on locatives (chapter 14, page 267). It is preferable to write
```
        (locf (symeval symbol))
```
which is equivalent, instead of calling this function explicitly.

This is actually the internal value cell; there can also be an external value cell. For details, see the section on closures (chapter 12, page 250).

For historical compatibility, value-cell-location of a quoted symbol is recognized specially by the compiler and treated like variable-location (page 30). However, such usage results in a compiler warning, and eventually this compatibility feature will be removed.

6.2 The Function Cell

Every symbol also has associated with it a *function cell*. The *function* cell is similar to the *value* cell; it refers to a Lisp object. When a function is referred to by name, that is, when a symbol is passed to apply or appears as the car of a form to be evaluated, that symbol's function cell is used to find its *definition*, the functional object which is to be applied. For example, when evaluating (+ 5 6), the evaluator looks in +'s function cell to find the definition of +, in this case a compiled function object, to apply to 5 and 6.

Maclisp does not have function cells; instead, it looks for special properties on the property list. This is one of the major incompatibilities between the two dialects.

Like the value cell, a function cell can be void, and it can be bound or assigned. (However, to bind a function cell you must use the %bind subprimitive; see page 284.) The following functions are analogous to the value-cell-related functions in the previous section.

fsymeval *symbol*
symbol-function *symbol*

Returns *symbol*'s definition, the contents of its function cell. If the function cell is void, fsymeval signals an error. symbol-function is the Common Lisp name for this function.

fset *symbol definition*

Stores *definition*, which may be any Lisp object, into *symbol*'s function cell. It returns *definition*.

(setf (fsymeval *symbol*) *definition*) is a more modern way to do this.

fboundp *symbol*

nil if *symbol*'s function cell is void, i.e. if *symbol* is undefined. Otherwise it returns **t**.

fmakunbound *symbol*

Causes *symbol* to be undefined, i.e. its function cell to be void. It returns *symbol*.

function-cell-location *symbol*

Returns a locative pointer to *symbol*'s function cell. See the section on locatives (chapter 14, page 267). It is preferable to write

```
(locf (fsymeval symbol))
```

rather than calling this function explicitly.

Since functions are the basic building block of Lisp programs, the system provides a variety of facilities for dealing with functions. Refer to chapter 11 for details.

6.3 The Property List

Every symbol has an associated property list. See section 5.10, page 113 for documentation of property lists. When a symbol is created, its property list is initially empty.

The Lisp language itself does not use a symbol's property list for anything. (This was not true in older Lisp implementations, where the print-name, value-cell, and function-cell of a symbol were kept on its property list.) However, various system programs use the property list to associate information with the symbol. For instance, the editor uses the property list of a symbol which is the name of a function to remember where it has the source code for that function, and the compiler uses the property list of a symbol which is the name of a special form to remember how to compile that special form.

Because of the existence of print-name, value, function, and package cells, none of the Maclisp system property names (**expr**, **fexpr**, **macro**, **array**, **subr**, **lsubr**, **fsubr**, and in former times **value** and **pname**) exist in Zetalisp.

plist *symbol*
symbol-plist

Returns the list which represents the property list of *symbol*. Note that this is not actually a property list; you cannot do **get** on it. This value is like what would be the cdr of a property list.

symbol-plist is the Common Lisp name.

setplist *symbol list*

Sets the list which represents the property list of *symbol* to *list*. **setplist** is to be used with caution (or not at all), since property lists sometimes contain internal system properties, which are used by many useful system functions. Also it is inadvisable to have the property lists of two different symbols be **eq**, since the shared list structure will cause unexpected effects on one symbol if **putprop** or **remprop** is done to the other.

setplist is equivalent to

```
        (setf (plist symbol) list)
```

property-cell-location *symbol*

Returns a locative pointer to the location of *symbol*'s property-list cell. This locative pointer may be passed to **get** or **putprop** with the same results as if as *symbol* itself had been passed. It is preferable to write

```
        (locf (plist symbol))
```

rather than using this function.

6.4 The Print Name

Every symbol has an associated string called the *print-name*, or *pname* for short. This string is used as the external representation of the symbol: if the string is typed in to **read**, it is read as a reference to that symbol (if it is interned), and if the symbol is printed, **print** types out the print-name.

If a symbol is uninterned, **#:** is normally printed as a prefix before the symbol's print-name. If the symbol is interned, a package prefix may be printed, depending on the current package and how it relates to the symbol's home package.

For more information, see the sections on the *reader* (see section 23.3, page 516), *printer* (see section 23.1, page 506), and packages (see chapter 27, page 636).

symbol-name *symbol*
get-pname *symbol*

Returns the print-name of the symbol *symbol*.
Example:

```
        (symbol-name 'xyz) => "XYZ"
```

get-pname is an older name for this function.

6.5 The Package Cell

Every symbol has a *package cell* which, for interned symbols, is used to point to the package which the symbol belongs to. For an uninterned symbol, the package cell contains **nil**. For information about packages in general, see the chapter on packages, chapter 27, page 636. For information about package cells, see page 639.

6.6 Creating Symbols

The functions in this section are primitives for creating symbols. However, before discussing them, it is important to point out that most symbols are created by a higher-level mechanism, namely the reader and the intern function. Nearly all symbols in Lisp are created by virtue of the reader's having seen a sequence of input characters that looked like the printed representation (p.r.) of a symbol. When the reader sees such a p.r., it calls intern (see page 645), which looks up the sequence of characters in a big table and sees whether any symbol with this print-name already exists. If it does, read uses the already-existing symbol. If it does not, then intern creates a new symbol and puts it into the table; read uses that new symbol.

A symbol that has been put into such a table is called an *interned* symbol. Interned symbols are normally created automatically; the first time that someone (such as the reader) asks for a symbol with a given print-name, that symbol is automatically created.

These tables are called *packages*. For more information, turn to the chapter on packages (chapter 27, page 636).

An *uninterned* symbol is a symbol that has not been recorded or looked up in a package. It is used simply as a data object, with no special cataloging. An uninterned symbol prints with a prefix #: when escaping is in use, unless *print-gensym* is nil. This allows uninterned symbols to be distinguishable and to read back in as uninterned symbols. See page 515.

The following functions can be used to create uninterned symbols explicitly.

make-symbol *pname* &optional *permanent-p*

Creates a new uninterned symbol, whose print-name is the string *pname*. The value and function cells are void and the property list is empty. If *permanent-p* is specified, it is assumed that the symbol is going to be interned and probably kept around forever; in this case it and its pname are put in the proper areas. If *permanent-p* is nil (the default), the symbol goes in the default area and the pname is not copied. *permanent-p* is mostly for the use of intern itself.

Examples:

```
(setq a (make-symbol "foo")) => foo
(symeval a) => ERROR!
```

Note that the symbol is *not* interned; it is simply created and returned.

copysymbol *symbol copy-props*
copy-symbol *symbol copy-props*

Returns a new uninterned symbol with the same print-name as *symbol*. If *copy-props* is non-nil, then the value and function-definition of the new symbol are the same as those of *symbol*, and the property list of the new symbol is a copy of *symbol*'s. If *copy-props* is nil, then the new symbol's function and value are void, and its property list is empty.

gensym &optional *x*

Invents a print-name and creates a new symbol with that print-name. It returns the new, uninterned symbol.

The invented print-name is a prefix (the value of si:*gensym-prefix) followed by the decimal representation of a number (the value of si:*gensym-counter), e.g. g0001. The number is increased by one every time gensym is called.

If the argument *x* is present and is a fixnum, then si:*gensym-counter is set to *x*. If *x* is a string or a symbol, then si:*gensym-prefix is set to it, so it becomes the prefix for this and successive calls to gensym. After handling the argument, gensym creates a symbol as it would with no argument.
Examples:

```
if      (gensym) => #:g0007
then    (gensym 'foo) => #:foo0008
        (gensym 32.) => #:foo0032
        (gensym) => #:foo0033
```

Note that the number is in decimal and always has four digits. #: is the prefix normally printed before uninterned symbols.

gensym is usually used to create a symbol which should not normally be seen by the user, and whose print-name is unimportant, except to allow easy distinction by eye between two such symbols. The optional argument is rarely supplied. The name comes from 'generate symbol', and the symbols produced by it are often called "gensyms".

gentemp &optional (*prefix* "t") (*a-package* package)
Creates and returns a new symbol whose name starts with *prefix*, interned in *a-package*, and distinct from any symbol already present there. This is done by trying names one by one until a name not already in use is found, which may be very slow.

7. Numbers

Zetalisp includes several types of numbers, with different characteristics. Most numeric functions accept any type of numbers as arguments and do the right thing. That is to say, they are *generic*. In Maclisp, there are generic numeric functions (like **plus**) and there are specific numeric functions (like **+**) which only operate on a certain type of number, but are much more efficient. In Zetalisp, this distinction does not exist; both function names exist for compatibility but they are identical. The microprogrammed structure of the machine makes it possible to have only the generic functions without loss of efficiency.

The types of numbers in Zetalisp are:

fixnum	Fixnums are 25-bit twos-complement binary integers. These are the preferred, most efficient type of number.
bignum	Bignums are arbitrary-precision binary integers.
ratio	Ratios represent rational numbers exactly as the quotient of two integers, each of which can be a fixnum or a bignum. Ratios with a denominator of one are not normally created, as an integer is returned instead.
single-float or full-size float	Full size floats are floating-point numbers. They have a mantissa of 31 bits and an exponent of 11 bits, providing a precision of about 9 digits and a range of about 10↑300. Stable rounding is employed.
short-float	Short floats are another form of floating-point number, with a mantissa of 17 bits and an exponent of 8 bits, providing a precision of about 5 digits and a range of about 10↑38. Stable rounding is employed. Short floats are useful because, like fixnums, and unlike full-size floats, they don't require any storage. Computing with short floats is more efficient than with full-size floats because the operations are faster and consing overhead is eliminated.
complexnum	Complexnums represent complex numbers with a real part and an imaginary part, which can be any type of number except complexnums. (They must be both rational or both floats of the same type). It is impossible to make a complexnum whose real part is rational and whose imaginary part is zero; it is always changed into a real number. However, it *is* possible to create complexnums with an imaginary part of 0.0, and such numbers may result from calculations involving complexnums. In fact, 5.0 and 5.0+0.0i are *always* distinct; they are not **eql**, and arithmetic operations will never canonicalize a complexnum with zero imaginary part into a real number.

Generally, Lisp objects have a unique identity; each exists, independent of any other, and you can use the **eq** predicate to determine whether two references are to the same object or not. Numbers are the exception to this rule; they don't work this way. The following function may return either **t** or **nil**. Its behavior is considered undefined; as this manual is written, it returns **t** when interpreted but **nil** when compiled.

```
(defun foo ()
  (let ((x (float 5)))
    (eq x (car (cons x nil)))))
```

This is very strange from the point of view of Lisp's usual object semantics, but the implementation works this way, in order to gain efficiency, and on the grounds that identity testing of numbers is not really an interesting thing to do. So the rule is that the result of applying **eq** to numbers is undefined, and may return either **t** or **nil** on what appear to be two pointers to the same numeric object. The only reasonable ways to compare numbers are **=** (see page 139) and **eql** (page 69), and other things (**equal** or **equalp**) based on them.

Conversely, fixnums and short floats have the unusual property that they are always **eq** if they are equal in value. This is because they do not point to storage; the "pointer" field of a fixnum is actually its numeric value, and likewise for short floats. Stylisticly it is better to avoid depending on this, by using **eql** rather than **eq**. Also, comparing floats of any sort for exact equality, even with **=** which is guaranteed to consider only the numeric values, is usually unwise since round-off error can make the answer unpredictable and meaningless.

The distinction between fixnums and bignums is largely transparent to the user. The user simply computes with integers, and the system represents some as fixnums and the rest (less efficiently) as bignums. The system automatically converts back and forth between fixnums and bignums based solely on the size of the integer. There are a few low level functions which only work on fixnums; this fact is noted in their documentation. Also, when using **eq** on numbers the user needs to be aware of the fixnum/bignum distinction.

Integer computations cannot overflow, except for division by zero, since bignums can be of arbitrary size. Floating-point computations can get exponent overflow or underflow, if the result is too large or small to be represented. Exponent overflow always signals an error. Exponent underflow normally signals an error, and assumes **0.0** as the answer if the user says to proceed from the error. However, if the value of the variable **zunderflow** is non-**nil**, the error is skipped and computation proceeds with **0.0** in place of the result that was too small.

When an arithmetic function of more than one argument is given arguments of different numeric types, uniform *coercion rules* are followed to convert the arguments to a common type, which is also the type of the result (for functions which return a number). When an integer meets a ratio, the result is a ratio. When an integer or ratio meets a float, the result is a float of the same sort. When a short-float meets a full-size float, the result is a full-size float.

If any argument of the arithmetic function is complex, the other arguments are converted to complex. The components of a complex result must be both full-size floats, both small-floats, or both rational; if they differ, the one whose type comes last in that list is converted to match the other. Finally, if the components of the result are rational and the imaginary part is zero, the result is simply the real part. If, however, the components are floats, the value is always complex even if the imaginary part is zero.

Thus if the constants in a numerical algorithm are written as short floats (assuming this provides adequate precision), and if the input is a short float, the computation is done with short floats and the result is a short float, while if the input is a full-size float the computation is done in full precision and the result is a full-size float.

Zetalisp never automatically converts between full-size floats and short floats in the same way as it automatically converts between fixnums and bignums since this would lead either to inefficiency or to unexpected numerical inaccuracies. (When a short float meets a full-size float, the result is a full-size float, but if you use only one type, all the results are of the same type too.) This means that a short float computation can get an exponent overflow error even when the result could have been represented as a full-size float.

Floating-point numbers retain only a certain number of bits of precision; therefore, the results of computations are only approximate. Full-size floats have 31 bits and short floats have 17 bits, not counting the sign. The method of approximation is "stable rounding". The result of an arithmetic operation is the float which is closest to the exact value. If the exact result falls precisely halfway between two representable floats, the result is rounded down if the least-significant bit is 0, or up if the least-significant bit is 1. This choice is arbitrary but insures that no systematic bias is introduced.

Unlike Maclisp, Zetalisp does not have number declarations in the compiler. Note that because fixnums and short floats require no associated storage they are as efficient as declared numbers in Maclisp. Bignums and full-size floats are less efficient; however, bignum and float intermediate results are garbage-collected in a special way that avoids the overhead of the full garbage collector.

The different types of numbers can be distinguished by their printed representations. If a number has an exponent separated by 's', it is a short float. If a number has an exponent separated by 'f', it is a full-size float. A leading or embedded (but *not* trailing) decimal point, and/or an exponent separated by 'e', indicates a float; which kind is controlled by the variable *read-default-float-format*, which is usually set to specify full-size floats. Short floats require a special indicator so that naive users will not accidentally compute with the lesser precision. Fixnums and bignums have similar printed representations since there is no numerical value that has a choice of whether to be a fixnum or a bignum; an integer is a bignum if and only if its magnitude is too big for a fixnum. See the examples on page 518, in the description of what the reader understands.

zunderflow *Variable*

When this is nil, floating point exponent underflow is an error. When this is t, exponent underflow proceeds, returning zero as the value. The same thing could be accomplished with a condition handler. However, zunderflow is useful for Maclisp compatibility, and is also faster.

sys:floating-exponent-overflow (sys:arithmetic-error error) *Condition*
sys:floating-exponent-underflow (sys:arithmetic-error error) *Condition*

sys:floating-exponent-overflow is signaled when the result of an arithmetic operation should be a floating point number, but the exponent is too large to be represented in the format to be used for the value. sys:floating-exponent-underflow is signaled when the exponent is too small.

The condition instance provides two additional operations: :function, which returns the arithmetic function that was called, and :small-float-p, which is t if the result was supposed to be a short float.

sys:floating-exponent-overflow provides the **:new-value** proceed type. It expects one argument, a new value.

sys:floating-exponent-underflow provides the **:use-zero** proceed type, which expects no argument.

Unfortunately, it is not possible to make the arguments to the operation available. Perhaps someday they will be.

7.1 Numeric Predicates

zerop *x*

Returns **t** if *x* is zero. Otherwise it returns **nil**. If *x* is not a number, **zerop** causes an error. For floats, this only returns **t** for exactly **0.0** or **0.0s0**. For complex numbers, it returns **t** if both real and imaginary parts are zero.

plusp *x*

Returns **t** if its argument is a positive number, strictly greater than zero. Otherwise it returns **nil**. If *x* is not a number, **plusp** causes an error.

minusp *x*

Returns **t** if its argument is a negative number, strictly less than zero. Otherwise it returns **nil**. If *x* is not a number, **minusp** causes an error.

oddp *number*

Returns **t** if *number* is odd, otherwise **nil**. If *number* is not a fixnum or a bignum, **oddp** causes an error.

evenp *number*

Returns **t** if *number* is even, otherwise **nil**. If *number* is not a fixnum or a bignum, **evenp** causes an error.

signp *test x* *Special form*

Tests the sign of a number. **signp** is present only for Maclisp compatibility and is not recommended for use in new programs. **signp** returns **t** if *x* is a number which satisfies the *test*, **nil** if it is not a number or does not meet the test. *test* is not evaluated, but *x* is. *test* can be one of the following:

l	x < 0
le	x ≤ 0
e	x = 0
n	x ≠ 0
ge	x ≥ 0
g	x > 0

Examples:
```
        (signp ge 12) => t
        (signp le 12) => nil
        (signp n 0) => nil
        (signp g 'foo) => nil
```

See also the data-type predicates integerp, rationalp, realp, complexp, floatp, bigp, small-floatp, and numberp (page 12).

7.2 Numeric Comparisons

All of these functions require that their arguments be numbers; they signal an error if given a non-number. Equality tests work on all types of numbers, automatically performing any required coercions (as opposed to Maclisp in which generally only the spelled-out names work for all kinds of numbers). Ordering comparisons allow only real numbers, since they are meaningless on complex numbers.

= &rest *numbers*

Returns t if all the arguments are numerically equal. They need not be of the same type; 1 and 1.0 are considered equal. Character objects are also allowed, and in effect coerced to integers for comparison.

See also eql, page 69, which insists that both the type and the value match when its arguments are numbers.

> &rest *numbers*
greaterp &rest *numbers*

> compares each pair of successive arguments. If any argument is not greater than the next, > returns nil. But if the arguments are monotonically strictly decreasing, the result is t. Zero arguments are always monotonically decreasing, and so is a single argument.
Examples:
```
        (>) => t
        (> 3) => t
        (> 4 3) => t
        (> 4 3 2 1 0) => t
        (> 4 3 1 2 0) => nil
```

greaterp is the Maclisp name for this function.

>= &rest *numbers*
≥ &rest *numbers*

≥ compares each pair of successive arguments. If any argument is less than the next, ≥ returns nil. But if the arguments are monotonically decreasing or equal, the result is t.

>= is the Common Lisp name for this function.

< &rest *numbers*
lessp &rest *numbers*

< compares each pair of successive arguments. If any argument is not less than the next, < returns nil. But if the arguments are monotonically strictly increasing, the result is t.
Examples:

```
(<) => t
(< 3) => t
(< 3 4) => t
(< 1 1) => nil
(< 0 1 2 3 4) => t
(< 0 1 3 2 4) => nil
```

lessp is the Maclisp name for this function.

<= &rest *numbers*
≤ &rest *numbers*

≤ compares its arguments from left to right. If any argument is greater than the next, ≤ returns nil. But if the arguments are monotonically increasing or equal, the result is t.

<= is the Common Lisp name for this function.

≠ &rest *numbers*
//= &rest *numbers*

t if no two arguments are numerically equal. This is the same as (not (= ...)) when there are two arguments, but not when there are more than two.

With zero or one argument, the value is always t, since there is no pair of arguments that fail to be equal.

//= is the Common Lisp name for this function. In Common Lisp syntax, it would be written /=.

max &rest *one-or-more-args*

Returns the largest of its arguments, which must not be complex.
Example:

```
(max 1 3 2) => 3
```

max requires at least one argument.

min &rest *one-or-more-args*

Returns the smallest of its arguments, which must not be complex.
Example:

```
(min 1 3 2) => 1
```

min requires at least one argument.

7.3 Arithmetic

All of these functions require that their arguments be numbers, and signal an error if given a non-number. They work on all types of numbers, automatically performing any required coercions (as opposed to Maclisp, in which generally only the spelled-out versions work for all kinds of numbers, and the '$' versions are needed for floats).

+ &rest *args*
plus &rest *args*
+$ &rest *args*

Returns the sum of its arguments. If there are no arguments, it returns 0, which is the identity for this operation.

plus and $+ are Maclisp names, supported for compatibility.

- *arg* &rest *args*
-$ *arg* &rest *args*

With only one argument, - returns the negative of its argument. With more than one argument, - returns its first argument minus all of the rest of its arguments.

Examples:

```
(- 1) => -1
(- -3.0) => 3.0
(- 3 1) => 2
(- 9 2 1) => 6
```

-$ is a Maclisp name, supported for compatibility.

minus *x*

Returns the negative of *x*, just like - with one argument.

difference *arg* &rest *args*

Returns its first argument minus all of the rest of its arguments. If there are at least two arguments, difference is equivalent to -.

abs *x*

Returns $|x|$, the absolute value of the number *x*. abs for real numbers could have been defined as

```
(defun abs (x)
    (cond ((minusp x) (minus x))
          (t x)))
```

abs of a complex number could be computed, though imprecisely, as

```
(sqrt (^ (realpart x) 2) (^ (imagpart x) 2))
```

***** &rest *args*
times &rest *args*
***$** &rest *args*

Returns the product of its arguments. If there are no arguments, it returns 1, which is the identity for this operation.

times and *$ are Maclisp names, supported for compatibility.

// *arg* &rest *args*
//$ *arg* &rest *args*

With more than one argument, // it returns the first argument divided by all of the rest of its arguments. With only one argument, (// *x*) is the same as (// 1 *x*).

The name of this function is written // rather than / because / is the escape character in traditional Lisp syntax and must be escaped in order to suppress that significance. //$ is a Maclisp name, supported for compatibility.

// of two integers returns an integer even if the mathematically correct value is not an integer. More precisely, the value is the same as the first value returned by truncate (see below). This will eventually be changed, and then the value will be a ratio if necessary so that the it is mathematically correct. All code that relies on // to return an integer value rather than a ratio should be converted to use truncate (or floor or ceiling, which may simplify the code further). In the mean time, use the function cli:// if you want a rational result.

Examples:

```
(// 3 2) => 1           ;Fixnum division truncates.
(// 3 -2) => -1
(// -3 2) => -1
(// -3 -2) => 1
(// 3 2.0) => 1.5
(// 3 2.0s0) => 1.5s0
(// 4 2) => 2
(// 12. 2. 3.) => 2
(// 4.0) => .25
```

quotient *arg* &rest *args*

Returns the first argument divided by all of the rest of its arguments. When there are two or more arguments, quotient is equivalent to //.

cli:// *number* &rest *numbers*

This is the Common Lisp division function. It is like // except that it uses exact rational division when the arguments are integers.

// will someday be changed to divide integers exactly. Then there will no longer be a distinct function cli://; that name will become equivalent to //.

Note that in Common Lisp syntax you would write just / rather than cli://.

There are four functions for "integer division", the sort which produces a quotient and a remainder. They differ in how they round the quotient to an integer, and therefore also in the sign of the remainder. The arguments must be real, since ordering is needed to compute the value. The quotient is always an integer, but the arguments and remainder need not be.

floor *x* &optional (*y* 1)

floor's first value is the largest integer less than or equal to the quotient of *x* divided by *y*.

The second value is the remainder, *x* minus *y* times the first value. This has the same sign as *y* (or may be zero), regardless of the sign of *x*.

With one argument, floor's first value is the largest integer less than or equal to the argument.

ceiling *x* &optional (*y* 1)

ceiling's first value is the smallest integer greater than or equal to the quotient of *x* divided by *y*.

The second value is the remainder, *x* minus *y* times the first value. This has the opposite sign from *y* (or may be zero), regardless of the sign of *x*.

With one argument, ceiling's first value is the smallest integer greater than or equal to the argument.

truncate *x* &optional (*y* 1)

truncate is the same as floor if the arguments have the same sign, ceiling if they have opposite signs. truncate is the function that the divide instruction on most computers implements.

truncate's first value is the nearest integer, in the direction of zero, to the quotient of *x* divided by *y*.

The second value is the remainder, *x* minus *y* times the first value. This has the same sign as *x* (or may be zero).

round *x* &optional (*y* 1)

round's first value is the nearest integer to the quotient of *x* divided by *y*. If the quotient is midway between two integers, the even integer of the two is used.

The second value is the remainder, *x* minus *y* times the first value. The sign of this remainder cannot be predicted from the signs of the arguments alone.

With one argument, round's first value is the integer nearest to the argument.

Here is a table which clarifies the meaning of floor, ceiling, truncate and round with one argument:

	floor	ceiling	truncate	round
2.6	2	3	2	3
2.5	2	3	2	2
2.4	2	3	2	2
0.7	0	1	0	1
0.3	0	1	0	0
-0.3	-1	0	0	0
-0.7	-1	0	0	-1
-2.4	-3	-2	-2	-2
-2.5	-3	-2	-2	-2
-2.5	-3	-2	-2	-2
-2.6	-3	-2	-2	-3

There are two kinds of remainder function, which differ in the treatment of negative numbers. The remainder can also be obtained as the second value of one of the integer division functions above, but if only the remaineder is desired it is simpler to use these functions.

\ *x y*
remainder *x y*
cli:rem *x y*

Returns the remainder of *x* divided by *y*. *x* and *y* must be integers (fixnums or bignums). This is the same as the second value of (truncate *x y*). Only the absolute value of the divisor is relevant.

```
(\ 3 2) => 1
(\ -3 2) => -1
(\ 3 -2) => 1
(\ -3 -2) => -1
```

Common Lisp gives this function the name rem, but since rem in traditional Zetalisp is a function to remove elements from lists (see page 105), the name rem is defined to mean remainder only in Common Lisp programs. Note that the name \ would have to be written as \\ in Common Lisp syntax; but the function \ is not standard Common Lisp.

mod *number divisor*

Returns the root of *number* modulo *divisor*. This is a number between 0 and *divisor*, or possibly 0, whose difference from *number* is a multiple of *divisor*. It is the same as the second value of (floor *number divisor*). Examples:

```
(mod 2 5)  =>  2
(mod -2 5)  =>  3
(mod -2 -5)  =>  -2
(mod 2 -5)  =>  -3
```

There are four "floating point integer division" functions. These produce a result which is a floating point number whose value is exactly integral.

ffloor *x* &optional (*y* 1)
fceiling *x* &optional (*y* 1)
ftruncate *x* &optional (*y* 1)
fround *x* &optional (*y* 1)

Like floor, ceiling, truncate and round except that the first value is converted from an integer to a float. If *x* is a float, then the result is the same type of float as *x*.

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

Dividing by zero, using any of the above division functions, signals this condition. The :function operation on the condition instance returns the name of the division function. The :dividend operation may be available to return the number that was divided.

1+ *x*
add1 *x*
1+$ *x*

(1+ x) is the same as (+ x 1). The other two names are for Maclisp compatibility.

1- *x*
sub1 *x*
1-$ *x*

(1- x) is the same as (- x 1). Note that the short name may be confusing: (1- *x*) does *not* mean 1-*x*; rather, it means *x*-1. The names sub1 and 1-$ are for Maclisp compatibility.

gcd &rest *integers*
\\ &rest *integers*

Returns the greatest common divisor of all its arguments, which must be integers. With one argument, the value is that argument. With no arguments, the value is zero.

In Common Lisp syntax \\ would be written as \\\\, but only the name gcd is valid in strict Common Lisp.

lcm *integer* &rest *more-integers*

Returns the least common multiple of the specified integers.

expt *x y*
^ *x y*
^$ *x y*

Returns *x* raised to the *y*'th power. The result is rational (and possibly an integer) if *x* is rational and *y* an integer. If the exponent is an integer a repeated-squaring algorithm is used; otherwise the result is (exp (* *y* (log *x*))).

If *y* is zero, the result is (+ 1 (* *x y*)); this is equal to one, but its type depends on those of *x* and *y*.

sys:zero-to-negative-power (sys:arithmetic-error error) *Condition*

This condition is signaled when **expt**'s first argument is zero and the second argument is negative.

sqrt *x*

Returns the square root of *x*. A mathematically unavoidable discontinuity occurs for negative real arguments, for which the value returned is a positive real times **i**.

```
(sqrt 4) => 2
(sqrt -4) => 0+2i
(sqrt -4+.0001i) => .00005+2i (approximately)
(sqrt -4-.0001i) => .00005-2i (approximately)
```

isqrt *x*

Integer square-root. *x* must be an integer; the result is the greatest integer less than or equal to the exact square root of *x*.

***dif** *x y*
***plus** *x y*
***quo** *x y*
***times** *x y*

These are the internal microcoded arithmetic functions. There is no reason why anyone should need to write code with these explicitly, since the compiler knows how to generate the appropriate code for **plus**, **+**, etc. These names are only here for Maclisp compatibility.

%div *dividend divisor*

The internal division function used by **cli://**, it was available before **cli://** was and may therefore be used in some programs. It takes exactly two arguments. Uses of **%div** should be changed to use **cli://**.

7.4 Complex Number Functions

See also the predicates **realp** and **complexp** (page 12).

complex *x* &optional *y*

Returns the complex number whose real part is *x* and whose imaginary part is *y*.

If *x* is rational and *y* is zero or omitted, the value is *x*, and not a complex number at all. If *x* is a float and *y* is zero or omitted, of if *y* is a floating zero, the result is a complexnum whose imaginary part is zero.

realpart *z*

Returns the real part of the number *z*. If *z* is real, this is the same as *z*.

imagpart *z*
Returns the imaginary part of the number *z*. If *z* is real, this is zero.

conjugate *z*
Returns the complex conjugate of the number *z*. If *z* is real, this is the same as *z*.

phase *z*
Returns the phase angle of the complex number *z* in its polar form. This is the angle from the positive *x* axis to the ray from the origin through *z*. The value is always in the interval (-π, π].

```
(phase -4) => π
(phase -4-.0001i) is just over -π.
(phase 0) => 0 (an arbitrary choice)
```

cis *angle*
Returns the complex number of unit magnitude whose phase is *angle*. This is equal to (complex (cos *angle*) (sin *angle*)). *angle* must be real.

signum *z*
Returns a number with unit magnitude and the same type and phase as *z*. If *z* is zero, the value is zero.

If *z* is real, the value is = to 1 or -1; it may be a float, however.

7.5 Transcendental Functions

These functions are only for floating-point arguments; if given an integer they convert it to a float. If given a short float, they return a short float.

pi *Constant*
The value of π, as a full-size float.

exp *x*
Returns *e* raised to the *x*'th power, where *e* is the base of natural logarithms.

log *x* &optional *base*
Returns the logarithm of *x* to base *base*. *base* defaults to *e*. When *base* is *e*, the imaginary part of the value is in the interval (-π, π]; for negative real *x*, the value has imaginary part π.

If *base* is specified, the result is
```
(// (log x) (log base))
```

sys:zero-log (sys:arithmetic-error error) *Condition*
This is signaled when the argument to log is zero.

sin *x*
cos *x*
tan *x*

Return, respectively, the sine, cosine and tangent of *x*, where *x* is expressed in radians. *x* may be complex.

sind *x*
cosd *x*
tand *x*

Return, respectively, the sine, cosine and tangent of *x*, where *x* is expressed in degrees.

asin *x*
acos *x*

Returns the angle (in radians) whose sine (respectively, cosine) is *x*. The real part of the result of asin is between $-\pi/2$ and $\pi/2$; acos and asin of any given argument always add up to $\pi/2$.

atan *y* &optional *x*

If only *y* is given, the value is the angle, in radians, whose tangent is *y*. The real part of the result is between zero and $-\pi$.

If *x* is also given, both arguments must be real, and the value is an angle, in radians, whose tangent is y/x. However, the signs of the two arguments are used to choose between two angles which differ by π and have the same tangent. The one chosen is the angle from the *x*-axis counterclockwise to the line from the origin to the point (x, y).

atan always returns a non-negative number between zero and 2π.

atan2 *y* &optional *x*
cli:atan *y* &optional *x*

Like atan but always returns a value whose real part is between $-\pi/2$ and $\pi/2$. The value is either the same as the value of atan or differs from it by π.

atan2 is the traditional name of this function. In Common Lisp it is called atan; it is documented as cli:atan since the name atan has a different meaning in traditional syntax.

sinh *x*
cosh *x*
tanh *x*
asinh *x*
acosh *x*
atanh *x*

The hyperbolic and inverse hyperbolic functions.

7.6 Numeric Type Conversions

These functions are provided to allow specific conversions of data types to be forced, when desired.

float *number* &optional *float*

Converts *number* to a floating point number and returns it.

If *float* is specified, the result is of the same floating point format as *float*. If *number* is a float of a different format then it is converted.

If *float* is omitted, then *number* is converted to a single-float unless it is already a floating point number.

A complex number is converted to one whose real and imaginary parts are full-size floats unless they are already both floats.

small-float *x*
short-float *x*

Converts any kind of real number to a short-float. A complex number is converted to one whose real and imaginary parts are short floats. The two names are synonymous.

numerator *x*

Returns the numerator of the rational number *x*. If *x* is an integer, the value equals *x*. If *x* is not an integer or ratio, an error is signaled.

denominator *x*

Returns the denominator of the rational number *x*. If *x* is an integer, the value is 1. If *x* is not an integer or ratio, an error is signaled.

rational *x*

Converts *x* to a rational number. If *x* is an integer or a ratio, it is returned unchanged. If it is a floating point number, it is regarded as an exact fraction whose numerator is the mantissa and whose denominator is a power of two. For any other argument, an error is signaled.

rationalize *x* &optional *precision*

Returns a rational approximation to *x*.

If there is only one argument, and it is an integer or a ratio, it is returned unchanged. If the argument is a floating point number, a rational number is returned which, if converted to a floating point number, would produce the original argument. Of all such rational numbers, the one chosen has the smallest numerator and denominator.

If there are two arguments, the second one specifies how much precision of the first argument should be considered significant. *precision* can be a positive integer (the number of bits to use), a negative integer (the number of bits to drop at the end), or a floating point number (minus its exponent is the number of bits to use).

If there are two arguments and the first is rational, the value is a "simpler" rational which approximates it.

fix *x*

Converts *x* from a float or ratio to an integer, truncating towards negative infinity. The result is a fixnum or a bignum as appropriate. If *x* is already a fixnum or a bignum, it is returned unchanged.

fix is the same as floor except that floor returns an additional value. fix is semi-obsolete, since the functions floor, ceiling, truncate and round provide four different ways of converting numbers to integers with different kinds of rounding.

fixr *x*

fixr is the same as round except that round returns an additional value. fixr is considered obsolete.

7.7 Floating Point Numbers

decode-float *float*

Returns three values which describe the value of *float*.

The first value is a positive float of the same format having the same mantissa, but with an exponent chosen to make it between 1/2 and 1, less than 1.

The second value is the exponent of *float*: the power of 2 by which the first value needs to be scaled in order to get *float* back.

The third value expresses the sign of *float*. It is a float of the same format as *float*, whose value is either 1 or -1. Example:

```
(decode-float 38.2)
  =>  0.596875   6   1.0
```

integer-decode-float *float*

Like decode-float except that the first value is scaled so as to make it an integer, and the second value is modified by addition of a constant to compensate.

```
(integer-decode-float 38.2)
  =>  #o11431463146   -25.   1.0
```

scale-float *float integer*

Multiplies *float* by 2 raised to the *integer* power. *float* can actually be an integer also; it is converted to a float and then scaled.

```
(scale-float 0.596875 6)  =>  38.2
(scale-float #o11431463146 -25.)  =>  38.2
```

float-sign *float1* &optional *float2*

Returns a float whose sign matches that of *float1* and whose magnitude and format are those of *float2*. If *float2* is omitted, 1.0 is used as the magnitude and *float1* 's format is used.

```
(float-sign -1.0s0 35.3)  =>  -35.3
(float-sign -1.0s0 35.3s0)  =>  -35.3s0
```

float-radix *float*

Defined by Common Lisp to return the radix used for the exponent in the format used for *float*. On the Lisp Machine, floating point exponents are always powers of 2, so float-radix ignores its argument and always returns 2.

float-digits *float*

Returns the number of bits of mantissa in the floating point format which float is an example of. It is 17 for short floats and 31 for full size ones.

float-precision *float*

Returns the number of significant figures present in in the mantissa of *float*. This is always the same as (float-digits *float*) for normalized numbers, and on the Lisp Machine all floats are normalized, so the two functions are the same.

7.8 Logical Operations on Numbers

Except for lsh and rot, these functions operate on both fixnums and bignums. lsh and rot have an inherent word-length limitation and hence only operate on 25-bit fixnums. Negative numbers are operated on in their 2's-complement representation.

logior &rest *integers*

Returns the bit-wise logical *inclusive or* of its arguments. With no arguments, the value is zero, which is the identity for this operation.
Example (in octal):

```
(logior #o4002 #o67) => #o4067
```

logand &rest *integers*

Returns the bit-wise logical *and* of its arguments. With no arguments, the value is -1, which is the identity for this operation.
Examples (in octal):

```
(logand #o3456 #o707) => #o406
(logand #o3456 #o-100) => #o3400
```

logxor &rest *integers*

Returns the bit-wise logical *exclusive or* of its arguments. With no arguments, the value is zero, which is the identity for this operation.
Example (in octal):

```
(logxor #o2531 #o7777) => #o5246
```

logeqv &rest *integers*

Combines the *integers* together bitwise using the equivalence operation, which, for two arguments, is defined to result in 1 if the two argument bits are equal. This operation is asociative. With no args, the value is -1, which is an identity for the equivalence operation.

Example (in octal):

```
        (logeqv #o2531 #o7707) => #o-5237 = ...77772541
```

Non-associative bitwise operations take only two arguments:

lognand *integer1 integer2*

Returns the bitwise-nand of the two arguments. A bit of the result is 1 if at least one of the corresponding argument bits is 0.

lognor *integer1 integer2*

Returns the bitwise-nor of the two arguments. A bit of the result is 1 if both of the corresponding argument bits are 0.

logorc1 *integer1 integer2*

Returns the bitwise-or of *integer2* with the complement of *integer1*.

logorc2 *integer1 integer2*

Returns the bitwise-or of *integer1* with the complement of *integer2*.

logandc1 *integer1 integer2*

Returns the bitwise-and of *integer2* with the complement of *integer1*.

logandc2 *integer1 integer2*

Returns the bitwise-and of *integer1* with the complement of *integer2*.

lognot *number*

Returns the logical complement of *number*. This is the same as **logxor**'ing *number* with -1.

Example:

```
        (lognot #o3456) => #o-3457
```

boole *fn* &rest *one-or-more-args*

boole is the generalization of **logand**, **logior**, and **logxor**. *fn* should be a fixnum between 0 and 17 octal inclusive; it controls the function which is computed. If the binary representation of *fn* is *abcd* (*a* is the most significant bit, *d* the least) then the truth table for the Boolean operation is as follows:

```
           y
       | 0   1
    ---------
      0| a   c
    x  |
      1| b   d
```

If boole has more than three arguments, it is associated left to right; thus,

```
        (boole fn x y z) = (boole fn (boole fn x y) z)
```

With two arguments, the result of boole is simply its second argument. At least two arguments are required.

Examples:

```
        (boole 1 x y) = (logand x y)
        (boole 6 x y) = (logxor x y)
        (boole 2 x y) = (logand (lognot x) y)
```

logand, logior, and so on are usually preferred over the equivalent forms of boole. boole is useful when the operation to be performed is not constant.

boole-ior *Constant*
boole-and *Constant*
boole-xor *Constant*
boole-eqv *Constant*
boole-nand *Constant*
boole-nor *Constant*
boole-orc1 *Constant*
boole-orc2 *Constant*
boole-andc1 *Constant*
boole-andc2 *Constant*

The boole opcodes that correspond to the functions logior, logand, etc.

boole-clr *Constant*
boole-set *Constant*
boole-1 *Constant*
boole-2 *Constant*

The boole opcodes for the four trivial operations. Respectively, they are those which always return zero, always return one, always return the first argument, and always return the second argument.

bit-test *x y*
logtest *x y*

bit-test is a predicate which returns t if any of the bits designated by the 1's in *x* are 1's in *y*. bit-test is implemented as a macro which expands as follows:

```
        (bit-test x y) ==> (not (zerop (logand x y)))
```

logtest is the Common Lisp name for this function.

lsh *x y*

Returns *x* shifted left *y* bits if *y* is positive or zero, or *x* shifted right |*y*| bits if *y* is negative. Zero bits are shifted in (at either end) to fill unused positions. *x* and *y* must be fixnums. (In some applications you may find ash useful for shifting bignums; see below.)

Examples:
```
        (lsh 4 1) => #o10
        (lsh #o14 -2) => 3
        (lsh -1 1) => -2
```

ash *x y*
Shifts *x* arithmetically left *y* bits if *y* is positive, or right -*y* bits if *y* is negative. Unused positions are filled by zeroes from the right, and by copies of the sign bit from the left. Thus, unlike lsh, the sign of the result is always the same as the sign of *x*. If *x* is a fixnum or a bignum, this is a shifting operation. If *x* is a float, this does scaling (multiplication by a power of two), rather than actually shifting any bits.

rot *x y*
Returns *x* rotated left *y* bits if *y* is positive or zero, or *x* rotated right |*y*| bits if *y* is negative. The rotation considers *x* as a 25-bit number (unlike Maclisp, which considers *x* to be a 36-bit number in both the pdp-10 and Multics implementations). *x* and *y* must be fixnums. (There is no function for rotating bignums.)
Examples:
```
        (rot 1 2) => 4
        (rot 1 -2) => #o20000000
        (rot -1 7) => -1
        (rot #o15 25.) => #o15
```

logcount *integer*
Returns the number of 1 bits in *integer*, if it is positive. Returns the number of 0 bits in *integer*, if it is negative. (There are infinitely many 1 bits in a negative integer.)
```
        (logcount #o15)  =>  3
        (logcount #o-15)  =>  2
```

integer-length *integer*
The minimum number of bits (aside from sign) needed to represent *integer* in two's complement. This is the same as haulong for positive numbers.
```
        (integer-length 0) => 0
        (integer-length 7) => 3
        (integer-length 8) => 4
        (integer-length -7) => 3
        (integer-length -8) => 3
        (integer-length -9) => 4
```

haulong *integer*
The same as integer-length of the absolute value of integer. This name exists for Maclisp compatibility only.

haipart *x n*
Returns the high *n* bits of the binary representation of |*x*|, or the low -*n* bits if *n* is negative. *x* may be a fixnum or a bignum; its sign is ignored. haipart could have been defined by:

```
(defun haipart (x n)
  (setq x (abs x))
  (if (minusp n)
      (logand x (1- (ash 1 (- n))))
      (ash x (min (- n (haulong x))
                  0))))
```

7.9 Byte Manipulation Functions

Several functions are provided for dealing with an arbitrary-width field of contiguous bits appearing anywhere in an integer (a fixnum or a bignum). Such a contiguous set of bits is called a *byte*. Note that we are not using the term *byte* to mean eight bits, but rather any number of bits within a number. These functions use numbers called *byte specifiers* to
designate a specific byte position within any word. A byte specifier contains two pieces of information: the size of the byte, and the position of the byte. The position is expressed as the number of least significant bits which are not included in the byte. A position of zero means that the byte is at the right (least significant) end of the number.

The maximum value of the size is 24, since a byte must fit in a fixnum although bytes can be loaded from and deposited into bignums. (Bytes are always positive numbers.)

Byte specifiers are represented as fixnums whose two lowest octal digits represent the *size* of the byte, and whose higher (usually two, but sometimes more) octal digits represent the *position* of the byte within a number. For example, the byte-specifier #o0010 (i.e. 10 octal) refers to the lowest eight bits of a word, and the byte-specifier #o1010 refers to the next eight bits. The format of byte-specifiers is taken from the pdp-10 byte instructions.

Much old code contains byte specifiers written explicitly as octal numbers. It is cleaner to construct byte specifiers using **byte** instead. Decomposition of byte specifiers should always be done with **byte-position** and **byte-size**, as at some time in the future other kinds of byte specifiers may be created to refer to fields whose size is greater than #o77.

byte *size position*
Returns a byte specifier for the byte of *size* bits, positioned to exclude the *position* least significant bits. This byte specifier can be passed as the first argument to **ldb**, **dpb**, **%logldb**, **%logdpb**, **mask-field**, **%p-ldb**, **%p-ldb-offset**, and so on.

byte-position *byte-spec*
byte-size *byte-spec*
Return, respectively, the size and the position of *byte-spec*. It is always true that
`(byte (byte-size` *byte-spec*`) (byte-position` *byte-spec*`))`
equals *byte-spec*.

ldb *byte-spec integer*
Extracts a byte from *integer* according to *byte-soec*. The contents of this byte are returned right-justified in a fixnum. The name of the function, **ldb**, means 'load byte'. *integer* may be a fixnum or a bignum. The returned value is always a fixnum.

Example:
```
        (ldb (byte 6 3) #o4567) => #o56
```

load-byte *integer position size*
This is like ldb except that instead of using a byte specifier, the *position* and *size* are passed as separate arguments. The argument order is not analogous to that of ldb so that load-byte can be compatible with Maclisp.

ldb-test *byte-spec integer*
ldb-test is a predicate which returns t if any of the bits designated by the byte specifier *byte-spec* are 1's in *integer*. That is, it returns t if the designated field is non-zero. ldb-test is implemented as a macro which expands as follows:
```
        (ldb-test byte-spec integer) ==> (not (zerop (ldb byte-spec integer)))
```

logbitp *index integer*
t if the bit *index* up from the least significant in *integer* is a 1. This is equivalent to (ldb-test (byte *index* 1) *integer*).

mask-field *byte-spec fixnum*
This is similar to ldb; however, the specified byte of *fixnum* is positioned in the same byte of the returned value. The returned value is zero outside of that byte. *fixnum* must be a fixnum.
Example:
```
        (mask-field (byte 6 3) #o4567) => #o560
```

dpb *byte byte-spec integer*
Returns a number which is the same as *integer* except in the bits specified by *byte-spec*. The low bits of *byte*, appropriately many, are placed in those bits. *byte* is interpreted as being right-justified, as if it were the result of ldb. *integer* may be a fixnum or a bignum. The name means 'deposit byte'.
Example:
```
        (dpb #o23 (byte 6 3) #o4567) => #o4237
```

deposit-byte *integer position size byte*
This is like dpb except that instead of using a byte specifier, the *position* and *size* are passed as separate arguments. The argument order is not analogous to that of dpb so that deposit-byte can be compatible with Maclisp.

deposit-field *byte byte-spec fixnum*
This is like dpb, except that *byte* is not taken to be left-justified; the *byte-spec* bits of *byte* are used for the *byte-spec* bits of the result, with the rest of the bits taken from *fixnum*. *fixnum* must be a fixnum.
Example:
```
        (deposit-field #o230 (byte 6 3) #o4567) => #o4237
```

The behavior of the following two functions depends on the size of fixnums, and so functions using them may not work the same way on future implementations of Zetalisp. Their names start with % because they are more like machine-level subprimitives than the previous functions.

%logldb *byte-spec fixnum*

%logldb is like ldb except that it only loads out of fixnums and allows a byte size of 25, i.e. all 25 bits of the fixnum including the sign bit.

%logdpb *byte byte-spec fixnum*

%logdpb is like dpb except that it only deposits into fixnums. Using this to change the sign-bit leaves the result as a fixnum, while dpb would produce a bignum result for arithmetic correctness. %logdpb is good for manipulating fixnum bit-masks such as are used in some internal system tables and data-structures.

7.10 Random Numbers

The functions in this section provide a pseudo-random number generator facility. The basic function you use is random, which returns a new pseudo-random number each time it is called.

random &optional *number random-state*

Returns a randomly generated number. If *number* is specified, the random number is of the same type as *number* (floating if *number* is floating, etc.), nonnegative, and less than *number*.

If *number* is omitted, the result is a randomly chosen fixnum, with all fixnums being equally likely.

If *random-state* is present, it is used and updated in generating the random number. Otherwise, the default random-state (the value of *random-state*) is used (and is created if it doesn't already exist). The algorithm is executed inside a without-interrupts (see page 684) so two processes can use the same random-state without colliding.

si:random-in-range *low high*

Returns a random float in the interval [*low*, *high*). The default random-state is used.

A *random-state* is a named structure of type random-state whose contents control the future actions of the random number generator. Each time you call the function random, it uses (and updates) one random-state. One random-state exists standardly and is used by default. To have several different controllable, resettable sources of random numbers, you can create your own random-states. Random-states print as

```
#s(random-state ...more data...)
```

so that they can be read back in.

random-state-p *object*

t if *object* is a random-state.

random-state *Variable*

This random-state is used by default when random is called and the random-state is not explicitly specified.

make-random-state &optional *random-state*

Creates and returns a new random-state object. If *random-state* is **nil**, the new random-state is a copy of ***random-state***. If *random-state* is a random-state, the new one is a copy of that one. If *random-state* is **t**, the new random-state is initialized truly randomly (based on the value of **(time)**).

A random-state actually consists of an array of numbers and two pointers into the array. The pointers circulate around the array; each time a random number is requested, both pointers are advanced by one, wrapping around at the end of the array. Thus, the distance forward from the first pointer to the second pointer stays the same, allowing for wraparound. Let the length of the array be *length* and the distance between the pointers be *offset*. To generate a new random number, each pointer is set to its old value plus one, modulo *length*. Then the two elements of the array addressed by the pointers are added together; the sum is stored back into the array at the location where the second pointer points, and is returned as the random number after being normalized into the right range.

This algorithm produces well-distributed random numbers if *length* and *offset* are chosen carefully, so that the polynomial *x* ^ *length* + *x* ^ *offset* + 1 is irreducible over the mod-2 integers. The system uses 71. and 35.

The contents of the array of numbers should be initialized to anything moderately random, to make the algorithm work. The contents get initialized by a simple random number generator, based on a number called the *seed*. The initial value of the seed is set when the random-state is created, and it can be changed.

si:random-create-array *length offset seed* &optional (*area* **nil**)

Creates and returns a new random-state according to precise specifications. *length* is the length of the array. *offset* is the distance between the pointers and should be an integer less than *length*. *seed* is the initial value of the seed, and should be a fixnum. This calls **si:random-initialize** on the random state before returning it.

si:random-initialize *random-state* &optional *new-seed*

random-state must be a random-state, such as is created by **si:random-create-array**. If *new-seed* is provided, it should be a fixnum, and the seed is set to it. **si:random-initialize** reinitializes the contents of the array from the seed (calling **random** changes the contents of the array and the pointers, but not the seed).

7.11 Information on Numeric Precision

Common Lisp defines some constants whose values give information in a standard way about the ranges of numbers representable in the individual Lisp implementation.

most-negative-fixnum *Constant*

Any integer smaller than this must be a bignum.

most-positive-fixnum *Constant*
Any integer larger than this must be a bignum.

most-positive-short-float *Constant*
No short float can be greater than this number.

least-positive-short-float *Constant*
No positive short float can be closer to zero than this number.

least-negative-short-float *Constant*
No negative short float can be closer to zero than this number.

most-negative-short-float *Constant*
No short float can be less than this (negative) number.

most-positive-single-float *Constant*
least-positive-single-float *Constant*
least-negative-single-float *Constant*
most-negative-single-float *Constant*
Similar to the above, but for full-size floats rather than for short floats.

most-positive-double-float *Constant*
least-positive-double-float *Constant*
least-negative-double-float *Constant*
most-negative-double-float *Constant*
most-positive-long-float *Constant*
least-positive-long-float *Constant*
least-negative-long-float *Constant*
most-negative-long-float *Constant*
These are defined by Common Lisp to be similar to the above, but for double-floats and long-floats. On the Lisp Machine, there are no distinct double and long floating formats; they are synonyms for single-floats. So these constants exist but their values are the same as those of most-positive-single-float and so on.

short-float-epsilon *Constant*
Smallest positive short float which can be added to 1.0s0 and make a difference. That is, for any short float *x* less than this, (+ 1.0s0 *x*) equals 1.0s0.

single-float-epsilon *Constant*
double-float-epsilon *Constant*
long-float-epsilon *Constant*
Smallest positive float which can be added to 1.0 and make a difference. The three names are synonyms on the Lisp Machine, for reasons explained above.

short-float-negative-epsilon *Constant*
Smallest positive short float which can be subtracted from 1.0s0 and make a difference.

single-float-negative-epsilon *Constant*
double-float-negative-epsilon *Constant*
long-float-negative-epsilon *Constant*

Smallest positive float which can be subtracted from 1.0 and make a difference.

7.12 Arithmetic Ignoring Overflow

Sometimes it is desirable to have a form of arithmetic which has no overflow checking (that would produce bignums), and truncates results to the word size of the machine.

%pointer-plus *pointer-1 pointer-2*

Returns a fixnum which is *pointer-1* plus *pointer-2*, modulo what could be stored in the size of the pointer field (currently 25 bits). Arguments other than fixnums are rarely useful, but no type checks are made.

%pointer-difference *pointer-1 pointer-2*

Returns a fixnum which is *pointer-1* minus *pointer-2*. If the arguments are fixnums, rather than true pointers, this provides subtraction modulo what can be stored in the pointer field.

%pointer-times *pointer-1 pointer-2*

Returns a fixnum which is *pointer-1* times *pointer-2*. Arguments other than fixnums are rarely useful, but no type checks are made. The two pointer fields are regarded as signed numbers.

7.13 24-Bit Arithmetic

Sometimes it is useful to have a form of truncating arithmetic with a strictly specified field width which is independent of the range of fixnums permissible on a particular machine. In Zetalisp, this is provided by the following set of functions. Their answers are correct only modulo 2↑24.

These functions should *not* be used for efficiency; they are probably less efficient than the functions which *do* check for overflow. They are intended for algorithms which require this sort of arithmetic, such as hash functions and pseudo-random number generation.

%24-bit-plus *x y*

Returns the sum of *x* and *y* modulo 2↑24. Both arguments must be fixnums.

%24-bit-difference *x y*

Returns the difference of *x* and *y* modulo 2↑24. Both arguments must be fixnums.

%24-bit-times *x y*

Returns the product of *x* and *y* modulo 2↑24. Both arguments must be fixnums.

7.14 Double-Precision Arithmetic

These peculiar functions are useful in programs that don't want to use bignums for one reason or another. They should usually be avoided, as they are difficult to use and understand, and they depend on special numbers of bits and on the use of twos-complement notation.

A double-precision number has 50 bits, of which one is the sign bit. It is represented as two fixnums. The less signficant fixnum conveys 25 signficant bits and is regarded as unsigned (that is, what is normally the sign bit is treated as an ordinary data bit); the more significant fixnum has the same sign as the double-precision number. Only %float-double handles negative double-precision numbers; for the other functions, the more signficant fixnum is always positive and contains only 24 bits of actual data.

%multiply-fractions *num1 num2*

Returns bits 25 through 48 (the most significant half) of the product of *num1* and *num2*, regarded as unsigned integers. If you call this and %pointer-times on the same arguments *num1* and *num2*, you can combine the results into a double-precision product. If *num1* and *num2* are regarded as two's-complement fractions, $-1 \le num < 1$, %multiply-fractions returns 1/2 of their correct product as a fraction.

[The name of this function isn't too great.]

%divide-double *dividend[25:48] dividend[0:24] divisor*

Divides the double-precision number given by the first two arguments by the third argument, and returns the single-precision quotient. Causes an error if *divisor* is zero or if the quotient won't fit in single precision.

There are only 24 bits in each half of the number, as neither sign bit is used to convey information.

%remainder-double *dividend[25:48] dividend[0:24] divisor*

Divides the double-precision number given by the first two arguments by the third argument, and returns the remainder. Causes an error if *divisor* is zero.

%float-double *high25 low25*

high25 and *low25*, which must be fixnums, are concatenated to produce a 50-bit unsigned positive integer. A full-size float containing the same value is constructed and returned. Note that only the 31 most significant bits are retained (after removal of leading zeroes.) This function is mainly for the benefit of read.

8. Arrays

An *array* is a Lisp object that consists of a group of cells, each of which may contain an object. The individual cells are selected by numerical *subscripts*. The type predicate **arrayp** (page 12) can be used to test whether an object is an array.

The *rank* of an array (the number of dimensions which the array has) is the number of subscripts used to refer to one of the elements of the array. The rank may be any integer from zero to seven, inclusively. An array of rank zero has a single element which is addressed using no subscripts. An array of rank one is called a *vector*; the predicate **vectorp** (see page 12) tests whether an object is a vector. A series of functions called the generic sequence functions accept either a vector or a list as argument indiscriminantly (see chapter 9, page 188).

array-rank-limit *Constant*

A constant giving the upper limit on the rank of an array. It is 8, indicating that 7 is the highest possible rank.

The lowest value for any subscript is zero; the highest value is a property of the array. Each dimension has a size, which is the lowest number which is too great to be used as a subscript. For example, in a one-dimensional array of five elements, the size of the one and only dimension is five, and the acceptable values of the subscript are zero, one, two, three, and four.

array-dimension-limit *Constant*

Any one dimension of an array must be smaller than this constant.

The *total size* of an array is the number of elements in it. It is the product of the sizes of the dimensions of the array.

array-total-size-limit *Constant*

The total number of elements of any array must be smaller than this constant.

A vector can have a *fill pointer* which is a number saying how many elements of the vector are *active*. For many purposes, only that many elements (starting with element zero) are used.

The most basic primitive functions for handling arrays are: **make-array**, which is used for the creation of arrays, **aref**, which is used for examining the contents of arrays, and **aset**, which is used for storing into arrays.

An array is a regular Lisp object, and it is common for an array to be the binding of a symbol, or the car or cdr of a cons, or, in fact, an element of an array. There are many functions, described in this chapter, which take arrays as arguments and perform useful operations on them.

Another way of handling arrays, inherited from Maclisp, is to treat them as functions. In this case each array has a name, which is a symbol whose function definition is the array. Zetalisp supports this style by allowing an array to be *applied* to arguments, as if it were a function. The arguments are treated as subscripts and the array is referenced appropriately. The **store** special form (see page 187) is also supported. This kind of array referencing is considered to be obsolete and is slower than the usual kind. It should not be used in new programs.

8.1 Array Types

There are several types of arrays, which differ primarily in which kinds of elements they are allowed to hold. Some types of arrays can hold Lisp objects of any type; such arrays are called *general* arrays. The other types of array restrict the possible elements to a certain type, usually a numeric type. Arrays of these types are called *specialized* arrays, or *numeric* arrays if the elements must be numbers. For example, one array type permits only complex numbers with floating components to be stored in the array. Another permits only the numbers zero and one; Common Lisp calls these *bit arrays*. The contents of a black-and-white screen are stored in a bit array. Several predicates exist for finding out which of these classifications an array belongs to: **simple-vector-p** (page 13), **bit-vector-p**, **simple-bit-vector-p**, **stringp** (page 12), and **simple-string-p**.

The array types are known by a set of symbols whose names begin with **art-** (for 'ARray Type').

The most commonly used type is called **art-q**. An **art-q** array simply holds Lisp objects of any type.

Similar to the **art-q** type is the **art-q-list**. Like the **art-q**, its elements may be any Lisp object. The difference is that the **art-q-list** array doubles as a list; the function **g-l-p** takes an **art-q-list** array and returns a list whose elements are those of the array, and whose actual substance is that of the array. If you **rplaca** elements of the list, the corresponding element of the array is changed, and if you store into the array, the corresponding element of the list changes the same way. An attempt to **rplacd** the list causes a **sys:rplacd-wrong-representation-type** error, since arrays cannot implement that operation.

The most important type of specialized array is the *string*, which is a vector of character objects. Character strings are implemented by the **art-string** array type. Many important system functions, including **read**, **print**, and **eval**, treat **art-string** arrays very differently from the other kinds of arrays. There are also many functions specifically for operating on strings, described in chapter 10.

As viewed by Common Lisp programs, the elements of a string are character objects. As viewed by traditional programs, the elements are integers in the range 0 to 255. While most code still accesses strings in the traditional manner and gets integers out, the Common Lisp viewpoint is considered the correct one. See page 203 for a discussion of this conflict of conventions and its effect on programs.

An **art-fat-string** array is a character string with wider characters, containing 16 bits rather than 8 bits. The extra bits are ignored by many string operations, such as comparison, on these strings; typically they are used to hold font information.

There is a set of types called **art-1b**, **art-2b**, **art-4b**, **art-8b**, and **art-16b**; these names are short for '1 bit', '2 bits', and so on. Each element of an **art-*n*b** array is a non-negative fixnum, and only the least significant *n* bits are remembered in the array; all of the others are discarded.

Thus **art-1b** arrays store only 0 and 1, and if you store a 5 into an **art-2b** array and look at it later, you will find a 1 rather than a 5.

These arrays are used when it is known beforehand that the fixnums which will be stored are non-negative and limited in size to a certain number of bits. Their advantage over the **art-q** array is that they occupy less storage, because more than one element of the array is kept in a single machine word. (For example, 32 elements of an **art-1b** array or 2 elements of an **art-16b** array fit into one word).

There are also **art-32b** arrays which have 32 bits per element. Since fixnums only have 24 bits anyway, these are the same as **art-q** arrays except that they only hold fixnums. They are not compatible with the other "bit" array types and generally should not be used.

An **art-half-fix** array contains half-size fixnums. Each element of the array is a signed 16-bit integer; the range is from -32768 to 32767 inclusive.

The **art-float** array type is a special-purpose type whose elements are floats. When storing into such an array the value (any kind of number) is converted to a float, using the **float** function (see page 149). The advantage of storing floats in an **art-float** array rather than an **art-q** array is that the numbers in an **art-float** array are not true Lisp objects. Instead the array remembers the numerical value, and when it is **aref**'ed creates a Lisp object (a float) to hold the value. Because the system does special storage management for bignums and floats that are intermediate results, the use of **art-float** arrays can save a lot of work for the garbage collector and hence greatly increase performance. An intermediate result is a Lisp object passed as an argument, stored in a local variable, or returned as the value of a function, but not stored into a special variable, a non-**art-float** array, or list structure. **art-float** arrays also provide a locality of reference advantage over **art-q** arrays containing floats, since the floats are contained in the array rather than being separate objects probably on different pages of memory.

The **art-fps-float** array type is another special-purpose type whose elements are floats. The internal format of this array is compatible with the PDP-11/VAX single-precision floating-point format. The primary purpose of this array type is to interface with the FPS array processor, which can transfer data directly in and out of such an array.

Any type of number may be stored into an **art-fps-float** array, but it is, in effect, converted to a float, and then rounded off to the 24-bit precision of the PDP-11. If the magnitude of the number is too large, the largest valid floating-point number is stored. If the magnitude is too small, zero is stored.

When an element of an **art-fps-float** array is read, a new float is created containing the value, just as with an **art-float** array.

The **art-complex** array type is a special purpose type whose elements are arbitrary numbers, which may be complex numbers. (Most of the numeric array types can only hold real numbers.) As compared with an ordinary **art-q** array, **art-complex** provides an advantage in garbage collection similar to what **art-float** provides for floating point numbers.

The art-complex-float array type is a special purpose type whose elements are numbers (real or complex) whose real and imaginary parts are both floating point numbers. (If you store a non-floating-point number into the array, its real and imaginary parts are converted to floating point.) This provides maximum advantage in garbage collection if all the elements you wish to store in the array are numbers with floating point real and imaginary parts.

The art-complex-fps-float array type is similar to art-complex-float but each real or imaginary part is stored in the form used by the FPS array processor. Each element occupies two words, the first being the real part and the second being the imaginary part.

There are three types of arrays which exist only for the implementation of *stack groups*; these types are called art-stack-group-head, art-special-pdl, and art-reg-pdl. Their elements may be any Lisp object; their use is explained in the section on stack groups (see chapter 13, page 256).

array-types *Constant*

The value of array-types is a list of all of the array type symbols such as art-q, art-4b, art-string and so on. The values of these symbols are internal array type code numbers for the corresponding type.

array-types *array-type-code*

Given an internal numeric array-type code, returns the symbolic name of that type.

array-elements-per-q *Constant*

array-elements-per-q is an association list (see page 110) which associates each array type symbol with the number of array elements stored in one word, for an array of that type. If the value is negative, it is instead the number of words per array element, for arrays whose elements are more than one word long.

array-elements-per-q *array-type-code*

Given the internal array-type code number, returns the number of array elements stored in one word, for an array of that type. If the value is negative, it is instead the number of words per array element, for arrays whose elements are more than one word long.

array-bits-per-element *Constant*

The value of array-bits-per-element is an association list (see page 110) which associates each array type symbol with the number of bits of unsigned number it can hold, or nil if it can hold Lisp objects. This can be used to tell whether an array can hold Lisp objects or not.

array-bits-per-element *array-type-code*

Given the internal array-type code numbers, returns the number of bits per cell for unsigned numeric arrays, or nil for a type of array that can contain Lisp objects.

array-element-size *array*

Given an array, returns the number of bits that fit in an element of that array. For arrays that can hold general Lisp objects, the result is 25., based on the assumption that you will be storing fixnums in the array.

8.2 Extra Features of Arrays

Any array may have an *array leader*. An array leader is like a one-dimensional **art-q** array which is attached to the main array. So an array which has a leader acts like two arrays joined together. The leader can be stored into and examined by a special set of functions, different from those used for the main array: **array-leader** and **store-array-leader**. The leader is always one-dimensional, and always can hold any kind of Lisp object, regardless of the type or rank of the main part of the array.

Very often the main part of an array is used as a homogeneous set of objects, while the leader is used to remember a few associated non-homogeneous pieces of data. In this case the leader is not used like an array; each slot is used differently from the others. Explicit numeric subscripts should not be used for the leader elements of such an array; instead the leader should be described by a **defstruct** (see page 374).

By convention, element 0 of the array leader of an array is used to hold the number of elements in the array that are "active". When the zeroth element is used this way, it is called a *fill pointer*. Many array-processing functions recognize the fill pointer. For instance, if a string (an array of type **art-string**) has seven elements, but its fill pointer contains the value five, then only elements zero through four of the string are considered to be active; the string's printed representation is five characters long, string-searching functions stop after the fifth element, etc. Fill pointers are a Common Lisp standard, but the array leader which is the Lisp Machine's way of implementing them is not standard.

fill-pointer *array*

Returns the fill pointer of *array*, or **nil** if it does not have one. This function can be used with **setf** to set the array's fill pointer.

The system does not provide a way to turn off the fill-pointer convention; any array that has a leader must reserve element 0 for the fill pointer or avoid using many of the array functions.

Leader element 1 is used in conjunction with the "named structure" feature to associate a user-defined data type with the array; see page 390. Element 1 is treated specially only if the array is flagged as a named structure.

8.2.1 Displaced Arrays

The following explanation of *displaced arrays* is probably not of interest to a beginner; the section may be passed over without losing the continuity of the manual.

Normally, an array is represented as a small amount of header information, followed by the contents of the array. However, sometimes it is desirable to have the header information removed from the actual contents. One such occasion is when the contents of the array must be located in a special part of the Lisp Machine's address space, such as the area used for the control of input/output devices, or the bitmap memory which generates the TV image. Displaced arrays are also used to reference certain special system tables, which are at fixed addresses so the microcode can access them easily.

If you give make-array a fixnum or a locative as the value of the :displaced-to option, it creates a displaced array referring to that location of virtual memory and its successors. References to elements of the displaced array will access that part of storage, and return the contents; the regular aref and aset functions are used. If the array is one whose elements are Lisp objects, caution should be used: if the region of address space does not contain typed Lisp objects, the integrity of the storage system and the garbage collector could be damaged. If the array is one whose elements are bytes (such as an art-4b type), then there is no problem. It is important to know, in this case, that the elements of such arrays are allocated from the right to the left within the 32-bit words.

It is also possible to have an array whose contents, instead of being located at a fixed place in virtual memory, are defined to be those of another array. Such an array is called an *indirect array*, and is created by giving make-array an array as the value of the :displaced-to option. The effects of this are simple if both arrays have the same type; the two arrays share all elements. An object stored in a certain element of one can be retrieved from the corresponding element of the other. This, by itself, is not very useful. However, if the arrays have different rank, the manner of accessing the elements differs. Thus, creating a one-dimensional array of nine elements, indirected to a second, two-dimensional array of three elements by three, allows access to the elements in either a one-dimensional or a two-dimensional manner. Weird effects can be produced if the new array is of a different type than the old array; this is not generally recommended. Indirecting an art-*m*b array to an art-*n*b array does the obvious thing. For instance, if *m* is 4 and *n* is 1, each element of the first array contains four bits from the second array, in right-to-left order.

It is also possible to create an indirect array in such a way that when an attempt is made to reference it or store into it, a constant number is added to the subscript given. This number is called the *index-offset*. It is specified at the time the indirect array is created, by giving a fixnum to make-array as the value of the :displaced-index-offset option. The length of the indirect array need not be the full length of the array it indirects to; it can be smaller. Thus the indirect array can cover just a subrange of the original array. The nsubstring function (see page 216) creates such arrays. When using index offsets with multi-dimensional arrays, there is only one index offset; it is added in to the linearized subscript which is the result of multiplying each subscript by an appropriate coefficient and adding them together.

8.3 Constructing Arrays

vector &rest *elements*

Constructs and returns a vector (one-dimensional array) whose elements are the arguments given.

make-array *dimensions* &rest *options.*

This is the primitive function for making arrays. *dimensions* should be a list of fixnums which are the dimensions of the array; the length of the list is the rank of the array. For convenience you can specify a single fixnum rather than a list of one fixnum, when making a one-dimensional array.

options are alternating keywords and values. The keywords may be any of the following:

:area
: The value specifies in which area (see chapter 16, page 296) the array should be created. It should be either an area number (a fixnum), or **nil** to mean the default area.

:type
: The value should be a symbolic name of an array type; the most common of these is **art-q**, which is the default. The elements of the array are initialized according to the type: if the array is of a type whose elements may only be fixnums or floats, then every element of the array is initially 0 or 0.0; otherwise, every element is initially **nil**. See the description of array types on page 163. The value of the option may also be the value of a symbol which is an array type name (that is, an internal numeric array type code).

:element-type
: *element-type* is the Common Lisp way to control the type of array made. Its value is a Common Lisp type specifier (see section 2.3, page 14). The array type used is the most specialized which can allow as an element anything which fits the type specifier. For example, if *element-type* is **(mod 4)**, you get an **art-2b** array. If *element-type* is **(mod 3)**, you still get an **art-2b** array, that being the most restrictive which can store the numbers 0, 1 and 2. If **element-type** is **string-char**, you get a string.

:initial-value
:initial-element
: Specifies the value to be stored in each element of the new array. If it is not specified, it is **nil** for arrays that can hold arbitrary objects, or 0 or 0.0 for numeric arrays. **:initial-value** is obsolete.

:initial-contents
: Specifies the entire contents for the new array, as a sequence of sequences of sequences... Array element 1 3 4 of a three-dimensional array would be **(elt (elt (elt *initial-contents* 1) 3) 4)**. Recall that a sequence is either a list or a vector, and vectors include strings.

:displaced-to
: If this is not **nil**, a *displaced* array is constructed. If the value is a fixnum or a locative, **make-array** creates a regular displaced array which refers to the specified section of virtual address space. If the value is an array, **make-array** creates an indirect array (see page 167).

:leader-length
: The value should be a fixnum. The array is made with a leader containing that many elements. The elements of the leader are initialized to **nil** unless the **:leader-list** option is given (see below).

:leader-list
: The value should be a list. Call the number of elements in the list *n*. The first *n* elements of the leader are initialized from successive elements of this list. If the **:leader-length** option is not specified, then the length of the leader is *n*. If the **:leader-length** option is given, and its value is greater than *n*, then the *n*th and following leader elements are initialized to **nil**. If its value is less than *n*, an error is signaled. The leader elements are filled in forward order; that is, the **car** of the list is stored in leader element 0, the **cadr** in element 1, and so on.

:fill-pointer The value should be a fixnum. The array is made with a leader containing at least one element, and this fixnum is used to initialize that first element.

Using the :fill-pointer option is equivalent to using :leader-list with a list one element long. It avoids consing the list, and is also compatible with Common Lisp.

:displaced-index-offset
If this is present, the value of the :displaced-to option should be an array, and the value should be a non-negative fixnum; it is made to be the index-offset of the created indirect array. (See page 167.)

:named-structure-symbol
If this is not nil, it is a symbol to be stored in the named-structure cell of the array. The array made is tagged as a named structure (see page 390.) If the array has a leader, then this symbol is stored in leader element 1 regardless of the value of the :leader-list option. If the array does not have a leader, then this symbol is stored in array element zero. Array leader slot 1, or array element 0, cannot be used for anything else in a named structure.

:adjustable-p In strict Common Lisp, a non-nil value for this keyword makes the array *adjustable*, which means that it is permissible to change the array's size with adjust-array (page 176). This is because other Lisp systems have multiple representations for arrays, one which is simple and fast to access, and another which can be adjusted. The Lisp Machine does not require two representations: any array's size may be changed, and this keyword is ignored.

Examples:

```
;; Create a one-dimensional array of five elements.
(make-array 5)
;; Create a two-dimensional array,
;; three by four, with four-bit elements.
(make-array '(3 4) :type 'art-4b)
;; Create an array with a three-element leader.
(make-array 5 :leader-length 3)
;; Create an array containing 5 t's,
;; and a fill pointer saying the array is full.
(make-array 5 :initial-value t :fill-pointer 5)
;; Create a named-structure with five leader
;; elements, initializing some of them.
(setq b (make-array 20 :leader-length 5
                       :leader-list '(0 nil foo)
                       :named-structure-symbol 'bar))
(array-leader b 0) => 0
(array-leader b 1) => bar
(array-leader b 2) => foo
(array-leader b 3) => nil
(array-leader b 4) => nil
```

make-array returns the newly-created array, and also returns, as a second value, the number of words allocated in the process of creating the array, i.e. the **%structure-total-size** of the array.

When make-array was originally implemented, it took its arguments in the following fixed pattern:

```
(make-array area type dimensions
            &optional displaced-to leader
                      displaced-index-offset
                      named-structure-symbol)
```

leader was a combination of the :leader-length and :leader-list options, and the list was in reverse order. This obsolete form is still supported so that old programs will continue to work, but the new keyword-argument form is preferred.

8.4 Accessing Array Elements

aref *array* &rest *subscripts*

Returns the element of *array* selected by the *subscripts*. The *subscripts* must be **fixnums** and their number must match the rank of *array*.

cli:aref *array* &rest *subscripts*

The Common Lisp version of **aref** differs from the traditional one in that it returns a character object rather than an integer when *array* is a string. See chapter 10 for a discussion of the data type of string elements.

aset *x array* &rest *subscripts*

Stores *x* into the element of *array* selected by the *subscripts*. The *subscripts* must be fixnums and their number must match the rank of *array*. The returned value is *x*.

aset is equivalent to

```
(setf (aref array subscripts...) x)
```

aloc *array* &rest *subscripts*

Returns a locative pointer to the element-cell of *array* selected by the *subscripts*. The *subscripts* must be fixnums and their number must match the rank of *array*. The array must not be a numeric array, since locatives to the middle of a numeric array are not allowed. See the explanation of locatives in chapter 14, page 267.

It is equivalent, and preferable, to write

```
(locf (aref array subscripts...))
```

ar-1-force *array i*
as-1-force *value array i*
ap-1-force *array i*

These functions access an array with a single subscript regardless of how many dimensions the array has. They may be useful for manipulating arrays of varying rank, as an alternative to maintaining and updating lists of subscripts or to creating one-dimensional indirect arrays. ar-1-force refers to an element, as-1-force sets an element, and ap-1-force returns a locative to the element's cell.

In using these functions, you must pay attention to the order in which the array elements are actually stored. See section 8.11, page 182.

array-row-major-index *array* &rest *indices*

Calculates the cumulative index in *array* of the element at indices *indices*.

```
(ar-1-force array
  (array-row-major-index array indices...))
```

is equivalent to (aref *array indices...*).

array-leader *array i*

array should be an array with a leader, and *i* should be a fixnum. This returns the *i* th element of *array*'s leader. This is analogous to **aref**.

store-array-leader *x array i*

array should be an array with a leader, and *i* should be a fixnum. *x* may be any object. *x* is stored in the *i* th element of *array*'s leader. store-array-leader returns *x*. This is analogous to **aset**.

It is equivalent, and preferable, to write

```
(setf (array-leader array i) x)
```

ap-leader *array i*

Is equivalent to

```
        (locf (array-leader array i))
```

The following array accessing functions generally need not be used by users.

ar-1 *array i*
ar-2 *array i j*
ar-3 *array i j k*
as-1 *x array i*
as-2 *x array i j*
as-3 *x array i j k*
ap-1 *array i*
ap-2 *array i j*
ap-3 *array i j k*

These are obsolete versions of **aref**, **aset** and **aloc** that only work for one-, two-, or three-dimensional arrays, respectively.

The compiler turns **aref** into **ar-1**, **ar-2**, etc. according to the number of subscripts specified, turns **aset** into **as-1**, **as-2**, etc., and turns **aloc** into **ap-1**, **ap-2**, etc. For arrays with more than three dimensions the compiler uses the slightly less efficient form since the special routines only exist for one, two and three dimensions. There is no reason for any program to call **ar-1**, **as-1**, **ar-2**, etc. explicitly; they are documented because there used to be such a reason, and many old programs use these functions. New programs should use **aref**, **aset**, and **aloc**.

A related function, provided only for Maclisp compatibility, is **arraycall** (page 187).

svref *vector index*

A special accessing function defined by Common Lisp to work only on simple general vectors: vectors with no fill pointer, not displaced, and not adjustable (see page 169). Some other Lisp systems open code **svref** so that it is faster than **aref**, but on the Lisp Machine **svref** is a synonym for **cli:aref**.

bit *bit-vector index*
sbit *bit-vector index*
char *bit-vector index*
schar *bit-vector index*

Special accessing functions defined to work only on bit vectors, only on simple bit vectors, only on strings, and only on simple strings, respectively. On the Lisp Machine they are all synonyms for **cli:aref**.

Here are the conditions signaled for various errors in accessing arrays.

sys:array-has-no-leader (sys:bad-array-mixin error) *Condition*

This is signaled on a reference to the leader of an array that doesn't have one. The condition instance supports the :array operation, which returns the array that was used.

The :new-array proceed-type is provided.

sys:bad-array-mixin *Condition Flavor*
This mixin is used in the conditions signaled by several kinds of problems pertaining to arrays. It defines prompting for the :new-array proceed type.

sys:array-wrong-number-of-dimensions (sys:bad-array-mixin error) *Condition*
This is signaled when an array is referenced (either reading or writing) with the wrong number of subscripts; for example, (aref "foo" 1 2).

The :array operation on the condition instance returns the array that was used. The :subscripts-used operation returns the list of subscripts used.

The :new-array proceed type is provided. It expects one argument, an array to use instead of the original one.

sys:subscript-out-of-bounds (error) *Condition*
This is signaled when there are the right number of subscripts but their values specify an element that falls outside the bounds of the array. The same condition is used by sys:%instance-ref, etc., when the index is out of bounds in the instance.

The condition instance supports the operations :object and :subscripts-used, which return the array or instance and the list of subscripts.

The :new-subscript proceed type is provided. It takes an appropriate number of subscripts as arguments. You should provide as many subscripts as there originally were.

sys:number-array-not-allowed (sys:bad-array-mixin error) *Condition*
This is signaled by an attempt to use **aloc** on a numeric array such as an **art-1b** array or a string. The :array operation and the :new-array proceed type are available.

8.5 Getting Information About an Array

array-type *array*
Returns the symbolic type of *array*.
Example:
```
        (setq a (make-array '(3 5)))
        (array-type a) => art-q
```

array-element-type *array*
Returns a type specifier which describes what elements could be stored in *array* (see section 2.3, page 14 for more about type specifiers). Thus, if *array* is a string, the value is string-char. If *array* is an art-1b array, the value is bit. If *array* is an art-2b array, the value is (mod 4). If array is an art-q array, the value is t (the type which all objects belong to).

array-length *array*
array-total-size *array*

array may be any array. This returns the total number of elements in *array*. For a one-dimensional array, this is one greater than the maximum allowable subscript. (But if fill pointers are being used, you may want to use array-active-length.)
Example:

```
        (array-length (make-array 3)) => 3
        (array-length (make-array '(3 5))) => 15
```

array-total-size is the Common Lisp name of this function.

array-active-length *array*

If *array* does not have a fill pointer, then this returns whatever (array-length *array*) would have. If *array* does have a fill pointer, array-active-length returns it. See the general explanation of the use of fill pointers on page 166.

array-rank *array*

Returns the number of dimensions of *array*.
Example:

```
        (array-rank (make-array '(3 5))) => 2
```

array-dimension *array n*

Returns the length of dimension *n* of *array*. Examples:

```
        (setq a (make-array '(2 3)))
        (array-dimension a 0) => 2
        (array-dimension a 1) => 3
```

array-dimension-n *n array*

array may be any kind of array, and *n* should be a fixnum. If *n* is between 1 and the rank of *array*, this returns the *n* th dimension of *array*. If *n* is 0, this returns the length of the leader of *array*; if *array* has no leader it returns nil. If *n* is any other value, this returns nil.

This function is obsolete; use array-dimension-n, whose calling sequence is cleaner.
Examples:

```
        (setq a (make-array '(3 5) :leader-length 7))
        (array-dimension-n 1 a) => 3
        (array-dimension-n 2 a) => 5
        (array-dimension-n 3 a) => nil
        (array-dimension-n 0 a) => 7
```

array-dimensions *array*

Returns a list whose elements are the dimensions of *array*.
Example:

```
        (setq a (make-array '(3 5)))
        (array-dimensions a) => (3 5)
```

Note: the list returned by (array-dimensions *x*) is equal to the cdr of the list returned by (arraydims *x*).

arraydims *array*

Returns a list whose first element is the symbolic name of the type of *array*, and whose remaining elements are its dimensions. *array* may be any array; it also may be a symbol whose function cell contains an array, for Maclisp compatibility (see section 8.14, page 186).

Example:

```
(setq a (make-array '(3 5)))
(arraydims a) => (art-q 3 5)
```

arraydims is for Maclisp compatibility only.

array-in-bounds-p *array* &rest *subscripts*

t if *subscripts* is a legal set of subscripts for *array*, otherwise nil.

array-displaced-p *array*

t if *array* is any kind of displaced array (including an indirect array), otherwise nil. *array* may be any kind of array.

array-indirect-p *array*

t if *array* is an indirect array, otherwise nil. *array* may be any kind of array.

array-indexed-p *array*

t if *array* is an indirect array with an index-offset, otherwise nil. *array* may be any kind of array.

array-index-offset *array*

Returns the index offset of *array* if it is an indirect array which has an index offset. Otherwise it returns nil. *array* may be any kind of array.

array-has-fill-pointer-p *array*

t if array has a fill pointer. It must have a leader and leader element 0 must be an integer. While array leaders are not standard Common Lisp, fill pointers are, and so is this function.

array-has-leader-p *array*

t if *array* has a leader, otherwise nil.

array-leader-length *array*

Returns the length of *array*'s leader if it has one, or nil if it does not.

adjustable-array-p *array*

According to Common Lisp, returns t if *array*'s size may be adjusted with adjust-array (see below). On the Lisp Machine, this function always returns t.

8.6 Changing the Size of an Array

adjust-array *array new-dimensions* &key *element-type initial-element initial-contents fill-pointer displaced-to displaced-index-offset*

Modifies various aspects of an array. *array* is modified in place if that is possible; otherwise, a new array is created and *array* is forwarded to it. In either case, *array* is returned. The arguments have the same names as arguments to make-array, and signify approximately the same thing. However:

element-type is just an error check. adjust-array cannot change the array type. If the array type of *array* is not what *element-type* would imply, you get an error.

If *displaced-to* is specified, the new array is displaced as specified by *displaced-to* and *displaced-index-offset*. If *array* itself was already displaced, it is modified in place provided that either *array* used to have an index offset and is supposed to continue to have one, or *array* had no index offset and is not supposed to have one.

Otherwise, if *initial-contents* was specified, it is used to set all the contents of the array. The old contents of *array* are irrelevant.

Otherwise, each element of *array* is copied forward into the new array to the slot with the same indices, if there is one. Any new slots whose indices were out of range in *array* are initialized to *initial-element*, or to nil or 0 if *initial-element* was not specified.

fill-pointer, if specified, is used to set the fill pointer of the array. Aside from this, the result has a leader with the same contents as the original *array*.

adjust-array is the only function in this section which is standard Common Lisp. According to Common Lisp, an array's dimensions can be adjusted only if the :adjustable option was specified to make-array with a non-nil value when the array was created (see page 169). The Lisp Machine does not distinguish adjustable and nonadjustable arrays; any array may be adjusted.

adjust-array-size *array new-size*

If *array* is a one-dimensional array, its size is changed to be *new-size*. If *array* has more than one dimension, its size (array-length) is changed to *new-size* by changing only the last dimension.

If *array* is made smaller, the extra elements are lost; if *array* is made bigger, the new elements are initialized in the same fashion as make-array (see page 167) would initialize them: either to nil or 0, depending on the type of array.

Example:

```
(setq a (make-array 5))
(aset 'foo a 4)
(aref a 4) => foo
(adjust-array-size a 2)
(aref a 4) => an error occurs
```

If the size of the array is being increased, **adjust-array-size** may have to allocate a new array somewhere. In that case, it alters *array* so that references to it will be made to the new array instead, by means of invisible pointers (see **structure-forward**, page 273). **adjust-array-size** returns the new array if it creates one, and otherwise it returns *array*. Be careful to be consistent about using the returned result of **adjust-array-size**, because you may end up holding two arrays which are not the same (i.e. not **eq**), but which share the same contents.

array-grow *array* &rest *dimensions*
Equivalent to (**adjust-array** *array dimensions*). This name is obsolete.

si:change-indirect-array *array type dimlist displaced-p index-offset*
Changes an indirect array *array*'s type, size, or target pointed at. *type* specifies the new array type, *dimlist* its new dimensions, *displaced-p* the target it should point to (an array, locative or fixnum), and *index-offset* the new offset in the new target.

array is returned.

8.7 Arrays Overlaid With Lists

These functions manipulate **art-q-list** arrays, which were introduced on page 163.

g-l-p *array*
array should be an **art-q-list** array. This returns a list which shares the storage of *array*. Example:

```
(setq a (make-array 4 :type 'art-q-list))
(aref a 0) => nil
(setq b (g-l-p a)) => (nil nil nil nil)
(rplaca b t)
b => (t nil nil nil)
(aref a 0) => t
(aset 30 a 2)
b => (t nil 30 nil)
```

g-l-p stands for 'get list pointer'.

The following two functions work strangely, in the same way that **store** does, and should not be used in new programs.

get-list-pointer-into-array *array-ref*
The argument *array-ref* is ignored, but should be a reference to an **art-q-list** array by applying the array to subscripts (rather than by **aref**). This returns a list object which is a portion of the "list" of the array, beginning with the last element of the last array which has been called as a function.

get-locative-pointer-into-array *array-ref*

get-locative-pointer-into-array is similar to get-list-pointer-into-array, except that it returns a locative, and doesn't require the array to be art-q-list. Use locf of aref in new programs.

8.8 Adding to the End of an Array

vector-push *x array*
array-push *array x*

array must be a one-dimensional array which has a fill pointer and *x* may be any object. vector-push attempts to store *x* in the element of the array designated by the fill pointer, and increase the fill pointer by one. If the fill pointer does not designate an element of the array (specifically, when it gets too big), it is unaffected and vector-push returns nil; otherwise, the two actions (storing and incrementing) happen uninterruptibly, and vector-push returns the *former* value of the fill pointer, i.e. the array index in which it stored *x*. If the array is of type art-q-list, an operation similar to nconc has taken place, in that the element has been added to the list by changing the cdr of the formerly last element. The cdr-coding is updated to ensure this.

array-push is an old name for this function. vector-push is preferable because it takes arguments in an order like push.

vector-push-extend *x array* &optional *extension*
array-push-extend *array x* &optional *extension*

vector-push-extend is just like vector-push except that if the fill pointer gets too large, the array grows to fit the new element; it never "fails" the way vector-push does, and so never returns nil. *extension* is the number of elements to be added to the array if it needs to grow. It defaults to something reasonable, based on the size of the array.

array-push-extend differs only in the order of arguments,

vector-pop *array*
array-pop *array*

array must be a one-dimensional array which has a fill pointer. The fill pointer is decreased by one and the array element designated by the new value of the fill pointer is returned. If the new value does not designate any element of the array (specifically, if it had already reached zero), an error is caused. The two operations (decrementing and array referencing) happen uninterruptibly. If the array is of type art-q-list, an operation similar to nbutlast has taken place. The cdr-coding is updated to ensure this.

The two names are synonymous.

sys:fill-pointer-not-fixnum (sys:bad-array-mixin error) *Condition*

This is signaled when one of the functions in this section is used with an array whose leader element zero is not a fixnum. Most other array accessing operations simply assume that the array has no fill pointer in such a case, but these cannot be performed without a fill pointer.

The :array operation on the condition instance returns the array that was used. The :new-array proceed type is supported, with one argument, an array.

8.9 Copying an Array

The new functions **replace** (page 189) and **fill** (page 190) are useful ways to copy parts of arrays.

array-initialize *array value* &optional *start end*

Stores *value* into all or part of *array*. *start* and *end* are optional indices which delimit the part of *array* to be initialized. They default to the beginning and end of the array.

This function is by far the fastest way to do the job.

fillarray *array x*

array may be any type of array, or, for Maclisp compatibility, a symbol whose function cell contains an array. It can also be nil, in which case an array of type art-q is created. There are two forms of this function, depending on the type of *x*.

If *x* is a list, then fillarray fills up *array* with the elements of *list*. If *x* is too short to fill up all of *array*, then the last element of *x* is used to fill the remaining elements of *array*. If *x* is too long, the extra elements are ignored. If *x* is nil (the empty list), *array* is filled with the default initial value for its array type (nil or 0).

If *x* is an array (or, for Maclisp compatibility, a symbol whose function cell contains an array), then the elements of *array* are filled up from the elements of *x*. If *x* is too small, then the extra elements of *array* are not affected.

If *array* is multi-dimensional, the elements are accessed in row-major order: the last subscript varies the most quickly. The same is true of *x* if it is an array.

fillarray returns *array*; or, if *array* was nil, the newly created array.

listarray *array* &optional *limit*

array may be any type of array, or, for Maclisp compatibility, a symbol whose function cell contains an array. listarray creates and returns a list whose elements are those of *array*. If *limit* is present, it should be a fixnum, and only the first *limit* (if there are more than that many) elements of *array* are used, and so the maximum length of the returned list is *limit*.

If *array* is multi-dimensional, the elements are accessed in row-major order: the last subscript varies the most quickly.

list-array-leader *array* &optional *limit*

array may be any type of array, or, for Maclisp compatibility, a symbol whose function cell contains an array. list-array-leader creates and returns a list whose elements are those of *array*'s leader. If *limit* is present, it should be a fixnum, and only the first *limit* (if there are more than that many) elements of *array*'s leader are used, and so the maximum length of the returned list is *limit*. If *array* has no leader, nil is returned.

copy-array-contents *from to*

from and *to* must be arrays. The contents of *from* is copied into the contents of *to*, element by element. If *to* is shorter than *from*, the rest of *from* is ignored. If *from* is shorter than *to*, the rest of *to* is filled with **nil**, 0 or 0.0 according to the type of array. This function always returns **t**.

The entire length of *from* or *to* is used, ignoring the fill pointers if any. The leader itself is not copied.

copy-array-contents works on multi-dimensional arrays. *from* and *to* are linearized subscripts, and elements are taken in row-major order.

copy-array-contents-and-leader *from to*

Like **copy-array-contents**, but also copies the leader of *from* (if any) into *to*.

copy-array-portion *from-array from-start from-end to-array to-start to-end*

The portion of the array *from-array* with indices greater than or equal to *from-start* and less than *from-end* is copied into the portion of the array *to-array* with indices greater than or equal to *to-start* and less than *to-end*, element by element. If there are more elements in the selected portion of *to-array* than in the selected portion of *from-array*, the extra elements are filled with the default value as by **copy-array-contents**. If there are more elements in the selected portion of *from-array*, the extra ones are ignored. Multi-dimensional arrays are treated the same way as **copy-array-contents** treats them. This function always returns **t**.

%blt and **%blt-typed** (page 280) are often useful for copying parts of arrays. They can be used to shift a part of an array either up or down.

8.10 Bit Array Functions

These functions perform bitwise boolean operations on the elements of arrays.

bit-and *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-ior *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-xor *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-eqv *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-nand *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-nor *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-andc1 *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-andc2 *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-orc1 *bit-array-1 bit-array-2* &optional *result-bit-array*
bit-orc2 *bit-array-1 bit-array-2* &optional *result-bit-array*

Perform boolean operations element by element on bit arrays. The arguments must match in their size and shape, and all of their elements must be integers. Corresponding elements of *bit-array-1* and *bit-array-2* are taken and passed to one of **logand**, **logior**, etc. to get an element of the result array.

If the third argument is non-nil, the result bits are stored into it, modifying it destructively. If it is t, the results are stored in *bit-array-1*. Otherwise a new array of the same type as *bit-array-1* is created and used for the result. In any case, the value returned is the array where the results are stored.

These functions were introduced for the sake of Common Lisp, which defines them only when all arguments are specialized arrays that hold only zero or one. In the Lisp machine, they accept not only such arrays (art-1b arrays) but any arrays whose elements are integers.

bit-not *bit-array* &optional *result-bit-array*

Performs lognot on each element of *bit-array* to get an element of the result. If *result-bit-array* is non-nil, the result elements are stored in that; it must match *bit-array* in size and shape. Otherwise, a new array of the same type as *bit-array* is created and used to hold the result. The value of bit-not is the array where the results are stored.

bitblt *alu width height from-array from-x from-y to-array to-x to-y*

from-array and *to-array* must be two-dimensional arrays of bits or bytes (art-1b, art-2b, art-4b, art-8b, art-16b, or art-32b). bitblt copies a rectangular portion of *from-array* into a rectangular portion of *to-array*. The value stored can be a Boolean function of the new value and the value already there, under the control of *alu* (see below). This function is most commonly used in connection with raster images for TV displays.

The top-left corner of the source rectangle is (ar-2-reverse *from-array from-x from-y*). The top-left corner of the destination rectangle is (ar-2-reverse *to-array to-x to-y*). *width* and *height* are the dimensions of both rectangles. If *width* or *height* is zero, bitblt does nothing. The *x* coordinates and *width* are used as the second dimension of the array, since the horizontal index is the one which varies fastest in the screen buffer memory and the array's last index varies fastest in row-major order.

from-array and *to-array* are allowed to be the same array. bitblt normally traverses the arrays in increasing order of *x* and *y* subscripts. If *width* is negative, then (abs *width*) is used as the width, but the processing of the *x* direction is done backwards, starting with the highest value of *x* and working down. If *height* is negative it is treated analogously. When bitblt'ing an array to itself, when the two rectangles overlap, it may be necessary to work backwards to achieve effects such as shifting the entire array downwards by a certain number of rows. Note that negativity of *width* or *height* does not affect the (*x*, *y*) coordinates specified by the arguments, which are still the top-left corner even if bitblt starts at some other corner.

If the two arrays are of different types, bitblt works bit-wise and not element-wise. That is, if you bitblt from an art-2b array into an art-4b array, then two elements of the *from-array* correspond to one element of the *to-array*.

If bitblt goes outside the bounds of the source array, it wraps around. This allows such operations as the replication of a small stipple pattern through a large array. If bitblt goes outside the bounds of the destination array, it signals an error.

If *src* is an element of the source rectangle, and *dst* is the corresponding element of the destination rectangle, then **bitblt** changes the value of *dst* to (**boole** *alu src dst*). See the **boole** function (page 152). There are symbolic names for some of the most useful *alu* functions; they are **tv:alu-seta** (plain copy), **tv:alu-ior** (inclusive or), **tv:alu-xor** (exclusive or), and **tv:alu-andca** (and with complement of source).

bitblt is written in highly-optimized microcode and goes very much faster than the same thing written with ordinary **aref** and **aset** operations would. Unfortunately this causes **bitblt** to have a couple of strange restrictions. Wrap-around does not work correctly if *from-array* is an indirect array with an index-offset. **bitblt** signals an error if the second dimensions of *from-array* and *to-array* are not both integral multiples of the machine word length. For **art-1b** arrays, the second dimension must be a multiple of 32., for **art-2b** arrays it must be a multiple of 16., etc.

8.11 Order of Array Elements

Currently, multi-dimensional arrays are stored in row-major order, as in Maclisp., and as specified by Common Lisp. This means that successive memory locations differ in the last subscript. In older versions of the system, arrays were stored in column-major order.

Most user code has no need to know about which order array elements are stored in. There are three known reasons to care: use of multidimensional indirect arrays; paging efficiency (if you want to reference every element in a multi-dimensional array and move linearly through memory to improve locality of reference, you must vary the last subscript fastest in row-major order); and access to the TV screen or to arrays of pixels copied to or from the screen with **bitblt**. The latter is the most important one.

The bits on the screen are actually stored in rows, which means that the dimension that varies fastest has to be the horizontal position. As a result, if arrays are stored in row-major order, the horizontal position must be the second subscript, but if arrays are stored in column-major order, the horizontal position must be the first subscript. To ease the conversion of code that uses arrays of pixels, several bridging functions are provided:

make-pixel-array *width height* &rest *options*
This is like **make-array** except that the dimensions of the array are *width* and *height*, in whichever order is correct. *width* is used as the dimension in the subscript that varies fastest in memory, and *height* as the other dimension. *options* are passed along to **make-array** to specify everything but the size of the array.

pixel-array-width *array*
Returns the extent of *array*, a two-dimensional array, in the dimension that varies faster through memory. For a screen array, this is always the width.

pixel-array-height *array*
Returns the extent of *array*, a two-dimensional array, in the dimension that varies slower through memory. For a screen array, this is always the height.

ar-2-reverse *array horizontal-index vertical-index*
Returns the element of *array* at *horizontal-index* and *vertical-index*. *horizontal-index* is used as the subscript in whichever dimension varies faster through memory.

as-2-reverse *newvalue array horizontal-index vertical-index*
Stores *newvalue* into the element of *array* at *horizontal-index* and *vertical-index*. *horizontal-index* is used as the subscript in whichever dimension varies faster through memory.

Code that was written before the change in order of array indices can be converted by replacing calls to make-array, array-dimension, aref and aset with these functions. It can then work either in old systems or in new ones. In more complicated circumstances, you can facilitate conversion by writing code which tests this variable.

sys:array-index-order *Constant*
This is t in more recent system versions which store arrays in row-major order (last subscript varies fastest). It is nil in older system versions which store arrays in column-major order.

8.12 Matrices and Systems of Linear Equations

The functions in this section perform some useful matrix operations. The matrices are represented as two-dimensional Lisp arrays. These functions are part of the mathematics package rather than the kernel array system, hence the 'math:' in the names.

math:multiply-matrices *matrix-1 matrix-2* &optional *matrix-3*
Multiplies *matrix-1* by *matrix-2*. If *matrix-3* is supplied, multiply-matrices stores the results into *matrix-3* and returns *matrix-3*, which should be of exactly the right dimensions for containing the result of the multiplication; otherwise it creates an array to contain the answer and returns that. All matrices must be either one- or two-dimensional arrays, and the first dimension of *matrix-2* must equal the second dimension of *matrix-1*.

math:invert-matrix *matrix* &optional *into-matrix*
Computes the inverse of *matrix*. If *into-matrix* is supplied, stores the result into it and returns it; otherwise it creates an array to hold the result and returns that. *matrix* must be two-dimensional and square. The Gauss-Jordan algorithm with partial pivoting is used. Note: if you want to solve a set of simultaneous equations, you should not use this function; use math:decompose and math:solve (see below).

math:transpose-matrix *matrix* &optional *into-matrix*
Transposes *matrix*. If *into-matrix* is supplied, stores the result into it and returns it; otherwise it creates an array to hold the result and returns that. *matrix* must be a two-dimensional array. *into-matrix*, if provided, must be two-dimensional and have exactly the right dimensions to hold the transpose of *matrix*.

math:determinant *matrix*
Returns the determinant of *matrix*. *matrix* must be a two-dimensional square matrix.

The next two functions are used to solve sets of simultaneous linear equations. math:decompose takes a matrix holding the coefficients of the equations and produces the LU decomposition; this decomposition can then be passed to math:solve along with a vector of right-hand sides to get the values of the variables. If you want to solve the same equations for many different sets of right-hand side values, you only need to call math:decompose once. In terms of the argument names used below, these two functions exist to solve the vector equation $A\ x = b$ for x. A is a matrix. b and x are vectors.

math:decompose *a* &optional *lu ps*
Computes the LU decomposition of matrix *a*. If *lu* is non-nil, stores the result into it and returns it; otherwise it creates an array to hold the result, and returns that. The lower triangle of *lu*, with ones added along the diagonal, is L, and the upper triangle of *lu* is U, such that the product of L and U is *a*. Gaussian elimination with partial pivoting is used. The *lu* array is permuted by rows according to the permutation array *ps*, which is also produced by this function; if the argument *ps* is supplied, the permutation array is stored into it; otherwise, an array is created to hold it. This function returns two values, the LU decomposition and the permutation array.

math:solve *lu ps b* &optional *x*
This function takes the LU decomposition and associated permutation array produced by math:decompose and solves the set of simultaneous equations defined by the original matrix *a* given to math:decompose and the right-hand sides in the vector *b*. If *x* is supplied, the solutions are stored into it and it is returned; otherwise an array is created to hold the solutions and that is returned. *b* must be a one-dimensional array.

math:list-2d-array *array*
Returns a list of lists containing the values in *array*, which must be a two-dimensional array. There is one element for each row; each element is a list of the values in that row.

math:fill-2d-array *array list*
This is the opposite of math:list-2d-array. *list* should be a list of lists, with each element being a list corresponding to a row. *array*'s elements are stored from the list. Unlike fillarray (see page 179), if *list* is not long enough, math:fill-2d-array "wraps around", starting over at the beginning. The lists which are elements of *list* also work this way.

math:singular-matrix (sys:arithmetic-error error) *Condition*
This is signaled when any of the matrix manipulation functions in this section has trouble because of a singular matrix. (In some functions, such as math:determinant, a singular matrix is not a problem.)

The :matrix operation on the condition instance returns the matrix which is singular.

8.13 Planes

A *plane* is effectively an array whose bounds, in each dimension, are plus-infinity and minus-infinity; all integers are legal as indices. Planes may be of any rank. When you create a plane, you do not need to specify any size, just the rank. You also specify a default value. At that moment, every component of the plane has that value. As you can't ever change more than a finite number of components, only a finite region of the plane need actually be stored. When you refer to an element for which space has not actually been allocated, you just get the default value.

The regular array accessing functions don't work on planes. You can use make-plane to create a plane, plane-aref or plane-ref to get the value of a component, and plane-aset or plane-store to store into a component. array-rank works on planes.

A plane is actually stored as an array with a leader. The array corresponds to a rectangular, aligned region of the plane, containing all the components in which a plane-store has been done (and, usually, others which have never been altered). The lowest-coordinate corner of that rectangular region is given by the plane-origin in the array leader. The highest-coordinate corner can be found by adding the plane-origin to the array-dimensions of the array. The plane-default is the contents of all the elements of the plane that are not actually stored in the array. The plane-extension is the amount to extend a plane by in any direction when the plane needs to be extended. The default is 32.

If you never use any negative indices, then the plane-origin remains all zeroes and you can use regular array functions, such as aref and aset, to access the portion of the plane that is actually stored. This can be useful to speed up certain algorithms. In this case you can even use the bitblt function on a two-dimensional plane of bits or bytes, provided you don't change the plane-extension to a number that is not a multiple of 32.

make-plane *rank* &key *type default-value extension initial-dimensions initial-origins*

Creates and returns a plane. *rank* is the number of dimensions. The keyword arguments are

type	The array type symbol (e.g. art-1b) specifying the type of the array out of which the plane is made.
default-value	The default component value as explained above.
extension	The amount by which to extend the plane, as explained above.
initial-dimensions	nil or a list of integers whose length is *rank*. If not nil, each element corresponds to one dimension, specifying the width to allocate the array initially in that dimension.
initial-origins	nil or a list of integers whose length is *rank*. If not nil, each element corresponds to one dimension, specifying the smallest index in that dimension for which storage should initially be allocated.

Example:

```
        (make-plane 2 :type 'art-4b :default-value 3)
```

creates a two-dimensional plane of type art-4b, with default value 3.

plane-origin *plane*
A list of numbers, giving the lowest coordinate values actually stored.

plane-default *plane*
This is the contents of the infinite number of plane elements that are not actually stored.

plane-extension *plane*
The amount to extend the plane by, in any direction, when plane-store is done outside of the currently-stored portion.

plane-aref *plane* &rest *subscripts*
plane-ref *plane subscripts*
These two functions return the contents of a specified element of a plane. They differ only in the way they take their arguments; plane-aref wants the subscripts as arguments, while plane-ref wants a list of subscripts.

plane-aset *datum plane* &rest *subscripts*
plane-store *datum plane subscripts*
These two functions store *datum* into the specified element of a plane, extending it if necessary, and return *datum*. They differ only in the way they take their arguments; plane-aset wants the subscripts as arguments, while plane-store wants a list of subscripts.

8.14 Maclisp Array Compatibility

The functions in this section are provided only for Maclisp compatibility and should not be used in new programs.

Fixnum arrays do not exist (however, see Zetalisp's small-positive-integer arrays). Float arrays exist but you do not use them in the same way; no declarations are required or allowed. Ungarbage-collected arrays do not exist. Readtables and obarrays are represented as arrays, but Zetalisp does not use special array types for them. See the descriptions of read (page 531) and intern (page 645) for information about readtables and obarrays (packages). There are no 'dead" arrays, nor are Multics "external" arrays provided.

The arraycall function exists for compatibility but should not be used (see aref, page 170.)

Subscripts are always checked for validity, regardless of the value of *rset and whether the code is compiled or not. However, in a multi-dimensional array, an error is caused only if the subscripts would have resulted in a reference to storage outside of the array. For example, if you have a 2 by 7 array and refer to an element with subscripts 3 and 1, no error occurs despite the fact that the reference is invalid; but if you refer to element 1 by 100, an error occurs. In other words, subscript errors are caught if and only if they refer to storage outside the array; some errors are undetected, but they can only clobber (alter randomly) some other element of the same array, not something completely unpredictable.

loadarrays and dumparrays are not provided. However, arrays can be put into QFASL files; see section 17.8, page 317.

The *rearray function is not provided, since not all of its functionality is available in Zetalisp. Its most common uses are implemented by adjust-array-size.

In Maclisp, arrays are usually kept on the array property of symbols, and the symbols are used instead of the arrays. In order to provide some degree of compatibility for this manner of using arrays, the array, *array, and store functions are provided, and when arrays are applied to arguments, the arguments are treated as subscripts and apply returns the corresponding element of the array.

array &quote *symbol type* &eval &rest *dims*
Creates an art-q type array in default-array-area with the given dimensions. (That is, *dims* is given to make-array as its first argument.) *type* is ignored. If *symbol* is nil, the array is returned; otherwise, the array is put in the function cell of *symbol*, and *symbol* is returned.

***array** *symbol type* &rest *dims*
Is like array, except that all of the arguments are evaluated.

store *array-ref x* *Special form*
Stores *x* into the specified array element. *array-ref* should be a form which references an array by calling it as a function (aref forms are not acceptable). First *x* is evaluated, then *array-ref* is evaluated, and then the value of *x* is stored into the array cell last referenced by a function call, presumably the one in *array-ref*.

xstore *x array-ref*
This is just like store, but it is not a special form; this is because the arguments are in the other order. This function only exists for the compiler to compile the store special form into, and should never be used by programs.

arraycall *ignored array* &rest *subscripts*
(arraycall t *array sub1 sub2*...) is the same as (aref *array sub1 sub2*...). It exists for Maclisp compatibility.

9. Generic Sequence Functions

The type specifier **sequence** is defined to include lists and vectors (arrays of rank one). Lists and vectors are similar in that both can be regarded as sequences of elements: there is a first element, a second element, and so on. Element *n* of a list is **(nth** *n list***)**, and element *n* of a vector is **(aref** *vector n***)**. Many useful operations which apply in principle to a sequence of objects can work equally well on lists and vectors. These are the generic sequence functions.

All the generic sequence functions accept **nil** as a sequence of length zero.

9.1 Primitive Sequence Operations

make-sequence *type size* &key *initial-element*

Creates a sequence of type *type*, *size* elements long. *size* must be an integer and *type* must be either **list** or some kind of array type. *type* could be just **array** or **vector** to make a general vector, it could be **(vector (byte 8))** to make an **art-8b** vector, and so on.

If *initial-element* is specified, each element of the new sequence contains *initial-element*. Otherwise, the new sequence is initialized to contain **nil** if that is possible, zero otherwise (for numeric array types).

```
(make-sequence 'list 3)
  =>  (nil nil nil)

(make-sequence 'array 5 :initial-element t)
  =>  #(t t t t t)

(make-sequence '(vector bit) 5)
  =>  #*00000
```

elt *sequence index*

Returns the element at index *index* in *sequence*. If *sequence* is a list, this is **(nth** *index sequence***)**. If *sequence* is a vector, this is **(aref** *index sequence***)**. Being microcoded, **elt** is as fast as either **nth** or **aref**.

(setf (elt *sequence index***)** *value***)** is the way to set an element of a sequence.

length *sequence*

Returns the length of *sequence*, as an integer. For a vector with a fill pointer, this is the fill pointer value. For a list, it is the traditional Lisp function; note that lists ending with atoms other than **nil** are accepted, so that the length of **(a b . c)** is 2.

9.2 Simple Sequence Operations

copy-seq *sequence*

Returns a new sequence of the same type, length and contents as *sequence*.

concatenate *result-type* &rest *sequences*

Returns a new sequence, of type *result-type*, whose contents are made from the contents of all the *sequences*. *result-type* can be **list** or any array type, just as in **make-sequence** above. Examples:

```
(concatenate 'list '(1 2) '#(A 3))  =>  (1 2 A 3)
(concatenate 'vector '(1 2) '#(A 3)  =>  #(1 2 A 3)
```

subseq *sequence start* &optional *end*

Returns a new sequence whose elements are a subsequence of *sequence*. The new sequence is of the same type as *sequence*.

start is the index of the first element of *sequence* to take. *end* is the index of where to stop—the first element *not* to take. *end* can also be **nil**, meaning take everything from *start* up to the end of *sequence*.

Examples:

```
(subseq "Foobar" 3 5)  =>  "ba"
(subseq '(a b c) 1)  =>  (b c)
```

It is also possible to **setf** a call to **subseq**. This means to store into part of the sequence passed to **subseq**. Thus,

```
(setf (subseq "Foobar" 3 5) "le")
```

modifies the string "Foobar" so that it contains "Fooler" instead.

replace *into-sequence-1 from-sequence-2* &key (*start1* 0) *end1* (*start2* 0) *end2*

Copies part of *from-sequence-2* into part of *to-sequence-1*. *start2* and *end2* are the indices of the part of *from-sequence-2* to copy from, and *start1* and *end1* are the indices of the part of *to-sequence-1* to copy into.

If the number of elements to copy out of *from-sequence-2* is less than the number of elements of *to-sequence-1* to be copied into, the extra elements of *to-sequence-1* are not changed. If the number of elements to copy out is more than there is room for, the last extra elements are ignored.

If the two sequence arguments are the same sequence, then the elements to be copied are copied first into a temporary sequence (if necessary) to make sure that no element is overwritten before it is copied. Example:

```
(setq str "Elbow")
(replace str str :start1 2 :end1 5 :start2 1 :end2 4)
```

modifies **str** to contain "Ellbo".

into-sequence-1 is returned as the value of **replace**.

fill *sequence item* &key (*start* 0) *end*

Modifies the contents of *sequence* by setting all the elements to *item*. *start* and *end* may be specified to limit the operation to some contiguous portion of *sequence*; then the elements before *start* or after *end* are unchanged. If *end* is nil, the filling goes to the end of *sequence*.

The value returned by fill is *sequence*. Example:

```
(setq l '(a b c d e))
(fill l 'lose :start 2)

l => (a b lose lose lose)
```

reverse *sequence*

Returns a new sequence containing the same elements as *sequence* but in reverse order. The new sequence is of the same type and length as *sequence*. reverse does not modify its argument, unlike nreverse which is faster but does modify its argument. The list created by reverse is not cdr-coded.

```
(reverse "foo")  =>  "oof"
(reverse '(a b (c d) e)) => (e (c d) b a)
```

nreverse *sequence*

Modifies *sequence* destructively to have its elements in reverse order, and returns *sequence* as modified. For a vector, this is done by copying the elements to different positions. For a list, this is done by modifying cdr pointers. This has two important consequences: it is most efficient when the list is not cdr-coded, and the rearranged list starts with the cell that used to be at the end. Although the altered list as a whole contains the same cells as the original, the actual value of the altered list is not eq to the original list. For this reason, one must always store the value of nreverse into the place where the list will be used. Do not just use nreverse for effect on a list.

```
(setq a '#(1 2 3 4 5))
(nreverse a)
(concatenate 'list a) => (5 4 3 2 1)

(setq b '(1 2 3 4 5)
      c b
      d (last b))
(setq b (nreverse b))

b => (5 4 3 2 1)
c => (1)
(eq b d)  =>  t
```

nreverse is most frequently used after a loop which computes elements for a new list one by one. These elements can be put on the new list with push, but this produces a list which has the elements in reverse order (first one generated at the end of the list).

```
(let (accumulate)
  (dolist (x input)
    (push (car x) accumulate)
    (push (cdr x) accumulate))
  (nreverse accumulate))
```

Currently, **nreverse** is inefficient with cdr-coded lists (see section 5.4, page 100), because it just uses **rplacd** in the straightforward way. This may be fixed someday. In the meantime **reverse** might be preferable in some cases.

9.3 Mapping On Sequences

cli:map *result-type function* &rest *sequences*

The Common Lisp **map** function maps *function* over successive elements of each *sequence*, constructing and returning a sequence of the results that *function* returns. The constructed sequence is of type *result-type* (see **make-sequence**, page 188).

function is called first on the first elements of all the sequences, then on the second elements of all, and so on until some argument sequence is exhausted.

```
(map 'list 'list '(1 2 3) '#(A B C D))
  =>  ((1 A) (2 B) (3 C))

(setq vect (map '(vector (mod 16.)) '+
                '(3 4 5 6 7) (circular-list 1)))
(concatenate 'list vect)  =>  (2 3 4 5 6)
(array-element-type vect)  =>  (mod 16.)
```

result-type can also be **nil**. Then the values returned by *function* are thrown away, no sequence is constructed, and **map** returns **nil**.

This function is available under the name **map** in Common Lisp programs. In traditional Zetalisp programs, **map** is another function which does something related but different; see page 84. Traditional programs can call this function as **cli:map**.

cli:some *predicate* &rest *sequences*

Applies *predicate* to successive elements of each sequence. If *predicate* ever returns a non-nil value, **cli:some** immediately returns the same value. If one of the argument sequences is exhausted, **cli:some** returns **nil**.

Each time *predicate* is called, it receives one argument from each sequence. The first time, it gets the first element of each sequence, then the second element of each, and so on until a sequence is exhausted. Examples:

```
(cli:some 'plusp '(-4 0 5 6))  =>  5
(cli:some '> '(-4 0 5 6) '(0 12 12 12))  =>  nil
(cli:some '> '(-4 0 5 6) '(3 3 3 3))  =>  5
(cli:some '> '(-4 0 5 6) '(3 3))  =>  nil
```

This function is available under the name some in Common Lisp programs. In traditional Zetalisp programs, some is another function which does something related but different; see page 106. Traditional programs can call this function as cli:some.

cli:every *predicate* &rest *sequences*

Applies *predicate* to successive elements of each sequence. If *predicate* ever returns nil, cli:every immediately returns nil. If one of the argument sequences is exhausted, cli:every returns t.

Each time *predicate* is called, it receives one argument from each sequence. The first time, it gets the first element of each sequence, then the second element of each, and so on until a sequence is exhausted. Examples:

```
(cli:every 'plusp '(-4 0 5 6))  =>  nil
(cli:every 'plusp '(5 6))  =>  t
```

This function is available under the name every in Common Lisp programs. In traditional Zetalisp programs, every is another function which does something related but different; see page 106. Traditional programs can call this function as cli:every.

notany *predicate* &rest *sequences*
notevery *predicate* &rest *sequences*

These are the opposites of cli:some and cli:every.

(notany ...) is equivalent to (not (cli:some ...)).
(notevery ...) is equivalent to (not (cli:every ...)).

reduce *function sequence* &key *from-end* (*start* 0) *end initial-value*

Combines the elements of *sequence* using *function*, a function of two args. *function* is applied to the first two elements; then to that result and the third element; then to that result and the fourth element; and so on.

start and *end* are indices that restrict the action to a part of *sequence*, as if the rest of *sequence* were not there. They default to 0 and nil (nil for *end* means go all the way to the end of *sequence*).

If *from-end* is non-nil, processing starts with the last of the elements. *function* is first applied to the last two elements; then to the previous element and that result; then to the previous element and that result; and so on until element number *start* has been used.

If *initial-value* is specified, it acts like an extra element of *sequence*, used in addition to the actual elements of the specified part of *sequence*. It comes, in effect, at the beginning if *from-end* is nil, but at the end if *from-end* is non-nil, so that in any case it is the first element to be processed.

If there is only one element to be processed, that element is returned and *function* is not called.

If there are no elements (*sequence* is of length zero and no *initial-value*), *function* is called with no arguments and its value is returned.

Examples:

```
(reduce '+ '(1 2 3))  =>  6
(reduce '- '(1 2 3))  =>  -4
(reduce '- '(1 2 3) :from-end t)  =>  2  ;; 1 - (2 - 3)
(reduce 'cons '(1 2 3) :from-end t)  =>  (1 2 . 3)
(reduce 'cons '(1 2 3))  =>  ((1 . 2) . 3)
```

9.4 Operating on Selected Elements

The generic sequence functions for searching, substituting and removing elements from sequences take similar arguments whose meanings are standard. This is because they all look at each element of the sequence to decide whether it should be processed.

Functions which conceptually modify the sequence come in pairs. One function in the pair copies the sequence if necessary and never modifies the argument. The copy is a list if the original sequence is a list; otherwise, the copy is an art-q array. If the sequence is a list, it may be copied only partially, sharing any unchanged tail with the original argument. If no elements match, the result sequence may be eq to the argument sequence.

The other function in the pair may alter the original sequence and return it, or may make a copy and return that.

There are two ways the function can decide which elements to operate on. The functions whose names end in -if or -if-not have an argument named *predicate* which should be a function of one argument. This function is applied to each element and the value determines whether the element is processed.

The other functions have an argument named *item* or something similar which is an object to compare each element with. The elements that match *item* are processed. By default, the comparison is done with eql. You can specify any function of two arguments to be used instead, as the *test* keyword argument. *item* is always the first argument, and an element of the sequence is the second argument. The element matches *item* if *test* returns non-nil. Alternatively, you can specify the *test-not* keyword argument; then the element matches if *test-not* returns nil.

The elements may be tested in any order, and may be tested more than once. For predictable results, your *predicate*, *test* and *test-not* functions should be side-effect free.

The five keyword arguments *start*, *end*, *key*, *count* and *from-end* have the same meanings for all of the functions, except that *count* is not relevant for some kinds of operations. Here is what they do:

start, end — *start* and *end* are indices in the sequence; they restrict the processing to the portion between those indices. Only elements in this portion are tested, replaced or removed. For the search functions, only this portion is searched. For element removal functions, elements outside the portion are unchanged.

start is the index of the first element to be processed, and *end* is the index of the element after the last element to be processed. *end* can also be **nil**, meaning that processing should continue to the end of the sequence.

start always defaults to 0, and *end* always defaults to **nil**.

key — *key*, if not **nil**, is a function of one argument which is applied to each element of the sequence to get a value which is passed to the *test*, *test-not* or *predicate* function in place of the element. For example, if *key* is **car**, the car of each element is compared or tested. The default for *key* is **nil**, which means to compare or test the element itself.

from-end — If *from-end* is non-**nil**, elements are (conceptually) processed in the reverse of the sequence order, from the later elements to the earlier ones. In some functions this argument makes no difference, or matters only when *count* is non-**nil**.

Note: the actual testing of elements may happen in any order.

count — *count*, if not **nil**, should be an integer specifying the number of matching elements to be processed. For example, if *count* is 2, only the first two elements that match are removed, replaced, etc. If *from-end* is non-**nil**, the last two matching elements are the ones removed or replaced.

The default for *count* is **nil**, which means all elements are tested and all matching ones are processed.

9.4.1 Removing Elements from Sequences

These functions remove certain elements of a sequence. The **remove** series functions copy the argument; the **delete** series functions can modify it destructively (currently they always copy anyway if the argument is a vector).

remove-if *predicate sequence* &key (*start* 0) *end count key from-end*
delete-if *predicate sequence* &key (*start* 0) *end count key from-end*

Returns a sequence like *sequence* but missing any elements that satisfy *predicate*. *predicate* is a function of one argument which is applied to one element at a time; if *predicate* returns non-**nil**, that element is removed. **remove-if** copies structure as necessary to avoid modifying *sequence*, while **delete-if** can either modify the original sequence and return it or make a copy and return that. (Currently, a list is always modified, and a vector is always copied, but don't depend on this.)

The *start*, *end*, *key count* and *from-end* arguments are handled in the standard way.

```
(remove-if 'plusp '(1 -1 2 -2 3 -3))  =>  (-1 -2 -3)
(remove-if 'plusp '(1 -1 2 -2 3 -3) :count 2)
  =>  (-1 -2 3 -3)
(remove-if 'plusp '(1 -1 2 -2 3 -3) :count 2 :from-end t)
  =>  (1 -1 -2 -3)
(remove-if 'plusp '(1 -1 2 -2 3 -3) :start 4)
  =>  (1 -1 2 -2 -3)
(remove-if 'zerop '(1 -1 2 -2 3 -3) :key '1-)
  =>  (-1 2 -2 3 -3)
```

remove-if-not *predicate sequence* &key (*start* 0) *end count key from-end*
delete-if-not *predicate sequence* &key (*start* 0) *end count key from-end*

Like remove-if and delete-if except that the elements removed are those for which *predicate* returns nil.

cli:remove *item sequence* &key (*test* 'eql) *test-not* (*start* 0) *end count key from-end*
cli:delete *item sequence* &key (*test* 'eql) *test-not* (*start* 0) *end count key from-end*

The Common Lisp functions for eliminating elements from a sequence test the elements of *sequence* one by one by comparison with *item*, using the *test* or *test-not* function, and eliminate the elements that match. cli:remove copies structure as necessary to avoid modifying *sequence*, while cli:delete can either modify the original sequence and return it or make a copy and return that. (Currently, a list is always modified, and a vector is always copied.)

The *start*, *end*, *key count* and *from-end* arguments are handled in the standard way.

```
(cli:remove 'x '(x (a) (x) (a x)))
  =>  ((a) (x) (a x))

(cli:remove 'x '((a) (x) (a x)) :test 'memq)
  =>  ((a))

(cli:remove 'x '((a) (x) (a x)) :test-not 'memq)
  =>  ((x) (a x))

(cli:remove 'x '((a) (x) (a x))
            :test 'memq :count 1)
  =>  ((a) (a x))

(cli:remove 'x '((a) (x) (a x)) :key 'car)
  =>  ((a) (a x))
```

These functions are available under the names remove and delete in Common Lisp programs. Traditional Zetalisp provides functions remove and delete which serve similar functions, on lists only, and with different calling sequences; see page 105 and page 105. Traditional programs can call these functions as cli:remove and cli:delete.

remove-duplicates *sequence* &key (*test* 'eql) *test-not* (*start* 0) *end* *key* *from-end*
delete-duplicates *sequence* &key (*test* 'eql) *test-not* (*start* 0) *end* *key* *from-end*

remove-duplicates returns a new sequence like *sequence* except that all but one of any set of matching elements have been removed. delete-duplicates is the same except that it may destructively modify and then return *sequence* itself.

Elements are compared using *test*, a function of two arguments. Two elements match if *test* returns non-nil. Each element is compared with all the following elements and slated for removal if it matches any of them.

If *test-not* is specified, it is used instead of *test*, but then elements match if *test-not* returns nil. If neither *test* nor *test-not* is specified, eql is used for *test*.

If *key* is non-nil, it should be a function of one argument. *key* is applied to each element, and the value *key* returns is passed to *test* or *test-not*.

If *from-end* is non-nil, then elements are processed (conceptually) from the end of *sequence* forward. Each element is compared with all the preceding ones and slated for removal if it matches any of them. For a well-behaved comparison function, the only difference *from-end* makes is which elements of a matching set are removed. Normally the last one is kept; with *from-end*, it is the first one that is kept.

If *start* or *end* is used to restrict processing to a portion of *sequence*, both removal and comparison are restricted. An element is removed only if it is itself within the specified portion, and matches another element within the specified portion.

9.4.2 Substitution Functions

The functions in this section substitute a new value for certain of the elements in a sequence—those that match a specified object or satisfy a predicate. For example, you could replace every t in the sequence with nil, leaving all elements other than t unchanged. The substitute series functions make a copy and return it, leaving the original sequence unmodified. The nsubstitute series functions always alter the original sequence destructively and return it. They do not use up any storage.

Note the difference between these functions and the function cli:subst. subst operates only on lists, and it searches all levels of list structure in both car and cdr positions. substitute, when given a list, considers for replacement only the elements of the list.

substitute-if *newitem* *predicate* *sequence* &key *start* *end* *count* *key* *from-end*
nsubstitute-if *newitem* *predicate* *sequence* &key *start* *end* *count* *key* *from-end*

substitute-if returns a new sequence like *sequence* but with *newitem* substituted for each element of *sequence* that satisfies *predicate*. *sequence* itself is unchanged. If it is a list, only enough of it is copied to avoid changing *sequence*.

nsubstitute-if replaces elements in *sequence* itself, modifying it destructively, and returns *sequence*.

start, *end*, *key*, *count* and *from-end* are handled in the standard fashion as described above.

```
        (substitute-if 0 'plusp '(1 -1 2 -2 3) :from-end t :count 2)
          =>  (1 -1 0 -2 0)
```

substitute-if-not *newitem predicate sequence* &key *start end count key from-end*
nsubstitute-if-not *newitem predicate sequence* &key *start end count key from-end*

Like substitute-if and nsubstitute-if except that the elements replaced are those for which *predicate* returns nil.

substitute *newitem olditem sequence* &key (*test* 'eql) *test-not start end count key from-end*
nsubstitute *newitem olditem sequence* &key (*test* 'eql) *test-not start end count key from-end*

Like substitute-if and nsubstitute-if except that elements are tested by comparison with *olditem*, using *test* or *test-not* as a comparison function.

start, *end*, *key*, *count* and *from-end* are handled in the standard fashion as described above.

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

9.4.3 Searching for Elements

The functions in this section find an element or elements of a sequence which satisfy a predicate or match a specified object. The position series functions find one element and return the index of the element found in the specified sequence. The find series functions return the element itself. The count series functions find all the elements that match and returns the number of them that were found.

All of the functions accept the keyword arguments *start*, *end*, *count* and *from-end*, and handle them in the standard way described in section 9.4, page 193.

position-if *predicate sequence* &key (*start* 0) *end key from-end*
find-if *predicate sequence* &key (*start* 0) *end key from-end*

Find the first element of *sequence* (last element, if *from-end* is non-nil) which satisfies *predicate*. position-if returns the index in sequence of the element found; find-if returns the element itself. If no element is found, the value is nil for either function.

See section 9.4, page 193 for a description of the standard arguments *start*, *end* and *key*. If *start* or *end* is used to restrict operation to a portion of *sequence*, elements outside the portion are not tested, but the index returned is still the index in the entire sequence.

```
(position-if 'plusp '(-3 -2 -1 0 1 2 3))
  =>  4
(find-if 'plusp '(-3 -2 -1 0 1 2 3))
  =>  1
(position-if 'plusp '(-3 -2 -1 0 1 2 3) :start 5)
  =>  5
(position-if 'plusp '(-3 -2 -1 0 1 2 3) :from-end t)
  =>  6
(find-if 'plusp '(-3 -2 -1 0 1 2 3) :from-end t)
  =>  3
```

position-if-not *predicate sequence* &key (*start* 0) *end key from-end*
find-if-not *predicate sequence* &key (*start* 0) *end key from-end*
Like position-if and find-if but search for an element for which *predicate* returns nil.

position *item sequence sequence* &key *test test-not* (*start* 0) *end key from-end*
find *item sequence sequence* &key *test test-not* (*start* 0) *end key from-end*
Like position-if and find-if but search for an element which matches *item*, using *test* or *test-not* for comparison.

```
(position #\A "BabA" :test 'char-equal)  =>  1
(position #*/A "BabA" :test 'equalp)  =>  1
(position #\A "BabA" :test 'char=)  =>  3
(position #*/A "BabA" :test 'eq)  =>  3
```

find-position-in-list is equivalent to position with eq as the value of *test*.

count-if *predicate sequence* &key *start end key*
Tests each element of *sequence* with *predicate* and counts how many times *predicate* returns non-nil. This number is returned.

start, *end* and *key* are used in the standard way, as described in section 9.4, page 193. The *from-end* keyword argument is accepted without error, but it has no effect.

```
(count-if 'symbolp #(a b "foo" 3))  =>  2
```

count-if-not *predicate sequence* &key *start end key*
Like count-if but returns the number of elements for which *predicate* returns **nil**.

count *item sequence* &key *test test-not start end key*
Like count but returns the number of elements which match *item*. *test* or *test-not* is the function used for the comparison.

```
(count 4 '(1 2 3 4 5) :test '>)  =>  3
```

9.5 Comparison Functions

mismatch *sequence1 sequence2* &key (*test* 'eql) *test-not* (*start1* 0) *end1* (*start2* 0) *end2* *key* *from-end*

Compares successive elements of *sequence1* with successive elements of *sequence2*, returning nil if they all match, or else the index in *sequence1* of the first mismatch. If the sequences differ in length but match as far as they go, the value is the index in *sequence1* of the place where one sequence ran out. If *sequence1* is the one which ran out, this value equals the length of *sequence1*, so it isn't the index of an actual element, but it still describes the place where comparison stopped.

Elements are compared using the function *test*, which should accept two arguments. If it returns non-nil, the elements are considered to match. If you specify *test-not* instead of *test*, it is used similarly as a function, but the elements match if *test-not* returns nil.

If *key* is non-nil, it should be a function of one argument. It is applied to each element to get an object to pass to *test* or *test-not* in place of the element. Thus, if car is supplied as *key*, the cars of the elements are compared using *test* or *test-not*.

start1 and *end1* can be used to specify a portion of *sequence1* to use in the comparison, and *start2* and *end2* can be used to specify a portion of *sequence2*. The comparison uses the first element of each sequence portion, then the second element of each sequence portion, and so on. If the two specified portions differ in length, comparison stops where the first one runs out. In any case, the index returned by **mismatch** is still relative to the whole of *sequence1*.

If *from-end* is non-nil, the comparison proceeds conceptually from the end of each sequence or portion. The first comparison uses the last element of each sequence portion, the second comparison uses the next-to-the-last element of each sequence portion, and so on. When a mismatch is encountered, the value returned is *one greater than* the index of the first mismatch encountered in order of processing (closest to the ends of the sequences).

```
(mismatch "Foo" "Fox")  =>  2
(mismatch "Foo" "FOO" :test 'char-equal)  =>  nil
(mismatch "Foo" "FOO" :key 'char-upcase)  =>  nil
(mismatch '(a b) #(a b c))  =>  2
(mismatch "Win" "The Winner" :start2 4 :end2 7)  =>  nil
(mismatch "Foo" "Boo" :from-end t)  =>  1
```

search *for-sequence-1 in-sequence-2* &key *from-end test test-not key* (*start1* 0) *end1* (*start2* 0) *end2*

Searches *in-sequence-2* (or portion of it) for a subsequence that matches *for-sequence-1* (or portion of it) element by element, and returns the index in *in-sequence-2* of the beginning of the matching subsequence. If no matching subsequence is found, the value is nil. The comparison of each subsequence of *in-sequence-2* is done with **mismatch**, and the *test*, *test-not* and *key* arguments are used only to pass along to **mismatch**.

Normally, subsequences are considered starting with the beginning of the specified portion of *in-sequence-2* and proceeding toward the end. The value is therefore the index of the earliest subsequence that matches. If *from-end* is non-nil, the subsequences are tried in the reverse order, and the value identifies the latest subsequence that matches. In either case, the value identifies the beginning of the subsequence found.

```
(search '(#\A #\B) "cabbage" :test 'char-equal)  =>  1
```

9.6 Sorting and Merging

Several functions are provided for sorting vectors and lists. These functions use algorithms which always terminate no matter what sorting predicate is used, provided only that the predicate always terminates. The main sorting functions are not *stable*; that is, equal items may not stay in their original order. If you want a stable sort, use the stable versions. But if you don't care about stability, don't use them since stable algorithms are significantly slower.

After sorting, the argument (be it list or vector) has been rearranged internally so as to be completely ordered. In the case of a vector argument, this is accomplished by permuting the elements of the vector, while in the list case, the list is reordered by rplacd's in the same manner as nreverse. Thus if the argument should not be clobbered, the user must sort a copy of the argument, obtainable by fillarray or copylist, as appropriate. Furthermore, sort of a list is like delq in that it should not be used for effect; the result is conceptually the same as the argument but in fact is a different Lisp object.

Should the comparison predicate cause an error, such as a wrong type argument error, the state of the list or vector being sorted is undefined. However, if the error is corrected the sort proceeds correctly.

The sorting package is smart about compact lists; it sorts compact sublists as if they were vectors. See section 5.4, page 100 for an explanation of compact lists, and MIT A. I. Lab Memo 587 by Guy L. Steele Jr. for an explanation of the sorting algorithm.

sort *sequence predicate*

The first argument to sort is a vector or a list whose elements are to be sorted. The second is a predicate, which must be applicable to all the objects in the sequence. The predicate should take two arguments, and return non-nil if and only if the first argument is strictly less than the second (in some appropriate sense).

The sort function proceeds to reorder the elements of the sequence according to the predicate, and returns a modified sequence. Note that since sorting requires many comparisons, and thus many calls to the predicate, sorting is much faster if the predicate is a compiled function rather than interpreted.

Example: Sort a list alphabetically by the first symbol found at any level in each element.

```
(defun mostcar (x)
    (cond ((symbolp x) x)
          ((mostcar (car x)))))

(sort 'fooarray
      #'(lambda (x y)
          (string-lessp (mostcar x) (mostcar y))))
```

If **fooarray** contained these items before the sort:

```
(Tokens (The alien lurks tonight))
(Carpenters (Close to you))
((Rolling Stones) (Brown sugar))
((Beach Boys) (I get around))
(Beatles (I want to hold you up))
```

then after the sort **fooarray** would contain:

```
((Beach Boys) (I get around))
(Beatles (I want to hold you up))
(Carpenters (Close to you))
((Rolling Stones) (Brown sugar))
(Tokens (The alien lurks tonight))
```

When **sort** is given a list, it may change the order of the conses of the list (using **rplacd**), and so it cannot be used merely for side-effect; only the *returned value* of **sort** is the sorted list. The original list may have some of its elements missing when **sort** returns. If you need both the original list and the sorted list, you must copy the original and sort the copy (see **copylist**, page 94).

Sorting a vector just moves the elements of the vector into different places, and so sorting a vector for side-effect only is all right.

If the argument to **sort** is a vector with a fill pointer, note that, like most functions, **sort** considers the active length of the vector to be the length, and so only the active part of the vector is sorted (see **array-active-length**, page 174).

sortcar *sequence predicate*

sortcar is the same as **sort** except that the predicate is applied to the cars of the elements of *sequence*, instead of directly to the elements of *sequence*. Example:

```
(sortcar '((3 . dog) (1 . cat) (2 . bird)) #'<)
                         =>   ((1 . cat) (2 . bird) (3 . dog))
```

Remember that **sortcar**, when given a list, may change the order of the conses of the list (using **rplacd**), and so it cannot be used merely for side-effect; only the *returned value* of **sortcar** is the sorted list. The original list is destroyed by sorting.

stable-sort *sequence predicate*

stable-sort is like sort, but if two elements of *sequence* are equal, i.e. *predicate* returns nil when applied to them in either order, then they remain in their original order.

stable-sortcar *sequence predicate*

stable-sortcar is like sortcar, but if two elements of *sequence* are equal, i.e. *predicate* returns nil when applied to their cars in either order, then they remain in their original order.

sort-grouped-array *array group-size predicate*

sort-grouped-array considers its array argument to be composed of records of *group-size* elements each. These records are considered as units, and are sorted with respect to one another. The *predicate* is applied to the first element of each record; so the first elements act as the keys on which the records are sorted.

sort-grouped-array-group-key *array group-size predicate*

This is like sort-grouped-array except that the *predicate* is applied to four arguments: an array, an index into that array, a second array, and an index into the second array. *predicate* should consider each index as the subscript of the first element of a record in the corresponding array, and compare the two records. This is more general than sort-grouped-array since the function can get at all of the elements of the relevant records, instead of only the first element.

merge *result-type sequence1 sequence2 predicate* &key *key*

Returns a single sequence containing the elements of *sequence1* and *sequence2* interleaved in order according to *predicate*. The length of the result sequence is the sum of the lengths of *sequence1* and *sequence2*. *result-type* specifies the type of sequence to create, as in make-sequence.

The interleaving is done by taking the next element of *sequence1* unless the next element of *sequence2* is "less" than it according to *predicate*. Therefore, if each of the argument sequences is sorted, the result of merge is also sorted.

key, if non-nil, is applied to each element to get the object to pass to *predicate*, rather than the element itself. Thus, if *key* is car, the cars of the elements are compared rather than the entire elements.

```
(merge 'list '(1 2 5 6) '(3 5.0 5.1) '<)
  =>  (1 2 3 5 5.0 5.1 6)
```

10. Characters and Strings

A string is a one-dimensional array representing a sequence of characters. The printed representation of a string is its characters enclosed in quotation marks, for example "foo bar". Strings are constants, that is, evaluating a string returns that string. Strings are the right data type to use for text-processing.

Individual characters can be represented by *character objects* or by fixnums. A character object is actually the same as a fixnum except that it has a recognizably different data type and prints differently. Without escaping, a character object is printed by outputting the character it represents. With escaping, a character object prints as *#\char* in Common Lisp syntax or as *#*/char* in traditional syntax; see section 10.1.1, page 205 and page 522. By contrast, a fixnum would in all cases print as a sequence of digits. Character objects are accepted by most numeric functions in place of fixnums, and may be used as array indices. When evaluated, they are constants.

The character object data type was introduced recently for Common Lisp support. Traditionally characters were always represented as fixnums, and nearly all system and user code still does so. Character objects are interchangeable with fixnums in most contexts, but not in eq, which is often used to compare the result of the stream input operations such as :tyi, since that might be nil. Therefore, the stream input operations still return fixnums that represent characters. Aside from this, Common Lisp functions that return a character return a character object, while traditional functions return a fixnum. The fixnum which is the character code representing *char* can be written as *#/char* in traditional syntax. This is equivalent to writing the fixnum using digits, but does not require you to know the character code.

Most strings are arrays of type art-string, where each element is stored in eight bits. Only characters with character code less than 256 can be stored in an ordinary string; these characters form the type string-char. A string can also be an array of type art-fat-string, where each element holds a sixteen-bit unsigned fixnum. The extra bits allow for multiple fonts or an expanded character set.

Since strings are arrays, the usual array-referencing function aref is used to extract characters from strings. For example, (aref "frob" 1) returns the representation of lower case r. The first character is at index zero.

Conceptually, the elements of a string are character objects. This is what Common Lisp programs expect to see when they do aref (or char, which on the Lisp Machine is synonymous with aref) on a string. But nearly all Lisp Machine programs are traditional, and expect elements of strings to be fixnums. Therefore, aref of a string actually returns a fixnum. A distinct version of aref exists for Common Lisp programs. It is cli:aref and it does return character objects if given a string. For all other kinds of arrays, aref and cli:aref are equivalent.

```
(aref "Foo" 1)  =>  #o157
(cli:aref "Foo" 1)  =>  #*/o
```

It is also legal to store into strings, for example using setf of aref. As with rplaca on lists, this changes the actual object; you must be careful to understand where side-effects will propagate. It makes no difference whether a character object or a fixnum is stored. When you are making strings that you intend to change later, you probably want to create an array with a fill-pointer (see page 166) so that you can change the length of the string as well as the contents. The length of a string is always computed using array-active-length, so that if a string has a fill-pointer, its value is used as the length.

The functions described in this section provide a variety of useful operations on strings. In place of a string, most of these functions accept a symbol or a fixnum as an argument, coercing it into a string. Given a symbol, its print name, which is a string, is used. Given a fixnum, a one-character string containing the character designated by that fixnum is used. Several of the functions actually work on any type of one-dimensional array and may be useful for other than string processing; these are the functions such as substring and string-length which do not depend on the elements of the string being characters.

The generic sequence functions in chapter 9 may also be used on strings.

10.1 Characters

The Lisp Machine data type for character objects is a recent addition to the system. Most programs still use fixnums to represent characters.

Common Lisp programs typically work with actual character objects but programs traditionally use fixnums to represent characters. The new Common Lisp functions for operating with characters have been implemented to accept fixnums as well, so that they can be used equally well from traditional programs.

characterp *object*

t if *object* is a character object; nil otherwise. In particular, it is nil if *object* is a fixnum such as traditional programs use to represent characters.

character *object*

Coerces object to a single character, represented as a fixnum. If object is a number, it is returned. If object is a string or an array, its first element is returned. If object is a symbol, the first character of its pname is returned. Otherwise an error occurs. The way characters are represented as fixnums is explained in section 10.1.1, page 205.

cli:character *object*

Coerces *object* into a character and returns the character as a character object for Common Lisp programs.

int-char *fixnum*

Converts *fixnum*, regarded as representing a character, to a character object. This is a special case of cli:character. (int-char #o101) is the character object for A. If a character object is given as an argument, it is returned unchanged.

char-int *char*

Converts *char*, a character object, to the fixnum which represents the same character. This is the inverse of **int-char**. It may also be given a fixnum as argument, in which case the value is the same fixnum.

10.1.1 Components of a Character

A character object, or a fixnum which is interpreted as a character, contains three separate pieces of information: the *character code*, the *font number*, and the *modifier bits*. Each of these things is an integer from a fixed range. The character code ranges from 0 to 377 (octal), the font number from 0 to 377 (octal), and the modifier bits from 0 to 17 (octal). These numeric constants should not appear in programs; instead, use the constant symbols **char-code-limit**, and so on, described below.

Ordinary strings can hold only characters whose font number and modifier bits are zero. Fat strings can hold characters with any font number, but the modifier bits must still be zero.

Character codes less than 200 octal are printing graphics; when output to a device they are assumed to print a character and move the cursor one character position to the right. (All software provides for variable-width fonts, so the term "character position" shouldn't be taken too literally.)

Character codes 200 through 236 octal are used for special characters. Character 200 is a "null character", which does not correspond to any key on the keyboard. The null character is not used for anything much; **fasload** uses it internally. Characters 201 through 236 correspond to the special function keys on the keyboard such as **Return** and **Call**. The remaining character codes 237 through 377 octal are reserved for future expansion.

Most of the special characters do not normally appear in files (although it is not forbidden for files to contain them). These characters exist mainly to be used as "commands" from the keyboard. A few special characters, however, are "format effectors" which are just as legitimate as printing characters in text files. The names and meanings of these characters are:

Return	The "newline" character, which separates lines of text. We do not use the PDP-10 convention which separates lines by a pair of characters, a "carriage return" and a "linefeed".
Page	The "page separator" character, which separates pages of text.
Tab	The "tabulation" character, which spaces to the right until the next "tab stop". Tab stops are normally every 8 character positions.

The space character is considered to be a printing character whose printed image happens to be blank, rather than a format effector.

When a letter is typed with any of the modifier bit keys (**Control**, **Meta**, **Super**, or **Hyper**), the letter is normally upper-case. If the **Shift** key is pressed as well, then the letter becomes lower-case. This is exactly the reverse of what the **Shift** key does to letters without control bits. (The **Shift-lock** key has no effect on letters with control bits.)

char-code *char*
char-font *char*
char-bits *char*

Return the character code of *char*, the font number of *char*, and the modifier bits value of *char*. *char* may be a fixnum or a character object; the value is always a fixnum.

These used to be written as

```
(ldb %%ch-char char)
(ldb %%ch-font char)
(ldb %%ch-control-meta char)
```

Such use of **ldb** is frequent but obsolete.

char-code-limit *Constant*

A constant whose value is a bound on the maximum code of any character. In the Lisp Machine, currently, it is 400 (octal).

char-font-limit *Constant*

A constant whose value is a bound on the maximum font number value of any character. In the Lisp Machine, currently, it is 400 (octal).

char-bits-limit *Constant*

A constant whose value is a bound on the maximum modifier bits value of any character. In the Lisp Machine, currently, it is 20 (octal). Thus, there are four modifier bits. These are just the familiar Control, Meta, Super and Hyper bits.

char-control-bit *Constant*
char-meta-bit *Constant*
char-super-bit *Constant*
char-hyper-bit *Constant*

Constants with values 1, 2, 4 and 8. These give the meanings of the bits within the bits-field of a character object. Thus, **(bit-test char-meta-bit (char-bits** *char***))** would be non-nil if *char* is a meta-character. (This can also be tested with **char-bit**.)

char-bit *char name*

t if *char* has the modifier bit named by *name*. *name* is one of the following four symbols: **:control**, **:meta**, **:super**, and **:hyper**.

```
(char-bit #\meta-x :meta) => t.
```

set-char-bit *char name newvalue*

Returns a character like *char* except that the bit specified by *name* is present if *newvalue* is non-nil, absent otherwise. Thus,

```
(set-char-bit #\x :meta t) => #\meta-x.
```

The value is a fixnum if *char* is one; a character object if *char* is one.

Until recently the only way to access the character code, font and modifier bits was with **ldb**, using the byte field names listed below. Most code still uses that method, but it is obsolete; **char-bit** should be used instead.

%%kbd-char
%%ch-char Specifies the byte containing the character code.

%%ch-font Specifies the byte containing the font number.

%%kbd-control
Specifies the byte containing the Control bit.

%%kbd-meta Specifies the byte containing the Meta bit.

%%kbd-super Specifies the byte containing the Super bit.

%%kbd-hyper Specifies the byte containing the Hyper bit.

%%kbd-control-meta
Specifies the byte containing all the modifier bits.

Characters are sometimes used to represent mouse clicks. The character says which button was pressed and how many times. Refer to the Window System manual for an explanation of how these characters are generated.

tv:kbd-mouse-p *char*
t if *char* is a character used to represent a mouse click. Such characters are always distinguishable from characters that represent keyboard input.

%%kbd-mouse-button *Constant*
The value of %%kbd-mouse-button is a byte specifier for the field in a mouse signal that says which button was clicked. The byte contains 0, 1, or 2 for the left, middle, or right button, respectively.

%%kbd-mouse-n-clicks *Constant*
The value of %%kbd-mouse-n-clicks is a byte specifier for the field in a mouse signal that says how many times the button was clicked. The byte contains one less than the number of times the button was clicked.

10.1.2 Constructing Character Objects

code-char *code* &optional (*bits* 0) (*font* 0)
make-char *code* &optional (*bits* 0) (*font* 0)
Returns a character object made from *code*, *bits* and *font*. Common Lisp says that not all combinations may be valid, and that nil is returned for an invalid combination. On the Lisp Machine, any combination is valid if the arguments are valid individually.

According to Common Lisp, code-char requires a number as a first argument, whereas make-char requires a character object, whose character code is used. On the Lisp Machine, either function may be used in either way.

digit-char *weight* &optional (*radix* 10.) (*font* 0)
Returns a character object which is the digit with the specified weight, and with font as specified. However, if there is no suitable character which has weight *weight* in the specified radix, the value is nil. If the "digit" is a letter (which happens if *weight* is greater than 9), it is returned in upper case.

tv:make-mouse-char *button n-clicks*

Returns the fixnum character code that represents a mouse click in the standard way. tv:mouse-char-p of this value is t. *button* is 0 for the leftbutton, 1 for the middle button, or 2 for the right button. *n-clicks* is one less than the number of clicks (1 for a double click, 0 normally).

10.1.3 The Character Set

Here are the numerical values of the characters in the Zetalisp character set. It should never be necessary for a user or a source program to know these values. Indeed, they are likely to be changed in the future. There are symbolic names for all characters; see the section on character names, below.

It is worth pointing out that the Zetalisp character set is different from the ASCII character set. File servers operating on hosts that use ASCII for storing text files automatically perform character set conversion when text files are read or written. The details of the mapping are explained in section 25.8, page 607.

000	center-dot (·)	040	space	100	@	140	`
001	down arrow (↓)	041	!	101	A	141	a
002	alpha (α)	042	"	102	B	142	b
003	beta (β)	043	#	103	C	143	c
004	and-sign (∧)	044	$	104	D	144	d
005	not-sign (¬)	045	%	105	E	145	e
006	epsilon (ε)	046	&	106	F	146	f
007	pi (π)	047	'	107	G	147	g
010	lambda (λ)	050	(	110	H	150	h
011	gamma (γ)	051	)	111	I	151	i
012	delta (δ)	052	*	112	J	152	j
013	uparrow (↑)	053	+	113	K	153	k
014	plus-minus (±)	054	,	114	L	154	l
015	circle-plus (⊕)	055	-	115	M	155	m
016	infinity (∞)	056	.	116	N	156	n
017	partial delta (∂)	057	/	117	O	157	o
020	left horseshoe (⊂)	060	0	120	P	160	p
021	right horseshoe (⊃)	061	1	121	Q	161	q
022	up horseshoe (∩)	062	2	122	R	162	r
023	down horseshoe (∪)	063	3	123	S	163	s
024	universal quantifier (∀)	064	4	124	T	164	t
025	existential quantifier (∃)	065	5	125	U	165	u
026	circle-X (⊗)	066	6	126	V	166	v
027	double-arrow (↔)	067	7	127	W	167	w
030	left arrow (←)	070	8	130	X	170	x
031	right arrow (→)	071	9	131	Y	171	y
032	not-equals (≠)	072	:	132	Z	172	z
033	diamond (altmode) (◊)	073	;	133	[	173	{
034	less-or-equal (≤)	074	<	134	\	174	|
035	greater-or-equal (≥)	075	=	135	]	175	}
036	equivalence (≡)	076	>	136	^	176	~
037	or (∨)	077	?	137	_	177	∫

200	Null character	210	Overstrike	220	Stop-output	230	Roman-iv
201	Break	211	Tab	221	Abort	231	Hand-up
202	Clear	212	Line	222	Resume	232	Hand-down
203	Call	213	Delete	223	Status	233	Hand-left
204	Terminal escape	214	Page	224	End	234	Hand-right
205	Macro/backnext	215	Return	225	Roman-i	235	System
206	Help	216	Quote	226	Roman-ii	236	Network
207	Rubout	217	Hold-output	227	Roman-iii		

237-377 reserved for the future

The Lisp Machine Character Set
(all numbers in octal)

10.1.4 Classifying Characters

string-char-p *char*
t if *char* is a character that can be stored in a string. On the Lisp Machine, this is true if the font and modifier bits of *char* are zero.

standard-char-p *char*
t if *char* is a standard Common Lisp character: any of the 95 ASCII printing characters (including Space), and the Return character. Thus (standard-char-p #\end) is nil.

graphic-char-p *char*
t if *char* is a graphic character: one which has a printed shape. A, -, Space and ε are all graphic characters; Return, End and Abort are not. A character whose modifier bits are nonzero is never graphic.

Ordinary output to windows prints graphic characters using the current font. Nongraphic characters are printed using lozenges unless they have special formatting meanings (as Return does).

alpha-char-p *char*
t if *char* is a letter with zero modifier bits.

digit-char-p *char* &optional (*radix* 10.)
If *char* is a digit available in the specified radix, returns the *weight* of that digit. Otherwise, it returns nil. If the modifier bits of *char* are nonzero, the value is always nil. (It would be more useful to ignore the modifier bits, but this decision provides Common Lisp with a foolish consistency.) Examples:

```
(digit-char-p #\8 8) => nil
(digit-char-p #\8 9) => 8
(digit-char-p #\F 16.) => 15.
(digit-char-p #\c-8 anything) => nil
```

alphanumericp *char*
t if *char* is a letter or a digit 0 through 9, with zero modifier bits.

10.1.5 Comparing Characters

char-equal &rest *chars*
This is the primitive for comparing characters for equality; many of the string functions call it. The arguments may be fixnums or character objects indiscriminately. The result is t if the characters are equal ignoring case, font and modifier bits, otherwise nil.

char-not-equal &rest *chars*
t if the arguments are all different as characters, ignoring case, font and modifier bits.

char-lessp &rest *chars*
char-greaterp &rest *chars*
char-not-lessp &rest *chars*
char-not-greaterp &rest *chars*

Ordered comparison of characters, ignoring case, font and modifier bits. These are the primitives for comparing characters for order; many of the string functions call it. The arguments may be fixnums or character objects. The result is t if the arguments are in strictly increasing (strictly decreasing, nonincreasing, nondecreasing) order. Details of the ordering of characters are in section 10.1.1, page 205.

char= *char1* &rest *chars*
char//= *char1* &rest *chars*
char> *char1* &rest *chars*
char< *char1* &rest *chars*
char>= *char1* &rest *chars*
char<= *char1* &rest *chars*

These are the Common Lisp functions for comparing characters and including the case, font and bits in the comparison. On the Lisp Machine they are synonyms for the numeric comparison functions =, >, etc. Note that in Common Lisp syntax you would write char/=, not char//=.

10.1.6 Character Names

Characters can sometimes be referred to by long names; as, for example, in the #\ construct in Lisp programs. Every basic character (zero modifier bits) which is not a graphic character has one or more standard names. Some graphic characters have standard names too. When a non-graphic character is output to a window, it appears as a lozenge containing the character's standard name.

char-name *char*

Returns the standard name (or one of the standard names) of *char*, or nil if there is none. The name is returned as a string. (char-name #\space) is the string "SPACE".

If *char* has nonzero modifier bits, the value is nil. Compound names such as Control-X are not constructed by this function.

name-char *name*

Returns (as a character object) the character for which *name* is a name, or returns nil if *name* is not a recognized character name. *name* may be a symbol or a string. Compound names such as Control-X are not recognized.

read uses this function to process the #\ construct when a character name is encountered.

The following are the recognized special character names, in alphabetical order except with synonyms together. Character names are encoded and decoded by the functions char-name and name-char (page 211).

First a list of the special function keys.

abort	break	call	clear-input, clear
delete	end	hand-down	hand-left
hand-right	hand-up	help	hold-output
line, lf	macro, back-next	network	
overstrike, backspace, bs		page, form, clear-screen	
quote	resume	return, cr	
roman-i	roman-ii	roman-iii	roman-iv
rubout	space, sp	status	stop-output
system	tab	terminal, esc	

These are printing characters that also have special names because they may be hard to type on the hosts that are used as file servers.

altmode	circle-plus	delta	gamma
integral	lambda	plus-minus	uparrow
center-dot	down-arrow	alpha	beta
and-sign	not-sign	epsilon	pi
lambda	gamma	delta	up-arrow
plus-minus	circle-plus	infinity	partial-delta
left-horseshoe	right-horseshoe	up-horseshoe	down-horseshoe
universal-quantifier		existential-quantifier	
circle-x	double-arrow	left-arrow	right-arrow
not-equal	altmode	less-or-equal	greater-or-equal
equivalence	or-sign		

The following names are for special characters sometimes used to represent single and double mouse clicks. The buttons can be called either l, m, r or 1, 2, 3 depending on stylistic preference.

mouse-l-1 or mouse-1-1	mouse-l-2 or mouse-1-2
mouse-m-1 or mouse-2-1	mouse-m-2 or mouse-2-2
mouse-r-1 or mouse-3-1	mouse-r-2 or mouse-3-2

10.2 Conversion to Upper or Lower Case

`upper-case-p` *char*
: t if *char* is an upper case letter with zero modifier bits.

`lower-case-p` *char*
: t if *char* is an lower case letter with zero modifier bits.

`both-case-p` *char*
: This Common Lisp function is defined to return t if *char* is a character which has distinct upper and lower case forms. On the Lisp Machine it returns t if *char* is a letter with zero modifier bits.

char-upcase *char*

If *char*, is a lower-case alphabetic character its upper-case form is returned; otherwise, *char* itself is returned. If font information or modifier bits are present, they are preserved. If *char* is a fixnum, the value is a fixnum. If *char* is a character object, the value is a character object.

char-downcase *char*

Similar, but converts to lower case.

string-upcase *string* &key (*start* 0) *end*

Returns a string like *string*, with all lower-case alphabetic characters replaced by the corresponding upper-case characters. If *start* or *end* is specified, only the specified portion of the string is converted, but in any case the entire string is returned.

The result is a copy of *string* unless no change is necessary. *string* itself is never modified.

string-downcase *string* &key (*start* 0) *end*

Similar, but converts to lower case.

string-capitalize *string* &key (*start* 0) *end*

Returns a string like *string* in which all, or the specified portion, has been processed by capitalizing each word. For this function, a word is any maximal sequence of letters or digits. It is capitalized by putting the first character (if it is a letter) in upper case and any letters in the rest of the word in lower case.

The result is a copy of *string* unless no change is necessary. *string* itself is never modified.

nstring-upcase *string* &key (*start* 0) *end*
nstring-downcase *string* &key (*start* 0) *end*
nstring-capitalize *string* &key (*start* 0) *end*

Like the previous functions except that they modify *string* itself and return it.

string-capitalize-words *string* &optional (*copy-p* t) (*spaces* t)

Puts each word in *string* into lower-case with an upper case initial, and if *spaces* is non-nil replaces each hyphen character with a space.

If *copy-p* is t, the value is a copy of *string*, and *string* itself is unchanged. Otherwise, *string* itself is returned, with its contents changed.

This function is somewhat obsolete. One can use string-capitalize followed optionally by string-subst-char.

See also the format operation ~(...~) on page 488.

10.3 Basic String Operations

make-string *size* &key (*initial-element* 0)
Creates and returns a string of length *size*, with each element initialized to *initial-element*, which may be a fixnum or a character.

string *x*
Coerces *x* into a string. Most of the string functions apply this to their string arguments. If *x* is a string (or any array), it is returned. If *x* is a symbol, its pname is returned. If *x* is a non-negative fixnum less than 400 octal, a one-character-long string containing it is created and returned. If *x* is an instance that supports the :string-for-printing operation (such as, a pathname) then the result of that operation is returned. Otherwise, an error is signaled.

If you want to get the printed representation of an object into the form of a string, this function is *not* what you should use. You can use format, passing a first argument of nil (see page 483). You might also want to use with-output-to-string (see page 474).

string-length *string*
Returns the number of characters in *string*. This is 1 if *string* is a number or character object, the array-active-length (see page 174) if *string* is an array, or the array-active-length of the pname if *string* is a symbol.

string-equal *string1* *string2* &key (*start1* 0) (*start2* 0) *end1* *end2*
Compares two strings, returning t if they are equal and nil if they are not. The comparison ignores the font and case of the characters. equal calls string-equal if applied to two strings.

The keyword arguments *start1* and *start2* are the starting indices into the strings. *end1* and *end2* are the final indices; the comparison stops just *before* the final index. nil for *end1* or *end2* means stop at the end of the string.
Examples:

```
(string-equal "Foo" "foo") => t
(string-equal "foo" "bar") => nil
(string-equal "element" "select" 0 1 3 4) => t
```

An older calling sequence in which the *start* and *end* arguments are positional rather than keyword is still supported. The arguments come in the order *start1* *start2* *end1* *end2*. This calling sequence is obsolete and should be changed whenever found.

string-not-equal *string1* *string2* &key (*start1* 0) *end1* (*start2* 0) *end2*
(not (string-equal ...))

string= *string1* *string2* &key (*start1* 0) (*start2* 0) *end1* *end2*
is like string-equal except that case is significant.

```
(string= "A" "a") => nil
```

string≠ *string1 string2* &key (*start1* 0) *end1* (*start2* 0) *end2*
string//= *string1 string2* &key (*start1* 0) *end1* (*start2* 0) *end2*
(not (string= ...)). Note that in Common Lisp syntax you would write string/=, not string//=.

string-lessp *string1 string2* &key (*start1* 0) *end1* (*start2* 0) *end2*
string-greaterp *string1 string2* &key (*start1* 0) *end1* (*start2* 0) *end2*
string-not-greaterp *string1 string2* &key (*start1* 0) *end1* (*start2* 0) *end2*
string-not-lessp *string1 string2* &key (*start1* 0) *end1* (*start2* 0) *end2*
Compare all or the specified portions of *string1* and *string2* using dictionary order. Characters are compared using char-lessp and char-equal so that font and alphabetic case are ignored.

You can use these functions as predicates, but they do more. If the strings fit the condition (e.g. *string1* is strictly less in string-lessp) then the value is a number, the index in *string1* of the first point of difference between the strings. This equals the length of *string1* if the strings match. If the condition is not met, the value is nil.

```
(string-lessp "aa" "Ab") => 1
(string-lessp "aa" "Ab" :end1 1 :end2 1) => nil
(string-not-greaterp "Aa" "Ab" :end1 1 :end2 1) => 1
```

string< *string1 string2* &key (*start1* 0) *end1* (*start2* 0) *end2*
string> *string1 string2* &key (*start1* 0) *end1* (*start2* 0) *end2*
string>= *string1 string2* &key (*start1* 0) *end1* (*start2* 0) *end2*
string<= *string1 string2* &key (*start1* 0) *end1* (*start2* 0) *end2*
string≤ *string1 string2* &key (*start1* 0) *end1* (*start2* 0) *end2*
string≥ *string1 string2* &key (*start1* 0) *end1* (*start2* 0) *end2*
Like string-lessp, etc., but treat case and font as significant when comparing characters.

```
(string< "AA" "aa") => 0
(string-lessp "AA" "aa") => nil
```

string-compare *string1 string2* &optional (*start1* 0) (*start2* 0) *end1 end2*
Compares two strings using dictionary order (as defined by char-lessp). The arguments are interpreted as in string-equal. The result is 0 if the strings are equal, a negative number if *string1* is less than *string2*, or a positive number if *string1* is greater than *string2*. If the strings are not equal, the absolute value of the number returned is one greater than the index (in *string1*) where the first difference occurred.

substring *string start* &optional *end area*
Extracts a substring of *string*, starting at the character specified by *start* and going up to but not including the character specified by *end*. *start* and *end* are 0-origin indices. The length of the returned string is *end* minus *start*. If *end* is not specified it defaults to the length of *string*. The area in which the result is to be consed may be optionally specified.
Example:

```
(substring "Nebuchadnezzar" 4 8) => "chad"
```

nsubstring *string start* &optional *end area*

Is like substring except that the substring is not copied; instead an indirect array (see page 167) is created which shares part of the argument *string*. Modifying one string will modify the other.

Note that nsubstring does not necessarily use less storage than substring: an nsubstring of any length uses at least as much storage as a substring 12 characters long. So you shouldn't use this for efficiency; it is intended for uses in which it is important to have a substring which, if modified, will cause the original string to be modified too.

string-append &rest *strings*

Copies and concatenates any number of strings into a single string. With a single argument, string-append simply copies it. If there are no arguments, the value is an empty string. In fact, vectors of any type may be used as arguments, and the value is a vector capable of holding all the elements of all the arguments. Thus string-append can be used to copy and concatenate any type of vector. If the first argument is not an array (for example, if it is a character), the value is a string.
Example:

```
(string-append #\! "foo" #\!) => "!foo!"
```

string-nconc *modified-string* &rest *strings*

Is like string-append except that instead of making a new string containing the concatenation of its arguments, string-nconc modifies its first argument. *modified-string* must have a fill-pointer so that additional characters can be tacked onto it. Compare this with array-push-extend (page 178). The value of string-nconc is *modified-string* or a new, longer copy of it; in the latter case the original copy is forwarded to the new copy (see adjust-array-size, page 176). Unlike nconc, string-nconc with more than two arguments modifies only its first argument, not every argument but the last.

string-trim *char-set string*

Returns a substring of *string*, with all characters in *char-set* stripped off the beginning and end. *char-set* is a set of characters, which can be represented as a list of characters, a string of characters or a single character.
Example:

```
(string-trim '(#\sp) "  Dr. No  ") => "Dr. No"
(string-trim "ab" "abbafooabb") => "foo"
```

string-left-trim *char-set string*

Returns a substring of *string*, with all characters in *char-set* stripped off the beginning. *char-set* is a set of characters, which can be represented as a list of characters, a string of characters or a single character.

string-right-trim *char-set string*

Returns a substring of *string*, with all characters in *char-set* stripped off the end. *char-set* is a set of characters, which can be represented as a list of characters, a string of characters or a single character.

string-remove-fonts *string*

Returns a copy of *string* with each character truncated to 8 bits; that is, changed to font zero.

If *string* is an ordinary string of array type art-string, this does not change anything, but it makes a difference if *string* is an art-fat-string.

string-reverse *string*
string-nreverse *string*

Like reverse and nreverse, but on strings only (see page 190). There is no longer any reason to use these functions except that they coerce numbers and symbols into strings like the other string functions.

string-pluralize *string*

Returns a string containing the plural of the word in the argument *string*. Any added characters go in the same case as the last character of *string*.
Example:

```
        (string-pluralize "event") => "events"
        (string-pluralize "trufan") => "trufen"
        (string-pluralize "Can") => "Cans"
        (string-pluralize "key") => "keys"
        (string-pluralize "TRY") => "TRIES"
```

For words with multiple plural forms depending on the meaning, string-pluralize cannot always do the right thing.

string-select-a-or-an *word*

Returns "a" or "an" according to the string *word*; whichever one appears to be correct to use before *word* in English.

string-append-a-or-an *word*

Returns the result of appending "a " or "an ", whichever is appropriate, to the front of *word*.

%string-equal *string1 start1 string2 start2 count*

%string-equal is the microcode primitive used by string-equal. It returns t if the *count* characters of *string1* starting at *start1* are char-equal to the *count* characters of *string2* starting at *start2*, or nil if the characters are not equal or if *count* runs off the length of either array.

Instead of a fixnum, *count* may also be nil. In this case, %string-equal compares the substring from *start1* to (string-length *string1*) against the substring from *start2* to (string-length *string2*). If the lengths of these substrings differ, then they are not equal and nil is returned.

Note that *string1* and *string2* must really be strings; the usual coercion of symbols and fixnums to strings is not performed. This function is documented because certain programs which require high efficiency and are willing to pay the price of less generality may want to use %string-equal in place of string-equal.

Examples:

To compare the two strings *foo* and *bar*:
```
(%string-equal foo 0 bar 0 nil)
```
To see if the string *foo* starts with the characters "bar":
```
(%string-equal foo 0 "bar" 0 3)
```

alphabetic-case-affects-string-comparison *Variable*

If this variable is t, the functions %string-equal and %string-search consider case (and font) significant in comparing characters. Normally this variable is nil and those primitives ignore differences of case.

This variable may be bound by user programs around calls to %string-equal and %string-search-char, but do not set it globally, for that may cause system malfunctions.

10.4 String Searching

string-search-char *char string* &optional (*from* 0) *to consider-case*

Searches through *string* starting at the index *from*, which defaults to the beginning, and returns the index of the first character that is char-equal to *char*, or nil if none is found. If *to* is non-nil, it is used in place of (string-length *string*) to limit the extent of the search.

Example:
```
(string-search-char #\a "banana") => 1
```
Case (and font) is significant in comparison of characters if *consider-case* is non-nil. In other words, characters are compared using char= rather than char-equal.
```
(string-search-char #\a "BAnana" 0 nil t) => 3
```

%string-search-char *char string from to*

%string-search-char is the microcode primitive called by string-search-char and other functions. *string* must be an array and *char*, *from*, and *to* must be fixnums. The arguments are all required. Case-sensitivity is controlled by the value of the variable alphabetic-case-affects-string-comparison rather than by an argument. Except for these these differences, %string-search-char is the same as string-search-char. This function is documented for the benefit of those who require the maximum possible efficiency in string searching.

string-search-not-char *char string* &optional (*from* 0) *to consider-case*

Like string-search-char but searches *string* for a character different from *char*.

Example:
```
(string-search-not-char #\B "banana") => 1
(string-search-not-char #\B "banana" 0 nil t) => 0
```

string-search *key string* &optional (*from* 0) *to* (*key-from* 0) *key-to consider-case*

Searches for the string *key* in the string *string*. The search begins at *from*, which defaults to the beginning of *string*. The value returned is the index of the first character of the first instance of *key*, or nil if none is found. If *to* is non-nil, it is used in place of (string-length *string*) to limit the extent of the search.

The arguments *key-from* and *key-to* can be used to specify the portion of *key* to be searched for, rather than all of *key*.

Case and font are significant in character comparison if *consider-case* is non-nil.
Example:

```
(string-search "an" "banana") => 1
(string-search "an" "banana" 2) => 3
(string-search "tank" "banana" 2 nil 1 3) => 3
(string-search "an" "BAnaNA" 0 nil 0 nil t) => nil
```

string-search-set *char-set string* &optional (*from* 0) *to consider-case*

Searches through *string* looking for a character that is in *char-set*. *char-set* is a set of characters, which can be represented as a sequence of characters or a single character.

The search begins at the index *from*, which defaults to the beginning. It returns the index of the first character that is char-equal to some element of *char-set*, or nil if none is found. If *to* is non-nil, it is used in place of (string-length *string*) to limit the extent of the search.

Case and font are significant in character comparison if *consider-case* is non-nil.
Example:

```
(string-search-set '(#\n #\o) "banana") => 2
(string-search-set "no" "banana") => 2
```

string-search-not-set *char-set string* &optional (*from* 0) *to consider-case*

Like string-search-set but searches for a character that is *not* in *char-set*.
Example:

```
(string-search-not-set '(#\a #\b) "banana") => 2
```

string-reverse-search-char *char string* &optional *from* (*to* 0) *consider-case*

Searches through *string* in reverse order, starting from the index one less than *from* (nil for *from* starts at the end of *string*), and returns the index of the first character which is char-equal to *char*, or nil if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. The last (leftmost) character of *string* examined is the one at index *to*.

Case and font are significant in character comparison if *consider-case* is non-nil. In this case, char= is used for the comparison rather than char-equal.
Example:

```
(string-reverse-search-char #\n "banana") => 4
```

string-reverse-search-not-char *char string* &optional *from* (*to* 0) *consider-case*

Like string-reverse-search-char but searches for a character in *string* that is different from *char*.
Example:

```
(string-reverse-search-not-char #\a "banana") => 4
;4 is the index of the second "n"
```

string-reverse-search *key string* &optional *from* (*to* 0) (*key-from* 0) *key-to consider-case*

Searches for the string *key* in the string *string*. The search proceeds in reverse order, starting from the index one less than *from*, and returns the index of the first (leftmost) character of the first instance found, or nil if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. The *from* condition, restated, is that the instance of *key* found is the rightmost one whose rightmost character is before the *from*'th character of *string*. nil for *from* means the search starts at the end of *string*. The last (leftmost) character of *string* examined is the one at index *to*.

Example:
```
(string-reverse-search "na" "banana") => 4
```

The arguments *key-from* and *key-to* can be used to specify the portion of *key* to be searched for, rather than all of *key*. Case and font are significant in character comparison if *consider-case* is non-nil.

string-reverse-search-set *char-set string* &optional *from* (*to* 0) *consider-case*

Searches through *string* in reverse order for a character which is char-equal to some element of *char-set*. *char-set* is a set of characters, which can be represented as a list of characters, a string of characters or a single character.

The search starts from an index one less than *from*, and returns the index of the first suitable character found, or nil if none is found. nil for *from* means the search starts at the end of *string*. Note that the index returned is from the beginning of the string, although the search starts from the end. The last (leftmost) character of *string* examined is the one at index *to*.

Case and font are significant in character comparison if *consider-case* is non-nil. In this case, char= is used for the comparison rather than char-equal.

```
(string-reverse-search-set "ab" "banana") => 5
```

string-reverse-search-not-set *char-set string* &optional *from* (*to* 0) *consider-case*

Like string-reverse-search-set but searches for a character which is *not* in *char-set*.
```
(string-reverse-search-not-set '(#\a #\n) "banana") => 0
```

string-subst-char *new-char old-char string* (*copy-p* t) (*retain-font-p* t)

Returns a copy of *string* in which all occurrences of *old-char* have been replaced by *new-char*.

Case and font are ignored in comparing *old-char* against characters of *string*. Normally the font information of the character replaced is preserved, so that an *old-char* in font 3 is replaced by a *new-char* in font 3. If *retain-font-p* is nil, the font specified in *new-char* is stored whenever a character is replaced.

If *copy-p* is nil, *string* is modified destructively and returned. No copy is made.

substring-after-char *char string* &optional *start end area*

Returns a copy of the portion of *string* that follows the next occurrence of *char* after index *start*. The portion copied ends at index *end*. If *char* is not found before *end*, a null string is returned.

The value is consed in area *area*, or in default-cons-area, unless it is a null string. *start* defaults to zero, and *end* to the length of *string*.

See also make-symbol (page 133), which given a string makes a new uninterned symbol with that print name, and intern (page 645), which given a string returns the one and only symbol (in the current package) with that print name.

10.5 Maclisp-Compatible Functions

The following functions are provided primarily for Maclisp compatibility.

alphalessp *string1 string2*

(alphalessp *string1 string2*) is equivalent to (string-lessp *string1 string2*).

samepnamep *sym1 sym2*

This predicate is equivalent to string=.

getchar *string index*

Returns the *index*'th character of *string* as a symbol. Note that 1-origin indexing is used. This function is mainly for Maclisp compatibility; aref should be used to index into strings (but aref does not coerce symbols or numbers into strings).

getcharn *string index*

Returns the *index*'th character of *string* as a fixnum. Note that 1-origin indexing is used. This function is mainly for Maclisp compatibility; aref should be used to index into strings (but aref does not coerce symbols or numbers into strings).

ascii *x*

Like character, but returns a symbol whose printname is the character instead of returning a fixnum.
Examples:

```
(ascii #o101) => A
(ascii #o56) => /.
```

The symbol returned is interned in the current package (see chapter 27, page 636).

maknam *char-list*

Returns an uninterned symbol whose print-name is a string made up of the characters in *char-list*.
Example:

```
(maknam '(a b #\0 d)) => ab0d
```

implode *char-list*

implode is like maknam except that the returned symbol is interned in the current package.

11. 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. Feel free to skip sections that look complicated or uninteresting when reading this for the first time.

11.1 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 in this chapter. There is one thing they all have in common: a function is a Lisp object that can be applied to arguments. All of the above objects may 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.

11.2 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, which specifies that the function cell of the symbol is the place to remember the function. We will see all the different types of function spec, 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 don't. The "name" remembered by a function can be any kind of function spec, although it is usually a symbol. In the examples of functions in the previous section, the one 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 didn't remember their names.

To *define a function spec* means to make that function spec remember a given function. Programs do this by calling **fdefine**; 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.

The definition of a function spec can be obtained with **fdefinition**. **(function** *function-spec***)** does so too, but here *function-spec* is not evaluated. For example, **(function foo)** evaluates to the function definition of **foo**. **fdefinition** is used by programs whose purpose is to examine function definitions, whereas **function** is used in this way by programs of all sorts to obtain a specific definition and use it. See page 48.

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** (page 415) and **defselect** (page 236), 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 is 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**, **breakon** 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 on encapsulations, section 11.9, page 244.

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 page 130 for an explanation of function cells and the primitive functions to manipulate them.

(:property *symbol property***)**
: The function is remembered on the property list of the symbol; doing **(get** *symbol property***)** returns 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 operation*)
(:method *flavor-name method-type operation*)
(:method *flavor-name method-type operation suboperation*)

The function is remembered inside internal data structures of the flavor system and is called automatically as part of handling the operation *operation* on instances of *flavor-name*. See the chapter on flavors (chapter 21, page 401) for details.

(:handler *flavor-name operation*)

This is a name for the function actually called when an *operation* 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 (see page 738), breakon (page 741), and advise (page 742).

(:select-method *function-spec operation*)

This function spec assumes that the definition of *function-spec* is a select-method object (see page 232) containing an alist of operation names and functions to handle them, and refers to one particular element of that alist: the one for operation *operation*.

The function is remembered in that alist element and is called when *function-spec*'s definition is called with first argument *operation*.

:select-method function specs are most often used implicitly through defselect. One of the things done by

```
(defselect foo
  (:win (x) (cons 'win x))
  ...)
```

is to define the function spec (:select-method foo :win).

:select-method function specs are explicitly given function definitions when you use defselect-incremental instead of defselect, as in

```
(defselect-incremental foo)
(defun (:select-method foo :win) (ignore x)
  (cons 'win x))
```

(:lambda-macro *name*)

This is a name for the function that expands the lambda macro *name*.

(: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 that there is no other way to describe. This form of function spec isn't 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 section 11.7, page 239).

(: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 on si:rename-within encapsulations (section 11.9.1, page 249) for more information.

It is rarely useful to define a :within function spec by hand, but often useful to trace or advise one. For example,

```
        (breakon '(:within myfunction eval))
```

allows you to break when eval is called from myfunction. Simply doing (breakon 'eval) will probably blow away your machine.

(: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 name of the containing function. *number* is a sequence number; the first internal function the compiler comes across in a given function is numbered 0, the next 1, etc. Internal functions are remembered inside the compiled function object of their containing function.

(:internal *function-spec symbol*)

If a Lisp function uses flet to name an internal function, you can use the local name defined with flet in the :internal function spec instead of a number. Here is an example of such a function:

```
        (defun foo (a)
          (flet ((square (x) (* x x)))
            (+ a (square a))))
```

After compiling foo, you could use the function spec (:internal foo square) to refer to the internal function locally named square. You could also use (:internal foo 0). If there are multiple flet's defining local functions with the same name, only the first can be referred to by name this way.

Here is an example defining a function whose name 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)
```

or

```
    (funcall #'(:property foo bar-maker) 'bax)
```

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 can't apply function specs to arguments.

A keyword symbol that identifies function specs (i.e., that may appear in the car of a list which is a function spec) is identified by a sys:function-spec-handler property whose value is a function that 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 defmacro) 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 that identify a function spec, since that would be ambiguous.

11.3 Simple Function Definitions

defun *Special form*

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. (These keywords are explained in section 3.3, page 38 and other keywords are explained in section 3.3.1, page 43.) See page 234 for some additional syntactic features of defun.

defun creates a list that 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 which 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 applies to the entire function definition; if it is a special declaration, it applies to bindings made in the lambda list and to free references anywhere in the function. For example,

```
        (defun foo (x)
          (declare (special x))
          (bar))                    ;bar uses x free.
```
causes the binding of x to be a dynamic binding, and
```
        (defun foo (&rest args)
          (declare (arglist a b c))
          (apply 'bar args))
```
causes (arglist 'foo) to return (a b c) rather than (&rest args), presumably because the former is more informative in the particular application.

A documentation string can also appear at the beginning of the body: it may precede or follow a declaration. This documentation string becomes part of the function's debugging info and can be obtained with the function documentation (see page 784). 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."
           ...)
```

A documentation string may not be the last element of the body; a string in that position is interpreted as a form to evaluate and return and is not considered to be a documentation string.

For more information on defining functions, and other ways of doing so, see section 11.6, page 234.

11.4 User Operations 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 (see page 528). This works only for interpreted functions. If the definition is a compiled function, it can't be printed out as Lisp code, but its compiled code can be printed by the disassemble function (see page 792).

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 (page 242) to see the argument names and documentation (page 784) to see the documentation string. There are also editor commands for doing these things: the Control-Shift-D and Meta-Shift-D commands are for looking at a function's documentation, and Control-Shift-A is for looking at an argument list.

Control-Shift-A and Control-Shift-D do not ask for the function name; they act on the function that is called by the innermost expression which the cursor is inside. Usually this is the function that will be called by the form you are in the process of writing. They are available in the rubout handler as well.

You can see the function's debugging info alist by means of the function debugging-info (see page 242).

When you are debugging, you can use trace (see page 738) to obtain a printout or a break loop whenever the function is called. You can use breakon (see page 741) to cause the error handler to be entered whenever the function is called; from there, you can step through further function calls and returns. You can customize the definition of the function, either temporarily or permanently, using advise (see page 742).

11.5 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.

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 QFASL 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 that can be applied to arguments, but when they are applied they dig up another function somewhere and apply it instead. These include select-methods, closures, instances, and entities.

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

11.5.1 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, interpreted functions can be printed out and read back in (they have a printed representation that the reader understands), can be pretty-printed (see page 528), and can be 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 section 3.3, page 38 for details. 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 values of the function are the values 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 error handler and other tools to give the user more information. You would not normally write a **named-lambda** yourself; **named-lambda** exists so that **defun** can use it. A **named-lambda** function looks like this:
```
(named-lambda name lambda-list bodyforms...)
```
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**, page 242.) Note that the name need not be a symbol; it can be any function spec. For example,
```
(defun (foo bar) (x)
  (car (reverse x)))
```
gives **foo** a **bar** property whose value is
```
(named-lambda ((:property foo bar)) (x) (car (reverse x)))
```

A **subst** is a function which is open-coded by the compiler. 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*. **subst**'s are described more fully on page 329, with the explanation of **defsubst**.

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**.

11.5.2 Lambda Macros

Lambda macros may appear in functions where **lambda** would have previously appeared. When the compiler or interpreter detects a function whose car is a lambda macro, they expand the macro in much the same way that ordinary Lisp macros are expanded—the lambda macro is called with the function as its argument and is expected to return another function as its value. The definition of a lambda macro (that is, the function which expands it) may be accessed with the (**:lambda-macro** *name*) function spec.

The value returned by the lambda macro expander function may be any valid function. Usually it is a list starting with **lambda**, **subst**, **named-lambda** or **named-subst**, but it could also be another use of a lambda macro, or even a compiled function.

lambda-macro *name lambda-list* &body *body* *Macro*

By analogy with **macro**, defines a lambda macro to be called *name*. *lambda-list* should consist of one variable, which is bound to the function that caused the lambda macro to be called. The lambda macro must return a function. For example:

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

defines a lambda macro called **ilisp** which can be used to define functions that accept arguments like a standard Interlisp function: all arguments are optional and extra arguments are ignored. A typical use would be:

```
(fun-with-functional-arg #'(ilisp (x y z) (list x y z)))
```

This passes to **fun-with-functional-arg** a function which will ignore extra arguments beyond the third, and will default **x**, **y** and **z** to **nil**.

deflambda-macro *Macro*

deflambda-macro is like **defmacro**, but defines a lambda macro instead of a normal macro. Here is how **ilisp** could be defined using **deflambda-macro**:

```
(deflambda-macro ilisp (argument-list &body body)
  `(lambda (&optional ,@argument-list &rest ignore) . ,body))
```

deffunction *function-spec lambda-macro-name lambda-list* &body *body* *Macro*

Defines a function with a definition that uses an arbitrary lambda macro instead of **lambda**. It takes arguments like **defun**, expect that the argument immediatly following the function specifier is the name of the lambda macro to be used. **deffunction** expands the lambda macro immediatly, so the lambda macro must have been previously defined.

Example:

```
(deffunction some-interlisp-like-function ilisp (x y z)
  (list x y z))
```

would define a function called **some-interlisp-like-function** with the definition **(ilisp (x y z) (list x y z))**.

(defun foo ...) could be considered an abbreviation for **(deffunction foo lambda ...)**

11.5.3 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 doesn't 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 micro-compiler.

11.5.4 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 chapter 12, page 250 for more information. Entities are slightly different from closures; see section 12.4, page 255.

A select-method (internal type code dtp-select-method) contains 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 non-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 (see page 236). If the select-method is the definition of a function-spec, the individual functions in the alist can be referred to or defined using :select-method function specs (see page 225).

An instance is a message-receiving object that has both a state and a table of message-handling functions (called *methods*). Refer to the chapter on flavors (chapter 21, page 401) for further information.

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 is for Maclisp compatibility and is not recommended usage. Use aref (page 170) instead.

A stack group can be called as a function. This is one way to pass control to another stack group. See chapter 13, page 256.

11.5.5 Special Forms and Functions

The special forms of Zetalisp, such as **quote**, **let** and **cond**, are actually implemented with an unusual sort of function.

First, let's restate the outline of 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 (see page 24). 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; these are explained in chapter 18, page 320. What about special forms?

A special form is implemented by a function that is flagged to tell the evaluator to refrain from evaluating some or all of the arguments to the function. Such functions make use of the lambda-list keyword **&quote**.

The evaluator, on seeing the **&quote** in the lambda list of an interpreted function (or something equivalent in a compiled function) skips the evaluation of the arguments to which the **&quote** applies. Aside from that, it calls the function normally.

For example, **quote** could be defined as

```
    (defun quote (&quote arg) arg)
```

Evaluation of **(quote x)** would see the **&quote** in the definition, implying that the argument *arg* should not be evaluated. Therefore, the argument passed to the definition of **quote** would be the symbol **x** rather than the value of **x**. From then on, the definition of **quote** would execute in the normal fashion, so **x** would be the value of **arg** and **x** would be returned.

&quote applies to all the following arguments, but it can be cancelled with **&eval**. A simple **setq** that accepted only one variable and one value could be defined as follows:

```
    (defun setq (&quote variable &eval value)
      (set variable value))
```

The actual definition of **setq** is more complicated and uses a lambda list **(&quote &rest variables-and-values)**. Then it must go through the rest-argument, evaluating every other element.

The definitions of special forms are designed with the assumption that they will be called by **eval**. It does not usually make much sense to call one with **funcall** or **apply**. **funcall** and **apply** do not evaluate any arguments; they receive *values* of arguments, rather than expressions for them, and pass these values directly to the function to be called. There is no evaluation for **funcall** or **apply** to refrain from performing. Most special forms explicitly call **eval** on some of their arguments, or parts of them, and if called with **apply** or **funcall** they will *still* do so. This behavior is rarely useful, so calling special forms with **apply** or **funcall** should be avoided. Encapsulations can do this successfully, because they can arrange that quoted arguments are quoted also on entry to the encapsulation.

It is possible to define your own special form using **&quote**. Macros can also be used to accomplish the same thing. 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 in an *ad hoc* way to understand each new special form so that code using it can be compiled. For example, it would not work for a compiled function to call the interpreted definition of **setq**; the **set** in that definition would not be able to act on local variables of the compiled function.

Since all real programs are eventually compiled, writing your own special functions is strongly discouraged. The purpose of **&quote** is to be used in the system's own standard special forms.

New Lisp constructs in the system are also implemented as macros most of the time; macros are less work for us, too.

11.6 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 interface 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 that 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**. See page 239.

All of these function-defining special forms alter only the basic definition of the function spec. Encapsulations are preserved. See section 11.9, page 244.

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 that defines the interpreted function, when processed by the compiler, yields the compiled function. See chapter 17, page 301 for details.

Note that the editor understands these and other "defining" special forms (e.g. **defmethod**, **defvar**, **defmacro**, **defstruct**, etc.) 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 that 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 which expands into a **defun**) are used for creating ordinary interpreted functions (see page 227).

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 (see page 238).

The defsubst special form is used to create substitutible 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 page 329 for full information.

The macro special form is the primitive means of creating a macro. It gives a function spec a definition that 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 chapter 18, page 320.

The defselect special form defines a select-method function. See page 236.

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*, etc. 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 which is not obtainable with the specific defining forms such as defun. For example,

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

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

```
(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.

deff-macro *function-spec definition-creator* *Special form*

Is like deff (see page 235) but for defining macros. *definition-creator* is evaluated to produce a suitable definition-as-a-macro and then *function-spec* is defined that way. The definition-as-a-macro should be a cons whose car is macro and whose cdr is an expander function. Alternatively, a definition as a subst function can be used; either a list starting with subst or named-subst or a FEF which records it was compiled from such a list.

The difference between deff and deff-macro is that compile-file assumes that deff-macro is defining something which should be expanded during compilation. For the rest of the file, the macro defined here is available for expansion. When the file is ultimately loaded, or if compilation is done in-core, deff and deff-macro are equivalent.

@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 that define objects (such as functions) that @ should cross-reference.

si:defun-compatibility *x*

This function is used by defun and the compiler to convert Maclisp-style lexpr, fexpr, and macro defuns to Zetalisp definitions. *x* should be the cdr of a (defun ...) form. defun-compatibility returns a corresponding (defun ...) or (macro ...) form, in the usual Zetalisp format. You shouldn't ever need to call this yourself.

defselect *Special form*

defselect defines a function which 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 *operation*, 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 arguments. defselect is useful for a variety of "dispatching" jobs. By analogy with the more general message-passing facilities described in chapter 21, page 401, the subfunctions are called *methods* and the list of arguments is sometimes called a *message*.

The special form looks like

```
(defselect (function-spec default-handler no-which-operations)
  (operation (args...)
        body...)
  (operation (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 which gets called if the select-method is called with an unknown operation. If *default-handler* is unsupplied or nil, then an unknown operation causes an error with condition name sys:unclaimed-message (see page 423).

Normally, methods for the operations :which-operations, :operation-handled-p and :send-if-handles are generated automatically based on the set of existing methods. These operations have the same meaning as they do on flavor instances; see section 21.10, page 432 for their definitions. If *no-which-operations* is non-nil, these methods are not created automatically; however, you can supply them yourself.

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** are the clauses, each defining one method. *operation* is the operation to be handled by this clause or a list of several operations to be handled by the same clause. *args* is a lambda-list; it should not include the first argument, which is the operation. *body* is the body of the function.

A clause can instead look like:

(*operation* . *symbol*)

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

The individual methods of the **defselect** can be examined, redefined, traced, etc. using **:select-method** function specs (see page 225).

defselect-incremental *function-spec default-handler* *Special form*

defselect defines a select-method function all at once. By contrast, **defselect-incremental** defines an empty select-method to which methods can be added with **defun**.

Specifically, **defselect-incremental** *function-spec*, with just a default handler and the standard methods **:which-operations**, **:operation-handled-p** and **:send-if-handles**.

Individual methods are defined by using **defun** on a function spec of the form (**:select-method** *function-spec operation*). *function-spec* specifies where to find the select-method, and *operation* is the operation for which a method should be defined. The argument list of the **defun** must include a first argument which receives the operation name.

Example:

```
(defselect-incremental foo ignore)
    ;The function ignore is the default handler
(defun (:select-method foo :lose) (ignore a)
  (1+ a))
```

defines the same function **foo** as

```
(defselect (foo ignore)
  (:lose (a) (1+ a)))
```

These two examples are not completely equivalent, however. Reevaluating the **defselect** gets rid of any methods that used to exist but have been deleted from the **defselect** itself. Reevaluating the **defselect-incremental** has no such effect, and reevaluating an individual **defun** redefines only that method. Methods can be removed only with **fundefine**.

11.6.1 Maclisp Lexprs

Lexprs are the way Maclisp functions can accept variable numbers of arguments. They are supported for compatibility only; &optional and &rest are much preferable. A lexpr definition looks like

```
(defun foo nargs body...)
```

where a symbol (nargs, here) appears in place of a lambda-list. When the function is called, nargs is bound to the number of arguments it was given. The arguments themselves are accessed using the functions arg, setarg, and listify.

arg *x*

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

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

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

setarg *i x*

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

listify *n*

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

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

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

11.7 How Programs Manipulate Function Specs

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

This is the primitive used by **defun** and everything else in the system 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**, page 241), 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 warning 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. This property is used by the **undefun** function (page 241), which restores the previous 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 which use **nil** for *carefully*.

si:record-source-file-name *name* &optional (*type* defun) *no-query*

Records a definition of *name*, of type *type*. *type* should be **defun** to record a function definition; then *name* is a function spec. *type* can also be **defvar**, **defflavor**, **defresource**, **defsignal** or anything else you want to use.

The value of **sys:fdefine-file-pathname** is assumed to be the generic pathname of the file the definition is coming from, or **nil** if the definition is not from a file. If a definition of the same *name* and *type* has already been seen but not in the same file, and *no-query* is **nil**, a condition is signaled and then the user is queried.

If **si:record-source-file-name** returns **nil**, it means that the user or a condition handler said the redefinition should not be performed.

sys:fdefine-file-pathname *Variable*

While the system is 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.

si:get-source-file-name *function-spec* &optional *type*

Returns the generic pathname for the file in which *function-spec* received a definition of type *type*. If *type* is **nil**, the most recent definition is used, regardless of its type.

function-spec really is a function spec only if *type* is **defun**; for example, if *type* is **defvar**, *function-spec* is a variable name. Other types that are used by the system are **defflavor** and **defstruct**.

This function returns the generic pathname of the source file. To obtain the actual source file pathname, use the :source-pathname operation (see page 563).

A second value is returned, which is the type of the definition that was reported.

si:get-all-source-file-names *function-spec*

Returns a list describing the generic pathnames of all the definitions this function-spec has received, of all types. The list is an alist whose elements look like

```
    (type pathname...)
```

sys:redefinition (sys:warning) *Condition Flavor*

This condition, which is not an error, is signaled by si:record-source-file-name when something is redefined by a different file. The handler for this condition can control what is done about the redefinition.

The condition instance provides the operations :name, :definition-type, :old-pathname and :new-pathname. :name and :definition-type return the *name* and *type* arguments to si:record-source-file-name. :old-pathname and :new-pathname return two generic pathnames saying where the old definition was and where this one is. The new pathname may be nil, meaning that the redefinition is being done by the user, not in any file.

Two proceed types are available, :proceed and :inhibit-definition. The first tells si:record-source-file-name to return t, the second tells it to return nil. If the condition is not handled at all, the user is queried or warned according to the value of inhibit-fdefine-warnings.

inhibit-fdefine-warnings *Variable*

This variable is normally nil. Setting it to t prevents si:record-source-file-name from warning you and asking about questionable redefinitions 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', i.e. that it is all right to redefine the function.

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

This function is obsolete. It is equivalent to

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

fdefinedp *function-spec*

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

fdefinition *function-spec*

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

fdefinition-location *function-spec*

Equivalent to (locf (fdefinition *function-spec*)). For some kinds of function specs, though not for symbols, this (whichever way you write it) 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.

fundefine *function-spec*

Makes *function-spec* undefined; the cell where its definition is stored becomes void. 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 flavor chapter).

undefun *function-spec*

If *function-spec* has a saved previous basic definition, this interchanges the current and previous basic definitions, leaving the encapsulations alone. If *function-spec* has no saved previous definition, **undefun** asks the user whether to make it undefined.

This undoes the effect of redefining a function. See also **uncompile** (page 301).

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

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*

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

si:function-spec-lessp *function-spec1 function-spec2*

Compares the two function specs with an ordering that is useful in sorting lists of function specs for presentation to the user.

si:function-parent *function-spec*

If *function-spec* does not have its own definition, textually speaking, but is defined as part of the definition of something else, this function returns the function spec for that something else. For example, if *function-spec* is an accessor function for a **defstruct**, the value returned is the name of the **defstruct**.

The intent is that if the caller has not been able to find the definition of *function-spec* in a more direct fashion, it can try looking for the definition of the *function-parent* of *function-spec*. This is used by the editor's **Meta-.** command.

sys:invalid-function-spec (**error**) *Condition*

This condition name belongs to the error signaled when you refer to a function spec that is syntactically invalid; such as, if it is a list whose car is not a recognized type of function spec.

The condition object supports the operation **:function-spec**, which returns the function spec which was invalid.

Note that in a few cases the condition **:wrong-type-argument** is signaled instead. These are the cases in which the error is correctable.

11.8 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).

The function **documentation** can be used to examine a function's documentation string. See page 784.

debugging-info *function*

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

arglist *function* &optional *real-flag*

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.

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 elements 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 has one of three values:

nil	Return the arglist declared by the user in preference to the actual one.
t	Return the actual arglist as computed from the function definition's handling of arguments, ignoring any **arglist** declaration. For a compiled function, this omits all keyword arguments (replacing them with a rest argument) and may replace initial values of optional arguments with **si:*hairy*** if the actual expressions are too complicated.
compile	Like **nil**, but in the case of a compiled function it returns the actual arglist of the lambda-expression that was originally compiled. The compiler uses this as a basis for checking for incorrect calls to the function.

Programs interested in how many and what kind (evaluated or quoted) of arguments to pass should use **args-info** instead.

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 is then returned by **arglist** as the second value.

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

function-name *function* &optional *try-flavor-name*

Returns the name of the function *function*, if that can be determined. If *function* does not describe what its name is, *function* itself is returned.

If *try-flavor-name* is non-nil, then if *function* is a flavor instance (which can, after all, be used as a function), then the flavor name is returned. If the optional argument is **nil**, flavor instances are treated as anonymous.

eh:arg-name *function arg-number*

Returns the name of argument number *arg-number* in function *function*. Returns **nil** if the function doesn't have such an argument, or if the name is not recorded. **&rest** arguments are not obtained with **arg-number**; use **rest-arg-name** to obtain the name of *function*'s **&rest** argument, if any.

eh:rest-arg-name *function*

Returns the name of the rest argument of function *function*, or **nil** if *function* does not have one.

eh:local-name *function local-number*

Returns the name of local variable number *local-number* in function *function*. If *local-number* is zero, this gets the name of the rest arg in any function that accepts a rest arg. Returns **nil** if the function doesn't have such a local.

args-info *function*

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 specifiers, 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, i.e. the number of required parameters.

%%arg-desc-max-args
: This is the maximum number of arguments that may be passed to this function, i.e. 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 does not return this bit, although %args-info does.

%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*
: 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.

11.9 Encapsulations

The definition of a function spec actually has two parts: the *basic definition*, and *encapsulations*. The basic definition is what is created by functions like defun, 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 (see page 738), breakon (see page 741) and advise (see page 742).

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 which 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 which is not an encapsulation; it does not have the sort of debugging info item which 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 which identifies the purpose of the encapsulation and says what the application is. For example, that 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 **si:encapsulation-standard-order**, page 247). *type* should have an **si:encapsulation-grind-function** property which tells **grindef** what to do with an encapsulation of this type.

body-form 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 section 18.2.2, page 325. **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 which the si:encapsulate macro produces automatically. When the body of the encapsulation is run arglist's value will be the list of the arguments which 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, which every encapsulation must have). Some applications find this useful for recording information about the encapsulation for their own later use.

If compile-encapsulations-flag is non-nil, the encapsulation is compiled before it is installed. The encapsulations on a particular function spec can be compiled by calling compile-encapsulations. See page 302. Compiled encapsulations can still be unencapsulated since the information needed to do so is stored in the debugging info alist, which is preserved by compilation. However, applications which wish to modify the code of the encapsulations they previously created must check for encapsulations that have been compiled and uncompile them. This can be done by finding the sys:interpreted-definition entry in the debugging info alist, which is present in all compiled functions except those made by file-to-file compilation.

When a special function is encapsulated, the encapsulation is itself a special function with the same argument quoting pattern. 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 execution of the expansion itself. The value returned by the encapsulation is the expansion of the macro call, not the value computed by the expansion.

A program which creates encapsulations often needs to examine an encapsulation it created and find the body. For example, adding a second piece of advice to one function requires doing this. The proper way to do it is to use si:encapsulation-body.

si:encapsulation-body *encapsulation*

Returns a list whose car is the body-form of *encapsulation*. It is the form that was the fourth argument of si:encapsulate when *encapsulation* was created. To illustrate this relationship,

```
    (si:encapsulate 'foo 'foo 'trace 'body))

    (si:encapsulation-body (fdefinition 'foo))
      => (body)
```

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 in which the encapsulations are supposed to be kept (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 breakon trace si:rename-within)
```

advise encapsulations are used to hold advice (see page 742). breakon encapsulations are used for implementing breakon (see page 741). trace encapsulations are used for implementing tracing (see page 738). 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 section 11.9.1, page 249 for more information).

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 which the user ought to see. Refer to the code for the grind function for advise to see how to write one.

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*

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 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 variable **si:encapsulation-standard-order**, which is used by **si:unencapsulate-function-spec**, knows the order.

11.9.1 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 (see page 528) 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*

Given *new-structure*, which 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.

12. 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 dynamic bindings of specified variables and then to refer to those bindings later, even after the construct (**let**, etc.) which made the bindings has been exited.

12.1 What a Closure Is

There is a view of dynamic variable binding that we use in this section because it makes it easier to explain what closures do. In this view, when a variable is bound dynamically, a new binding is created for it. The old binding is saved away somewhere and is inaccessible. Any references to the variable then get the contents of the new binding, and any **setq**'s change the contents of the new value cell. Evantually the new binding goes away, and the old binding, along with its contents, becomes current again.

For example, consider the following sequence of Lisp forms:

```
(defvar a 3)             ; a becomes 3.

(let ((a 10))            ; a rebound to 10.
  (print (+ a 6)))       ; 16 is printed.

(print a)                ; 3 is printed.
```

Initially there is a binding for **a**, and the **setq** form makes the contents of that binding be **3**. Then the **lambda**-combination is evaluated. **a** is bound to **10**: the old binding, which still contains **3**, is saved away, and a new binding 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 binding, namely **10**. So **16** is printed. Then the newer binding is discarded and the old binding, which still contains a **3**, is restored. The final **print** prints **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 of the bindings of the variables on *var-list* are saved away, and the bindings that those variables had *at the time* **closure** *was called* (that is, at the time the closure was created) are made to be the bindings of the symbols. Then *function* is applied to the arguments. (This paragraph is somewhat complex, but it completely describes the operation of closures; if you don't understand it, come back and read it again after reading the next two paragraphs.)

Here is another, lower-level explanation. The closure object stores several things inside it. First, it saves the *function*. Secondly, for each variable in *var-list*, it remembers what that variable's binding was when the closure was created. Then when the closure is called as a function, it first temporarily restores the bindings 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 bindings are restored to be as they were before the closure was called.

Now, if we evaluate the form

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

what happens is that a new binding is created for x, containing a fixnum 3. Then a closure is created, which remembers the function frob, the symbol x, and that binding. Finally the old binding of x is restored, and the closure is returned. Notice that the new binding is still around, because it is still known about by the closure. When the closure is applied, say by doing (funcall a 7), this binding is temporarily restored and the value of x is 3 again. If frob uses x as a free variable, it sees 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 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, use function, as in:

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

Explicit closures made with closure record only the dynamic bindings of the specified variables. Another closure mechanism is activated automatically to record lexical bindings whenever function is used around an explicit lambda expression, but closure itself has no interaction with lexical bindings.

It is the user's responsibility to make sure that the bindings that the closure is intended to record are dynamic bindings, either by means of special declarations (see page 51) as shown above or by making the variables globally special with defvar or equivalent. If the function closed over is an explicit lambda expression, it is occasionally necessary to use declarations within it to make sure that the variables are considered special there. But this is not needed if the variables are globally special or if a special declaration is lexically visible where closure is called.

Usually the compiler can tell when a special declaration is missing, but when making a closure the compiler detects this only after acting on the assumption that the variable is lexical, by which time it is too late to fix things. The compiler warns you if this happens.

In Zetalisp's implementation of closures, lambda-binding never really allocates any storage to create new bindings. Bindings receive separate storage only when the closure function itself finds they need it. Thus, there is no cost associated with closures when they are not in use.

Zetalisp closures differ from the closures of Lisp 1.5 (which were made with function) in that they save specific variables rather than the entire variable-binding environment. For their intended applications, this is an advantage. The explicit declaration of the variables in closure permits higher efficiency and more flexibility. In addition the program is clearer because the intended effect of the closure is made manifest by listing the variables to be affected. Lisp 1.5 closures are more similar to Zetalisp's automatic handling of lexical variables.

Closure implementation (which it not usually necessary for you to understand) involves two kinds of value cells. Every symbol has an *internal value cell*, part of the symbol itself, which is where its dynamic value is normally stored. When a variable is closed over, it 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**, etc.), because the internal value cell is made to contain a forwarding pointer to the external value cell that is current. Such a forwarding pointer is present in a symbol's value cell whenever its current binding is being remembered by some closure; at other times, there won't be an invisible pointer, and the value resides directly in the symbol's internal value cell.

12.2 Examples of the Use of Closures

One thing we can do with closures is to implement a *generator*, which is a kind of function which is called successively to obtain successive elements of a sequence. We implement a function **make-list-generator**, which takes a list and returns a generator that returns 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 to solve this problem using closures:

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

(make-list-generator '(1 2 3)) returns a generator which, on successive calls, returns 1, 2, 3, and **nil**.

Now we can make as many list generators as we like; they won't get in each other's way because each has its own binding for **l**. Each of these bindings was created when the **make-list-generator** function was entered, and the bindings are remembered by the closures.

The following example uses closures which share bindings:

```
(defvar a)
(defvar 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 bindings are created for the symbols **a** and **b**, and two closures are created that both point to those bindings. If we do **(funcall x)**, the function **foo** is be run, and it changes the contents of the remembered binding of **a** to **5**. If we then do **(funcall y)**, the function **bar** returns **(5 . 1)**. This shows that the binding of **a** seen by the closure **y** is the same binding seen by the closure **x**. The top-level binding of **a** is unaffected.

Here is how we can create a function that prints always using base 16:

```
(deff print-in-base-16
      (let ((*print-base* 16.))
        (closure '(*print-base*) 'print)))
```

12.3 Closure-Manipulating Functions

closure *var-list function*

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 page 13) or **(typep** *object* **'closure)**.

symeval-in-closure *closure symbol*

Returns the binding of *symbol* in the environment of *closure*; that is, it returns what you would get if you restored the bindings 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**.

symbol may be a locative pointing to a value cell instead of a symbol (this goes for all the whatever-in-closure functions).

set-in-closure *closure symbol x*

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

locate-in-closure *closure symbol*

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*)).

boundp-in-closure *closure symbol*

Returns t if *symbol*'s binding in *closure* is not void. This is what (boundp *symbol*) would return if executed in *closure*'s saved environment.

makunbound-in-closure *closure symbol*

Makes *symbol*'s binding in *closure* be void. This is what (makunbound *symbol*) would do if executed in *closure*'s saved environment.

closure-alist *closure*

Returns an alist of (*symbol* . *value*) pairs describing the bindings that 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 does not change the closure.

The list that is returned may contain void cells if some of the closed-over variables were void in the closure's environment. In this case, printing the value will get an error (accessing a cell that contains a void marker is always an error unless done in a special, careful way) but the value can still be passed around.

closure-variables *closure*

Returns a list of variables closed over in *closure*. This is equal to the first argument specified to the function closure when this closure was created.

closure-function *closure*

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

closure-bindings *closure*

Returns the actual list of bindings to be performed when *closure* is entered. This list can be passed to sys:%using-binding-instances to enter the closure's environment without calling the closure. See page 287.

copy-closure *closure*

Returns a new closure that has the same function and variable values as *closure*. The bindings are not shared between the old closure and the new one, so that if the old closure changes some closed variable's values, the values in the new closure do not change.

let-closed ((*variable value*)...) *function* *Macro*

When using closures, it is very common to bind a set of variables with initial values only in order to make a closure over those variables. Furthermore, the variables must be declared special. let-closed is a special form which does all of this. It is best described by example:

```
(let-closed ((a 5) b (c 'x))
   (function (lambda () ...)))
```
macro-expands into
```
(let ((a 5) b (c 'x))
  (declare (special a b c))
  (closure '(a b c)
    (function (lambda () ...))))
```

Note that the following code, which would often be useful, does not work as intended if x is not special outside the let-closed:
```
(let-closed ((x x))
  (function ...))
```
This is because the reference to x as an initialization for the new binding of x is affected by the special declaration that the let-closed produces. It therefore does not see any lexical binding of x. This behavior is unfortunate, but it is required by the Common Lisp specifications. To avoid the problem, write
```
(let ((y x))
  (let-closed ((x y))
    (function ...)))
```
or simply change the name of the variable outside the let-closed to something other than x.

12.4 Entities

An entity is almost the same thing as a closure; an entity behaves just like a closure when applied, but it has a recognizably different data type which allows certain parts of the system such as the printer and describe to treat it differently. A closure is simply a kind of function, but an entity is assumed to be a message-receiving object. Thus, when the Lisp printer (see section 23.1, page 506) 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 (see page 791) also sends entities messages when it is handed them. So when you want to make a message-receiving object out of a closure, as described on page 407, you should use an entity instead.

To a large degree, entities are made obsolete by flavors (see chapter 21, page 401). Flavors have had considerably more attention paid to their efficiency and to good tools for using them. If what you are doing is flavor-like, it is better to use flavors.

entity *variable-list function*

Returns a newly constructed entity. This function is just like the function closure except that it returns an entity instead of a closure.

The *function* argument should be a symbol which has a function definition and a value. When typep is applied to this entity, it returns the value of that symbol.

To test whether an object is an entity, use the entityp predicate (see page 13). The functions symeval-in-closure, closure-alist, closure-function, etc. also operate on entities.

13. Stack Groups

A *stack group* (usually abbreviated 'SG') is a type of Lisp object useful for implementation of certain advanced control structures such as coroutines and generators. Processes, which are a kind of coroutine, are built on top of stack groups (see chapter 29, page 682). A stack group represents a computation and its internal state, including the Lisp stack.

At any time, the computation being performed by the Lisp Machine is associated with one stack group, called the *current* or *running* stack group. The operation of making some stack group be the current stack group is called a *resumption* or a *stack group switch*; the previously running stack group is said to have *resumed* the new stack group. The *resume* operation has two parts: first, the state of the running computation is saved away inside the current stack group, and secondly the state saved in the new stack group is restored, and the new stack group is made current. Then the computation of the new stack group resumes its course.

The stack group itself holds a great deal of state information. It contains the control stack, or *regular PDL*. The control stack is what you are shown by the backtracing commands of the error handler (Control-B, Meta-B, and Control-Meta-B); it remembers the function which is running, its caller, its caller's caller, etc., and the point of execution of each function (the *return address* of each function). A stack group also contains the dynamic environment stack, or *special PDL*. The name 'stack group' derives from the existence of these two stacks. Finally, the stack group contains various internal state information (contents of machine registers and so on).

When the stack group is running, the special PDL contains all the dynamic bindings that are shadowed by other bindings in this stack group; bindings that are current reside in the symbols' value cells. When the stack group is not running, all of the dynamic bindings it has made reside in its special PDL. Switching to a stack group moves the current bindings from the special PDL to the symbol value cells, exchanging them with the global or other shadowed bindings. Switching out of a stack group does the reverse process. Note that unwind-protect handlers are *not* run by a stack-group switch (see let-globally, page 32).

Each stack group is a separate environment for purposes of function calling, throwing, dynamic variable binding, and condition signalling. All stack groups run in the same address space; thus they share the same Lisp data and the same global (not lambda-bound) variables.

When a new stack group is created, it is empty: it doesn't contain the state of any computation, so it can't be resumed. In order to get things going, the stack group must be set to an initial state. This is done by *presetting* the stack group. To preset a stack group, you supply a function and a set of arguments. The stack group is placed in such a state that when it is first resumed it will apply this function to those arguments. The function is called the *initial function* of the stack group.

13.1 Resuming of Stack Groups

The interesting thing that happens to stack groups is that they resume each other. When one stack group resumes a second stack group, the current state of Lisp execution is saved away in the first stack group and is restored from the second stack group. Resuming is also called *switching stack groups*.

At any time, there is one stack group associated with the current computation; it is called the current stack group. The computations associated with other stack groups have their states saved away in memory and are not computing. So the only stack group that can do anything at all, in particular resuming other stack groups, is the current one.

You can look at things from the point of view of one computation. Suppose it is running along, and it resumes some stack group. The state of the computation state is saved away into its own stack group, and the computation associated with the called stack group starts up. The original computation lies dormant in the original stack group, while other computations go around resuming each other, until finally the original stack group is resumed by someone. Then the computation is restored from the stack group and gets to run again.

There are several ways that the current stack group can resume other stack groups. This section describes all of them.

Each stack group records a *resumer* which is **nil** or another stack group. Some forms of resuming examine and alter the resumer of some stack groups.

Resuming has another ability: it can transmit a Lisp object from the old stack group to the new stack group. Each stack group specifies a value to transmit whenever it resumes another stack group; whenever a stack group is resumed, it receives a value.

In the descriptions below, let *c* stand for the current stack group, *s* stand for some other stack group, and *x* stand for any arbitrary Lisp object.

Stack groups can be used as functions. They accept one argument. If *c* calls *s* as a function with one argument *x*, then *s* is resumed, and the object transmitted is *x*. When *c* is resumed (usually—but not necessarily—by *s*), the object transmitted by that resumption is returned as the value of the call to *s*. This is one of the simple ways to resume a stack group: call it as a function. The value you transmit is the argument to the function, and the value you receive is the value returned from the function. Furthermore, this form of resuming sets *s*'s resumer to be *c*.

Another way to resume a stack group is to use **stack-group-return**. Rather than allowing you to specify which stack group to resume, this function always resumes the resumer of the current stack group. Thus, this is a good way to go back to the stack group which called the current one, assuming that this was done through a function call. **stack-group-return** takes one argument which is the object to transmit. It returns when something resumes the current stack group, and returns one value, the object that was transmitted by that resumption. **stack-group-return** does not change the resumer of any stack group.

The most fundamental way to do resuming is with stack-group-resume, which takes two arguments: the stack group, and a value to transmit. It returns when someone resumes the current stack group, returning the value that was transmitted by that resumption, and does not affect any stack group's resumer.

If the initial function of *c* attempts to return a value *x*, the regular kind of Lisp function return cannot take place, since the function did not have any caller (it got there when the stack group was initialized). So instead of normal function returning, a "stack group return" happens. *c*'s resumer is resumed, and the value transmitted is *x*. *c* is left in a state ("exhausted") from which it cannot be resumed again; any attempt to resume it signals an error. Presetting it will make it work again.

Those are the "voluntary" forms of stack group switch; a resumption happens because the computation said it should. There are also two "involuntary" forms, in which another stack group is resumed without the explicit request of the running program.

If an error occurs, the current stack group resumes the error handler stack group. The value transmitted is partially descriptive of the error, and the error handler looks inside the saved state of the erring stack group to get the rest of the information. The error handler recovers from the error by changing the saved state of the erring stack group and then resuming it.

When certain events occur, typically a 1-second clock tick, a *sequence break* occurs. This forces the current stack group to resume a special stack group called the *scheduler* (see section 29.1, page 683). The scheduler implements processes by resuming, one after another, the stack group of each process that is ready to run.

current-stack-group-resumer *Variable*

Is the resumer of the current stack group.

current-stack-group *Variable*

Is the stack group which is currently running. A program can use this variable to get its hands on its own stack group.

13.2 Stack Group States

A stack group has a *state*, which controls what it will do when it is resumed. The code number for the state is returned by the function sys:sg-current-state. This number is the value of one of the following symbols. Only the states actually used by the current system are documented here; some other codes are defined but not used.

sys:sg-state-active
The stack group is the current one.

sys:sg-state-resumable
The stack group is waiting to be resumed, at which time it will pick up its saved machine state and continue doing what it was doing before.

sys:sg-state-awaiting-return
The stack group called some other stack group as a function. When it is resumed, it will return from that function call.

sys:sg-state-awaiting-initial-call
: The stack group has been preset (see below) but has never been called. When it is resumed, it will call its initial function with the preset arguments.

sys:sg-state-exhausted
: The stack group's initial function has returned. It cannot be resumed.

sys:sg-state-awaiting-error-recovery
: When a stack group gets an error it goes into this state, which prevents anything from happening to it until the error handler has looked at it. In the meantime it cannot be resumed.

sys:sg-state-invoke-call-on-return
: When the stack group is resumed, it will call a function. The function and arguments are already set up on the stack. The debugger uses this to force the stack group being debugged to do things.

13.3 Stack Group Functions

make-stack-group *name* &optional *options*

Creates and returns a new stack group. *name* may be any symbol or string; it is used in the stack group's printed representation. *options* is a list of alternating keywords and values. The options are not too useful; most calls to make-stack-group don't need any options at all. The options are:

:sg-area
: The area in which to create the stack group structure itself. Defaults to the default area (the value of default-cons-area).

:regular-pdl-area
: The area in which to create the regular PDL. Only certain areas specially designated when they were created may be used for regular PDLs, because regular PDLs are cached in a hardware device called the *pdl buffer*. The default is sys:pdl-area.

:special-pdl-area
: The area in which to create the special PDL. Defaults to the default area (the value of default-cons-area).

:regular-pdl-size
: Length of the regular PDL to be created. Defaults to 3000 octal.

:special-pdl-size
: Length of the special PDL to be created. Defaults to 2000 octal.

:swap-sv-on-call-out
:swap-sv-of-sg-that-calls-me
: These flags default to 1. If these are 0, the system does not maintain separate binding environments for each stack group. You do not want to use this feature.

:trap-enable
: This determines what to do if a microcode error occurs. If it is 1 the system tries to handle the error; if it is 0 the machine halts. Defaults to

1. It is 0 only in the error handler stack group, a trap in which would not work anyway.

:safe — If this flag is 1 (the default), a strict call-return discipline among stack-groups is enforced. If 0, no restriction on stack-group switching is imposed.

sys:pdl-overflow (error) *Condition*

This condition is signaled when there is overflow on either the regular pdl or the special pdl. The :pdl-name operation on the condition instance returns either :special or :regular, to tell handlers which one.

The :grow-pdl proceed type is provided. It takes no arguments. Proceeding from the error automatically makes the affected pdl bigger.

eh:pdl-grow-ratio *Variable*

This is the factor by which to increase the size of a pdl after an overflow. It is initially 1.5.

eh:require-pdl-room *regpdl-space specpdl-space*

Makes the current stack group larger if necessary, to make sure that there are at least *regpdl-space* free words in the regular pdl, and at least *specpdl-space* free words in the special pdl, not counting the words currently in use.

stack-group-preset *stack-group function* &rest *arguments*

This sets up *stack-group* so that when it is resumed, *function* will be applied to *arguments* within the stack group. Both stacks are made empty; all saved state in the stack group is destroyed. stack-group-preset is typically used to initialize a stack group just after it is made, but it may be done to any stack group at any time. Doing this to a stack group which is not exhausted destroys its present state without properly cleaning up by running unwind-protects.

stack-group-resume *s x*

Resumes *s*, transmitting the value *x*. No stack group's resumer is affected.

si:sg-resumable-p *s*

t if *s*'s state permits it to be resumed.

sys:wrong-stack-group-state (error) *Condition*

This is signaled if, for example, you try to resume a stack group which is in the exhausted state.

stack-group-return *x*

Resumes the current stack group's resumer, transmitting the value *x*. No stack group's resumer is affected.

symeval-in-stack-group *symbol sg* &optional *frame as-if-current*

Evaluates the variable *symbol* as a special variable in the binding environment of *sg*. If *frame* is not **nil**, it evaluates *symbol* in the binding environment of execution in that frame. (A frame is an index in the stack group's regular pdl).

Two values are returned: the symbol's value, and a locative to where the value is stored. If *as-if-current* is not **nil**, the locative points to where the value *would* be stored if *sg* were running. This may be different from where the value is stored now; for example, the current binding in stack group *sg* is stored in *symbol*'s value cell when *sg* is running, but is probably stored in *sg*'s special pdl when *sg* is not running. *as-if-current* makes no difference if *sg* actually *is* the current stack group.

If *symbol*'s current dynamic binding in the specified stack group and frame is void, this signals a **sys:unbound-variable** error.

13.4 Analyzing Stack Frames

A stack frame is represented by an index in the regular pdl array of the stack group. The word at this index is the function executing, or to be called, in the frame. The following words in the pdl contain the arguments.

sg-regular-pdl *sg*

Returns the regular pdl of *sg*. This is an array of type **art-reg-pdl**. Stack frames are represented as indices into this array.

sg-regular-pdl-pointer *sg*

Returns the index in *sg*'s regular pdl of the last word pushed.

sg-special-pdl *sg*

Returns the special pdl of *sg*. This is an array of type **art-special-pdl**, used to hold special bindings made by functions executing in that stack group.

sg-special-pdl-pointer *sg*

Returns the index in *sg*'s special pdl of the last word pushed.

The following functions are used to move from one stack frame to another.

eh:sg-innermost-active *sg*

Returns (the regular pdl index of) the innermost frame in *sg*, the one that would be executing if *sg* were current. If *sg* is current, the value is the frame of the caller of this function.

eh:sg-next-active *sg frame*

Returns the next active frame out from *frame* in *sg*. This is the one that called *frame*. If *frame* is the outermost frame, the value is *nil*.

eh:sg-previous-active *sg frame*
Returns the previous active frame in from *frame* in *sg*. This is the one called by *frame*. If *frame* is the currently executing frame, the value is nil. If *frame* is nil, the value is the outermost or initial frame.

eh:sg-innermost-open *sg*
Returns the innermost open frame in *sg*, which may be the same as the innermost active one or it may be within that. In other respects, this is like eh:sg-innermost-active.

eh:sg-next-open *sg frame*
Like eh:sg-next-active but includes frames which are *open*, that is, still accumulating arguments prior to calling the function.

eh:sg-previous-open *sg frame*
Like eh:sg-previous-active but includes frames which are *open*, that is, still accumulating arguments prior to calling the function.

eh:sg-frame-active-p *sg frame*
Returns t if *frame* is active; that is, if the function has been entered.

Running interpreted code involves calls to eval, cond, etc. which would not be there in compiled code. The following three functions can be used to skip over the stack frames of such functions, showing only the frames for the functions the user would know about.

eh:sg-next-interesting-active *sg frame*
Like eh:sg-next-active but skips over uninteresting frames.

eh:sg-previous-interesting-active *sg frame*
Like eh:sg-previous-active but skips over uninteresting frames.

eh:sg-out-to-interesting-active *sg frame*
If *frame* is interesting, returns *frame*. Otherwise, it returns the next interesting active frame.

Functions to analyze the data in a particular stack frame:

sys:rp-function-word *regpdl frame*
Returns the function executing in *frame*. *regpdl* should be the sg-regular-pdl of the stack group.

eh:sg-frame-number-of-spread-args *sg frame*
Returns the number of arguments received by *frame*, which should be an active frame. The rest argument (if any) and arguments received by it, do not count.

eh:sg-frame-arg-value *sg frame n*
Returns the value of argument number *n* of stack frame *frame* in *sg*. An error is signaled if *n* is out of range, if the frame is active. (For an open frame, the number of arguments is not yet known, so there is no error check.)

The second value is the location in which the argument is stored when *sg* is running. The location may not actually be in the stack, if the argument is special. The location may then contain other contents when the stack group is not running.

eh:sg-frame-rest-arg-value *sg frame*
Returns the value of the rest argument in *frame*, or nil if there is none.

The second value is t if the function called in *frame* expects an explicitly passed rest argument.

The third value is t if the rest argument was passed explicitly. If this is nil, the rest arg is a stack list that overlaps the arguments of stack frame *frame*. If it was passed explicitly, it may still be a stack list, but not in this frame. See section 5.9, page 112 for more information on stack lists.

eh:sg-frame-number-of-locals *sg frame*
Returns the number of local variables in stack frame *frame*.

eh:sg-frame-local-value *sg frame n*
Returns the value of local variable number *n* of stack frame *frame* in *sg*. An error is signaled if *n* is out of range.

The second value is the location in which the local is stored when *sg* is running. The location may not actually be in the stack; if not, it may have other contents when the stack group is not running.

eh:sg-frame-value-value *sg frame n* &optional *create-slot*
Returns the value and location of the *n*'th multiple value *frame* has returned. If *frame* has not begun to return values, the first value returned is nil but the location still validly shows where value number *n* will be stored.

If *frame* was called with multiple-value-list, it can return any number of values, but they do not have cells to receive them until *frame* returns them. In this case, a non-nil *create-slot* means that this function should allocate cells as necessary so that a valid location can be returned. Otherwise, the location as well as the value is nil.

eh:sg-frame-value-list *sg frame* &optional *new-number-of-values*
Returns three values that describe whether *frame*'s caller wants multiple values, and any values *frame* has returned already.

The first value is a list in which live the values being, or to be, returned by *frame*.

The second value is nil if this frame has not been invoked to return multiple values, a number which is the number of values it has been asked for, or a locative, meaning the frame was called with multiple-value-list. In the last case, the first value includes only the values *frame* has returned already, and the locative points to a cell that points to the cons whose cdr should receive the next link of the list.

The third value is how many values *frame* has returned so far.

If *new-number-of-values* is non-nil, it is used to alter the "number of values already returned" as recorded in the stack group. This may alter the length of the list that is the first value. The value you get is the altered one, in that case.

eh:sg-frame-special-pdl-range *sg frame*
Returns two values delimiting the range of *sg*'s special pdl that belongs to the specified stack frame. The first value is the index of the first special pdl word that belongs to the frame, and the second value is the index of the next word that does not belong to it.

If the specified frame has no special bindings, both values are nil. Otherwise, the indicated special pdl words describe bindings made on entry to or during execution in this frame. The words come in pairs.

The first word of each pair contains the saved value; the second points to the location that was bound. When the stack group is not current, the saved value is the value for the binding made in this frame. When the stack group is current, the saved value is the shadowed value, and the value for this binding is either in the cell that was bound, or is the saved value of another binding, at a higher index, of the same cell.

The bit sys:%%specpdl-closure-binding is nonzero in the first word of the pair if the binding was made before entry to the function itself. This includes bindings made by closures, and by instances (including self). Otherwise, the binding was made by the function itself. This includes arguments that are declared special.

symeval-in-stack-group can be used to find the value of a special variable at a certain stack frame (page 261).

13.5 Input/Output in Stack Groups

Because each stack group has its own set of dynamic bindings, a stack group does not inherit its creator's value of *terminal-io* (see page 460), nor its caller's, unless you make special provision for this. The *terminal-io* a stack group gets by default is a "background" stream that does not normally expect to be used. If it is used, it turns into a "background window" that will request the user's attention. Often this happens when an error invokes the debugger.

If you write a program that uses multiple stack groups, and you want them all to do input and output to the terminal, you should pass the value of *terminal-io* to the top-level function of each stack group as part of the stack-group-preset, and that function should bind the variable *terminal-io*.

Another technique is to use a closure as the top-level function of a stack group. This closure can bind *terminal-io* and any other variables that should be shared between the stack group and its creator.

13.6 An Example of Stack Groups

The canonical coroutine example is the so-called samefringe problem: Given two trees, determine whether they contain the same atoms in the same order, ignoring parenthesis structure. A better way of saying this is, given two binary trees built out of conses, determine whether the sequence of atoms on the fringes of the trees is the same, ignoring differences in the arrangement of the internal skeletons of the two trees. Following the usual rule for trees, **nil** in the cdr of a cons is to be ignored.

One way of solving this problem is to use *generator* coroutines. We make a generator for each tree. Each time the generator is called it returns the next element of the fringe of its tree. After the generator has examined the entire tree, it returns a special "exhausted" flag. The generator is most naturally written as a recursive function. The use of coroutines, i.e. stack groups, allows the two generators to recurse separately on two different control stacks without having to coordinate with each other.

The program is very simple. Constructing it in the usual bottom-up style, we first write a recursive function that takes a tree and **stack-group-returns** each element of its fringe. The **stack-group-return** is how the generator coroutine delivers its output. We could easily test this function by changing **stack-group-return** to **print** and trying it on some examples.

```
(defun fringe (tree)
  (cond ((atom tree) (stack-group-return tree))
        (t (fringe (car tree))
           (if (not (null (cdr tree)))
               (fringe (cdr tree))))))
```

Now we package this function inside another, which takes care of returning the special "exhausted" flag.

```
(defun fringe1 (tree exhausted)
  (fringe tree)
  exhausted)
```

The **samefringe** function takes the two trees as arguments and returns **t** or **nil**. It creates two stack groups to act as the two generator coroutines, presets them to run the **fringe1** function, then goes into a loop comparing the two fringes. The value is **nil** if a difference is discovered, or **t** if they are still the same when the end is reached.

```
(defun samefringe (tree1 tree2)
  (let ((sg1 (make-stack-group "samefringe1"))
        (sg2 (make-stack-group "samefringe2"))
        (exhausted (ncons nil)))
    (stack-group-preset sg1 #'fringe1 tree1 exhausted)
    (stack-group-preset sg2 #'fringe1 tree2 exhausted)
    (do ((v1) (v2)) (nil)
      (setq v1 (funcall sg1 nil)
            v2 (funcall sg2 nil))
      (cond ((neq v1 v2) (return nil))
            ((eq v1 exhausted) (return t))))))
```

Now we test it on a couple of examples:

```
(samefringe '(a b c) '(a (b c))) => t
(samefringe '(a b c) '(a b c d)) => nil
```

As stack groups are large, and slow to create, it is desirable to avoid the overhead of creating one each time two fringes are compared. It can easily be eliminated with a modest amount of explicit storage allocation, using the resource facility (see page 124). While we're at it, we can avoid making the exhausted flag fresh each time; its only important property is that it not be an atom.

```
(defresource samefringe-coroutine ()
   :constructor (make-stack-group "for-samefringe"))

(defvar exhausted-flag (ncons nil))

(defun samefringe (tree1 tree2)
  (using-resource (sg1 samefringe-coroutine)
    (using-resource (sg2 samefringe-coroutine)
      (stack-group-preset sg1 #'fringe1 tree1 exhausted-flag)
      (stack-group-preset sg2 #'fringe1 tree2 exhausted-flag)
      (do ((v1) (v2)) (nil)
        (setq v1 (funcall sg1 nil)
              v2 (funcall sg2 nil))
        (cond ((neq v1 v2) (return nil))
              ((eq v1 exhausted-flag) (return t)))))))
```

Now we can compare the fringes of two trees with no allocation of memory whatsoever.

14. Locatives

A *locative* is a type of Lisp object used as a *pointer* to a *cell*. Locatives are inherently a more low level construct than most Lisp objects; they require some knowledge of the nature of the Lisp implementation.

14.1 Cells and Locatives

A *cell* is a machine word that can hold a (pointer to a) Lisp object. For example, a symbol has five cells: the print name cell, the value cell, the function cell, the property list cell, and the package cell. The value cell holds (a pointer to) the binding of the symbol, and so on. Also, an array leader of length *n* has *n* cells, and an art-q array of *n* elements has *n* cells. (Numeric arrays do not have cells in this sense.) A locative is an object that points to a cell; it lets you refer to a cell so that you can examine or alter its contents.

contents *locative*

Returns the contents of the cell which the locative points to. This is actually the same as cdr, for reasons explained below, but it is clearer to use contents when the argument is normally a locative.

To modify the contents of the cell, use setf on contents:

```
        (setf (contents loc) newvalue)
```

The macro locf (see page 38) can be used to convert a form that accesses a cell to one that creates a locative pointer to that cell: for example,

```
    (locf (fsymeval x))
```

evaluates to a locative that points to the function cell of the value of x; that is to say, it points to the place where (fsymeval x) is stored.

locf is very convenient because it saves the writer and reader of a program from having to remember the names of many functions that would create locatives to cells found in different fashions.

One thing you should know is that it is not possible to make a locative to an element of a numeric array. For example,

```
    (setq foo (make-array 10 :type art-1b))
    (locf (aref foo 0))
```

signals an error. Locatives may only point at entire words of memory, which contain standard Lisp data.

Because of cdr-coding (see section 5.4, page 100), a cons does not always contain an explicit cell which points to its cdr. Therefore, it is impossible to obtain a locative which points to such a cell. However, this is such a useful thing to do that *the cons itself* is usually treated as if it were a locative pointing to a cell which holds the cons's cdr. (locf (cdr x)) returns the value of x, and (contents x) returns the cdr when *x* is a cons, so (contents (locf (cdr x))) is the same as (cdr x), as it should be. Most functions that are normally given locatives also accept a cons and treat it as if it were a magic locative to the (nonexistent) cell containing the cdr of the cons.

A cons always does contain a cell which points to the car, and (locf (car x)) returns a locative whose pointer field is the same as that of x's value.

14.2 Functions That Operate on Locatives

location-boundp *locative*

Returns t if the cell to which *locative* points contains anything except a void marker.

The void marker is a special data type, dtp-null, which is stored in cells to say that their value is missing. For example, an unbound variable actually has a void marker in its value cell, and (location-boundp (locf x)) is equivalent to (variable-boundp x).

location-makunbound *locative* &optional *pointer*

Stores a void marker into the cell to which *locative* points. This consists of data type field dtp-null and a pointer copied from *pointer*.

The pointer field of the void marker is used to tell the error handler what variable was unbound. In the case of a symbol's value cell or function cell, it should point to the symbol header. In the case of a flavor method, it should point to the beginning of the block of data that holds the definition, which is a word containing the method's function spec.

If the second arg is not specified, then where the void marker points is not defined.

Other functions with which locatives are expected or useful include get (the locative points to the cell in which the plist is stored), store-conditional (the locative points to the cell to be tested and modified), and %bind (the locative points to the cell to be bound).

14.3 Mixing Locatives with Lists

Either of the functions car and cdr (see page 87) may be given a locative, and will return the contents of the cell at which the locative points. They are both equivalent to contents when the argument is a locative.

Similarly, either of the functions rplaca and rplacd may be used to store an object into the cell at which a locative points.
For example,

```
    (rplaca locative y)
```
or
```
    (rplaca locative y)
```
is the same as
```
    (setf (contents locative) y)
```

If you are just using locatives, you should use contents rather than car or cdr. But you can also mix locatives and conses. For example, the same variable may usefully sometimes have a locative as its value and sometimes a cons. Then it is useful that car and cdr work on locatives, and it also matters which one you use. Pick the one that is right for the case of a cons.

For example, the following function conses up a list in the forward order by adding onto the end. It needs to know where to put the pointer to the next cell. Usually it goes in the previous cell's cdr, but the first cell gets put in the cell where the list is supposed to end up. A locative is used as the pointer to this cell. The first time through the loop, the **rplacd** is equivalent to **(setq res ...)**; on later times through the loop the **rplacd** tacks an additional cons onto the end of the list.

```
(defun simplified-version-of-mapcar (fcn lst)
  (do ((lst lst (cdr lst))
       (res nil)
       (loc (locf res)))
      ((null lst) res)
    (setf (cdr loc)
          (setq loc (ncons (funcall fcn (car lst)))))))
```

cdr is used here rather than **contents** because the normal case is that the argument is a list.

15. Subprimitives

Subprimitives are functions which are not intended to be used by the average program, only by system programs. They allow one to manipulate the environment at a level lower than normal Lisp. They are described in this chapter. Subprimitives usually have names starting with a % character. The primitives described in other sections of the manual typically use subprimitives to accomplish their work. To some extent the subprimitives take the place of what in other systems would be individual machine instructions. Subprimitives are normally hand-coded in microcode.

There is plenty of stuff in this chapter that is not fully explained; there are terms that are undefined, there are forward references, and so on. Furthermore, most of what is in here is considered subject to change without notice. In fact, this chapter does not exactly belong in this manual, but in some other more low-level manual. Since the latter manual does not exist, it is here for the interim.

Subprimitives by their very nature cannot do full checking. Improper use of subprimitives can destroy the environment. Subprimitives come in varying degrees of dangerousness. Generally, those without a % sign in their name are not directly dangerous, whereas those whose names begin with % can ruin the Lisp world just as readily as they can do something useful. The subprimitives are documented here since they need to be documented somewhere, but this manual does not document all the things you need to know in order to use them. Still other subprimitives are not documented here because they are very specialized. Most of these are never used explicitly by a programmer; the compiler inserts them into the program to perform operations which are expressed differently in the source code.

The most common problem you can cause using subprimitives, though by no means the only one, is to create illegal pointers: pointers that are, for one reason or another, according to storage conventions, not allowed to exist. The storage conventions are not documented; as we said, you have to be an expert to use a lot of the functions in this chapter correctly. If you create such an illegal pointer, it probably will not be detected immediately, but later on parts of the system may see it, notice that it is illegal, and (probably) halt the Lisp Machine.

In a certain sense **car**, **cdr**, **rplaca**, and **rplacd** are subprimitives. If these are given a locative instead of a list, they access or modify the cell addressed by the locative without regard to what object the cell is inside. Subprimitives can be used to create locatives to strange places.

15.1 Data Types

data-type *arg*

data-type returns a symbol that is the name for the internal data-type of *arg*. The **type-of** function (page 20) is a higher-level primitive that is more useful in most cases; normal programs should always use **type-of** (or, when appropriate, **typep**) rather than **data-type**.

Note that some types as seen by the user are not distinguished from each other at this level, and some user types may be represented by more than one internal type. For example, **dtp-extended-number** is the symbol that **data-type** would return for either a single-float or a bignum, even though those two types are quite different.

Some of these type codes occur in memory words but cannot be the type of an actual Lisp object. These include header types such as dtp-symbol-header, which identify the first word of a structure, and forwarding or "invisible" pointer types such as dtp-one-q-forward.

dtp-symbol	The object is a symbol.
dtp-fix	The object is a fixnum; the numeric value is contained in the address field of the pointer.
dtp-small-flonum	The object is a short float; the numeric value is contained in the address field of the pointer.
dtp-extended-number	The object is a single-float, ratio, bignum or complexnum. This value will also be used for future numeric types.
dtp-character	The object is a character object; the value is contained in the address field of the pointer.
dtp-list	The object is a cons.
dtp-locative	The object is a locative pointer.
dtp-array-pointer	The object is an array.
dtp-fef-pointer	The object is a compiled function.
dtp-u-entry	The object is a microcode entry.
dtp-closure	The object is a closure; see chapter 12, page 250.
dtp-stack-closure	The object is a closure which lives inside a stack, and which must be copied if it is stored anywhere but farther down in the same stack. Lexical scoping is implemented using these.
dtp-instance	The object is an instance of a flavor; see chapter 21, page 401.
dtp-entity	The object is an entity; see section 12.4, page 255.
dtp-select-method	The object is a select-method; see page 232.
dtp-stack-group	The object is a stack-group; see chapter 13, page 256.

The remaining types are internal only.

dtp-header	An internal type used to mark the first word of several kinds of multi-word structure, including single-floats, ratios, bignums and FEFs.
dtp-array-header	An internal type used to mark the first word of an array.
dtp-symbol-header	An internal type used to mark the first word of a symbol. The pointer field points to the symbol's print-name, which is a string.
dtp-instance-header	An internal type used to mark the first word of an instance. The pointer field points to the structure that describes the instance's flavor.
dtp-null	Nothing to do with nil. This type code identifies a void marker. An attempt to refer to the contents of a cell that contains a dtp-null signals an error. This is how "unbound variable" and

"undefined function" errors are detected.

dtp-trap	The zero data-type, which is not used. This hopes to detect microcode bugs.
dtp-free	This type is used to fill free storage, to catch wild references.
dtp-external-value-cell-pointer	An "invisible pointer" used for external value cells, which are part of the closure mechanism (see chapter 12, page 250), and used by compiled code to address value and function cells.
dtp-self-ref-pointer	An "invisible pointer" used to refer to an instance variable of self. This data type appears in FEFs of flavor methods.
dtp-header-forward	An "invisible pointer" used to indicate that the structure containing it has been moved elsewhere. The "header word" of the structure is replaced by one of these invisible pointers. See the function structure-forward (page 273).
dtp-body-forward	An "invisible pointer" used to indicate that the structure containing it has been moved elsewhere. This points to the word containing the header-forward, which points to the new copy of the structure.
dtp-one-q-forward	An "invisible pointer" used to indicate that the single cell containing it has been moved elsewhere.
dtp-gc-forward	This is used by the copying garbage collector to flag the obsolete copy of an object; it points to the new copy.

q-data-types *Constant*

The value of q-data-types is a list of all of the symbolic names for data types described above under data-type. These are the symbols whose print names begin with 'dtp-'. The values of these symbols are the internal numeric data-type codes for the various types.

q-data-types *type-code*

Given the internal numeric data-type code, returns the corresponding symbolic name. This "function" is actually an array.

15.2 Forwarding

An *invisible pointer* or *forwarding pointer* is a kind of pointer that does not represent a Lisp object, but just resides in memory. There are several kinds of invisible pointer, and there are various rules about where they may or may not appear. The basic property of an invisible pointer is that if the Lisp Machine reads a word of memory and finds an invisible pointer there, instead of seeing the invisible pointer as the result of the read, it does a second read, at the location addressed by the invisible pointer, and returns that as the result instead. Writing behaves in a similar fashion. When the Lisp Machine writes a word of memory it first checks to see if that word contains an invisible pointer; if so it goes to the location pointed to by the invisible pointer and tries to write there instead. Many subprimitives that read and write memory do not do this checking.

The simplest kind of invisible pointer has the data type code dtp-one-q-forward. It is used to forward a single word of memory to someplace else. The invisible pointers with data types dtp-header-forward and dtp-body-forward are used for moving whole Lisp objects (such as cons cells or arrays) somewhere else. The dtp-external-value-cell-pointer is very similar to the dtp-one-q-forward; the difference is that it is not "invisible" to the operation of binding. If the (internal) value cell of a symbol contains a dtp-external-value-cell-pointer that points to some other word (the external value cell), then symeval or set operations on the symbol consider the pointer to be invisible and use the external value cell, but binding the symbol saves away the dtp-external-value-cell-pointer itself, and stores the new value into the internal value cell of the symbol. This is how closures are implemented.

dtp-gc-forward is not an invisible pointer at all; it only appears in "old spaced" and can never be seen by any program other than the garbage collector. When an object is found not to be garbage, and the garbage collector moves it from "old space" to "new space", a dtp-gc-forward is left behind to point to the new copy of the object. This ensures that other references to the same object get the same new copy.

structure-forward *old-object new-object*

This causes references to *old-object* actually to reference *new-object*, by storing invisible pointers in *old-object*. It returns *old-object*.

An example of the use of structure-forward is adjust-array. If the array is being made bigger and cannot be expanded in place, a new array is allocated, the contents are copied, and the old array is structure-forwarded to the new one. This forwarding ensures that pointers to the old array, or to cells within it, continue to work. When the garbage collector goes to copy the old array, it notices the forwarding and uses the new array as the copy; thus the overhead of forwarding disappears eventually if garbage collection is in use.

follow-structure-forwarding *object*

Normally returns *object*, but if *object* has been structure-forward'ed, returns the object at the end of the chain of forwardings. If *object* is not exactly an object, but a locative to a cell in the middle of an object, a locative to the corresponding cell in the latest copy of the object is returned.

forward-value-cell *from-symbol to-symbol*

This alters *from-symbol* so that it always has the same value as *to-symbol*, by sharing its value cell. A dtp-one-q-forward invisible pointer is stored into *from-symbol*'s value cell. Do not do this while *from-symbol*'s current dynamic binding is not global, as the microcode does not bother to check for that case and something bad will happen when *from-symbol*'s binding is unbound. The microcode check is omitted to speed up binding and unbinding.

This is how synonymous variables (such as *terminal-io* and terminal-io) are created.

To forward one arbitrary cell to another (rather than specifically one value cell to another), given two locatives, do

```
(%p-store-tag-and-pointer locative1 dtp-one-q-forward locative2)
```

follow-cell-forwarding *loc evcp-p*

loc is a locative to a cell. Normally *loc* is returned, but if the cell has been forwarded, this follows the chain of forwardings and returns a locative to the final cell. If the cell is part of a structure which has been forwarded, the chain of structure forwardings is followed, too. If *evcp-p* is t, external value cell pointers are followed; if it is nil they are not.

15.3 Pointer Manipulation

It should again be emphasized that improper use of these functions can damage or destroy the Lisp environment. It is possible to create pointers with illegal data-type, pointers to non-existent objects, and pointers to untyped storage, which will completely confuse the garbage collector.

%data-type *x*

Returns the data-type field of *x*, as a fixnum.

%pointer *x*

Returns the pointer field of *x*, as a fixnum. For most types, this is dangerous since the garbage collector can copy the object and change its address.

%make-pointer *data-type pointer*

Makes up a pointer, with *data-type* in the data-type field and *pointer* in the pointer field, and returns it. *data-type* should be an internal numeric data-type code; these are the values of the symbols that start with dtp-. *pointer* may be any object; its pointer field is used. This is most commonly used for changing the type of a pointer. Do not use this to make pointers which are not allowed to be in the machine, such as dtp-null, invisible pointers, etc.

%make-pointer-offset *data-type pointer offset*

Returns a pointer with *data-type* in the data-type field, and *pointer* plus *offset* in the pointer field. The *data-type* and *pointer* arguments are like those of %make-pointer; *offset* may be any object but is usually a fixnum. The types of the arguments are not checked; their pointer fields are simply added together. This is useful for constructing locative pointers into the middle of an object. However, note that it is illegal to have a pointer to untyped data, such as the inside of a FEF or a numeric array.

%pointer-difference *pointer-1 pointer-2*

Returns a fixnum which is *pointer-1* minus *pointer-2*. No type checks are made. For the result to be meaningful, the two pointers must point into the same object, so that their difference cannot change as a result of garbage collection.

%pointerp *object*

t if *object* points to storage. For example, (%pointerp "foo") is t, but (%pointerp 5) is nil.

%pointer-type-p *data-type*

t if the specified data type is one which points to storage. For example, (%pointer-type-p dtp-fix) returns nil.

15.4 Special Memory Referencing

%p-pointerp *location*

t if the contents of the word at *location* points to storage. This is similar to (%pointerp (contects *location*)), but the latter may get an error if *location* contains a forwarding pointer, a header type, or a void marker. In such cases, %p-pointerp correctly tells you whether the header or forward points to storage.

%p-pointerp-offset *location offset*

Similar to %p-pointerp but operates on the word *offset* words beyond *location*.

%p-contents-offset *base-pointer offset*

Returns the contents of the word *offset* words beyond *base-pointer*. This first checks the cell pointed to by *base-pointer* for a forwarding pointer. Having followed forwarding pointers to the real structure pointed to, it adds *offset* to the resulting forwarded *base-pointer* and returns the contents of that location.

There is no %p-contents, since car performs that operation.

%p-contents-safe-p *location*

t if the contents of word *location* are a valid Lisp object, at least as far as data type is concerned. It is nil if the word contains a header type, a forwarding pointer, or a void marker. If the value of this function is t, you will not get an error from (contents *location*).

%p-contents-safe-p-offset *location offset*

Similar to %p-contents-safe-p but operates on the word *offset* words beyond *location*.

%p-contents-as-locative *pointer*

Given a pointer to a memory location containing a pointer that isn't allowed to be "in the machine" (typically an invisible pointer) this function returns the contents of the location as a dtp-locative. It changes the disallowed data type to dtp-locative so that you can safely look at it and see what it points to.

%p-contents-as-locative-offset *base-pointer offset*

Extracts the contents of a word like %p-contents-offset, but changes it into a locative like %p-contents-as-locative. This can be used, for example, to analyze the dtp-external-value-cell-pointer pointers in a FEF, which are used by the compiled code to reference value cells and function cells of symbols.

%p-safe-contents-offset *location offset*

Returns the contents of the word *offset* words beyond *location* as accurately as possible without getting an error.

If the contents are a valid Lisp object, it is returned exactly.

If the contents are not a valid Lisp object but do point to storage, the value returned is a locative which points to the same place in storage.

If the contents are not a valid Lisp object and do not point to storage, the value returned is a fixnum with the same pointer field.

Forwarding pointers are checked as in %p-contents-offset.

%p-store-contents *pointer value*

Stores *value* into the data-type and pointer fields of the location addressed by *pointer*, and returns *value*. The cdr-code field of the location remains unchanged.

%p-store-contents-offset *value base-pointer offset*

Stores *value* in the location *offset* beyond words beyond *base-pointer*, then returns *value*. The cdr-code field remains unchanged. Forwarding pointers in the location at *base-pointer* are handled as they are in %p-contents-offset.

%p-store-tag-and-pointer *pointer miscfields pointerfield*

Stores *miscfields* and *pointerfield* into the location addressed by *pointer*. 25 bits are taken from *pointerfield* to fill the pointer field of the location, and the low 7 bits of *miscfields* are used to fill both the data-type and cdr-code fields of the location. The low 5 bits of *miscfields* become the data-type, and the top two bits become the cdr-code. This is a good way to store a forwarding pointer from one structure to another (for example).

%p-store-tag-and-pointer should be used only for storing into 'boxed' words, for the same reason as %blt-typed: the microcode could halt if the data stored is not valid boxed data. See page 280.

%p-ldb *byte-spec pointer*

Extracts a byte according to *byte-spec* from the contents of the location addressed by *pointer*, in effect regarding the contents as a 32-bit number and using ldb. The result is always a fixnum. For example, (%p-ldb %%q-cdr-code *loc*) returns the cdr-code of the location addressed by *loc*.

%p-ldb-offset *byte-spec base-pointer offset*

Extracts a byte according to *byte-spec* from the contents of the location *offset* words beyond *base-pointer*, after handling forwarding pointers like %p-contents-offset.

This is the way to reference byte fields within a structure without violating system storage conventions.

%p-mask-field *byte-spec pointer*
Like %p-ldb, except that the selected byte is returned in its original position within the word instead of right-aligned.

%p-mask-field-offset *byte-spec base-pointer offset*
Like %p-ldb-offset, except that the selected byte is returned in its original position within the word instead of right-aligned.

Note: %p-dbp, %p-dpb-offset, %p-deposit-field and %p-deposit-field-offset should never be used to modify the pointer field of a boxed word if the data type is one which actually points to storage, unless you are sure that the new pointer is such as to cause no trouble (such as, if it points to a static area). Likewise, it should never be used to change a data type which does not point to storage into one which does. Either action could confuse the garbage collector.

%p-dpb *value byte-spec pointer*
Stores *value*, a fixnum, into the byte selected by *byte-spec* in the word addressed by *pointer*. nil is returned. You can use this to alter data types, cdr-codes, etc., but see the note above for restrictions.

%p-dpb-offset *value byte-spec base-pointer offset*
Stores *value* into the specified byte of the location *offset* words beyond that addressed by *base-pointer*, after first handling forwarding pointers in the location addressed by *base-pointer* as in %p-contents-offset. nil is returned.

This is the way to alter unboxed data within a structure without violating system storage conventions. You can use this to alter boxed words too, but see the note above for restrictions.

%p-deposit-field *value byte-spec pointer*
Like %p-dpb, except that the selected byte is stored from the corresponding bits of *value* rather than the right-aligned bits. See the note above %p-dpb for restrictions.

%p-deposit-field-offset *value byte-spec base-pointer offset*
Like %p-dpb-offset, except that the selected byte is stored from the corresponding bits of *value* rather than the right-aligned bits. See the note above %p-dpb for restrictions.

%p-pointer *pointer*
Extracts the pointer field of the contents of the location addressed by *pointer* and returns it as a fixnum.

%p-data-type *pointer*
Extracts the data-type field of the contents of the location addressed by *pointer* and returns it as a fixnum.

%p-cdr-code *pointer*
Extracts the cdr-code field of the contents of the location addressed by *pointer* and returns it as a fixnum.

%p-store-pointer *pointer value*
Stores *value* in the pointer field of the location addressed by *pointer*, and returns *value*.

%p-store-data-type *pointer value*
Stores *value* in the data-type field of the location addressed by *pointer*, and returns *value*.

%p-store-cdr-code *pointer value*
Stores *value* in the cdr-code field of the location addressed by *pointer*, and returns *value*.

%stack-frame-pointer
Returns a locative pointer to its caller's stack frame. This function is not defined in the interpreted Lisp environment; it only works in compiled code. Since it turns into a "misc" instruction, the "caller's stack frame" really means "the frame for the FEF that executed the %stack-frame-pointer instruction".

15.5 Storage Layout Definitions

The following special variables have values which define the most important attributes of the way Lisp data structures are laid out in storage. In addition to the variables documented here, there are many others that are more specialized. They are not documented in this manual since they are in the system package rather than the global package. The variables whose names start with %% are byte specifiers, intended to be used with subprimitives such as %p-ldb. If you change the value of any of these variables, you will probably bring the machine to a crashing halt.

%%q-cdr-code *Constant*
The field of a memory word that contains the cdr-code. See section 5.4, page 100.

%%q-data-type *Constant*
The field of a memory word that contains the data-type code. See page 270.

%%q-pointer *Constant*
The field of a memory word that contains the pointer address, or immediate data.

%%q-pointer-within-page *Constant*
The field of a memory word that contains the part of the address that lies within a single page.

%%q-typed-pointer *Constant*
The concatenation of the %%q-data-type and %%q-pointer fields.

%%q-all-but-typed-pointer *Constant*
This is now synonymous with %%q-cdr-code, and therefore obsolete.

%%q-all-but-pointer *Constant*
The concatenation of all fields of a memory word except for %%q-pointer.

%%q-all-but-cdr-code *Constant*
The concatenation of all fields of a memory word except for %%q-cdr-code.

%%q-high-half *Constant*
%%q-low-half *Constant*
The two halves of a memory word. These fields are only used in storing compiled code.

cdr-normal *Constant*
cdr-next *Constant*
cdr-nil *Constant*
cdr-error *Constant*
The values of these four variables are the numeric values that go in the cdr-code field of a memory word. See section 5.4, page 100 for the details of cdr-coding.

15.6 Analyzing Structures

%find-structure-header *pointer*
This subprimitive finds the structure into which *pointer* points, by searching backward for a header. It is a basic low-level function used by such things as the garbage collector. *pointer* is normally a locative, but its data-type is ignored. Note that it is illegal to point into an "unboxed" portion of a structure, for instance the middle of a numeric array.

In structure space, the "containing structure" of a pointer is well-defined by system storage conventions. In list space, it is considered to be the contiguous, cdr-coded segment of list surrounding the location pointed to. If a cons of the list has been copied out by rplacd, the contiguous list includes that pair and ends at that point.

%find-structure-leader *pointer*
This is identical to %find-structure-header, except that if the structure is an array with a leader, this returns a locative pointer to the leader-header, rather than returning the array-pointer itself. Thus the result of %find-structure-leader is always the lowest address in the structure. This is the one used internally by the garbage collector.

%structure-boxed-size *object*
Returns the number of "boxed Q's" in *object*. This is the number of words at the front of the structure which contain normal Lisp objects. Some structures, for example FEFs and numeric arrays, contain additional "unboxed Q's" following their boxed Q's. Note that the boxed size of a PDL (either regular or special) does not include Q's above the current top of the PDL. Those locations are boxed, but their contents are considered garbage and are not protected by the garbage collector.

%structure-total-size *object*

Returns the total number of words occupied by the representation of *object*, including boxed Q's, unboxed Q's, and garbage Q's off the ends of PDLs.

15.7 Creating Objects

%allocate-and-initialize *data-type header-type header second-word area size*

This is the subprimitive for creating most structured-type objects. *area* is the area in which it is to be created, as a fixnum or a symbol. *size* is the number of words to be allocated. The value returned points to the first word allocated and has data-type *data-type*. Uninterruptibly, the words allocated are initialized so that storage conventions are preserved at all times. The first word, the header, is initialized to have *header-type* in its data-type field and *header* in its pointer field. The second word is initialized to *second-word*. The remaining words are initialized to nil. The cdr-codes of all words except the last are set to cdr-next; the cdr-code of the last word is set to cdr-nil. It is probably a bad idea to rely on this.

%allocate-and-initialize-array *header data-length leader-length area size*

This is the subprimitive for creating arrays, called only by make-array. It is different from %allocate-and-initialize because arrays have a more complicated header structure.

The basic functions for creating list-type objects are cons and make-list; no special subprimitive is needed. Closures, entities, and select-methods are based on lists, but there is no primitive for creating them. To create one, create a list and then use %make-pointer to change the data type from dtp-list to the desired type.

15.8 Copying Data

%blt and %blt-typed are subprimitives for copying blocks of data, word aligned, from one place in memory to another with little or no type checking.

%blt *from to count increment*
%blt-typed *from to count increment*

Copies *count* words, separated by *increment*. The word at address *from* is moved to address *to*, the word at address *from* + *increment* is moved to address *to* + *increment*, and so on until *count* words have been moved.

Only the pointer fields of *from* and *to* are significant; they may be locatives or even fixnums. If one of them must point to the unboxed data in the middle of a structure, you must make it a fixnum, and you must do so with interrupts disabled, or else garbage collection could move the structure after you have already created the fixnum.

%blt-typed assumes that each copied word contains a data type field and checks that field, interfacing suitably with the garbage collector if necessary. %blt does not check the data type fields of the copied words.

%blt may be used on any data except boxed data containing pointers to storage, while **%blt-typed** may be used on any boxed data. Both **%blt** and **%blt-typed** can be used validly on data which is formatted with data types (boxed) but whose contents never point to storage. This includes words whose contents are always fixnums or short floats, and also words which contain array headers, array leader headers, or FEF headers. Whether or not the machine is told to examine the data types of such data makes no difference since, on examining them, it would decide that nothing needed to be done.

For unboxed data (data which is formatted not containing valid data type fields), such as the inside of a numeric array or the instruction words of a FEF, only **%blt** may be used. If **%blt-typed** were used, it would examine the data type fields of the data words, and probably halt due to an invalid data type code.

For boxed data which may contain pointers, only **%blt-typed** may be used. If **%blt** were used, it would appear to work, but problems could appear mysteriously later because nothing would notice the presence of the pointer there. For example, the pointer might point to a bignum in the number consing area, and moving it in this way would fail to copy it into a nontemporary area. Then the pointer would become invalidated the next time the number consing area was emptied out. There could also be problems with lexical closures and with garbage collection.

15.9 Returning Storage

return-storage *object*

This peculiar function attempts to return *object* to free storage. If it is a displaced array, this returns the displaced array itself, not the data that the array points to. Currently **return-storage** does nothing if the object is not at the end of its region, i.e. if it was not either the most recently allocated non-list object in its area, or the most recently allocated list in its area.

If you still have any references to *object* anywhere in the Lisp world after this function returns, the garbage collector can get a fatal error if it sees them. Since the form that calls this function must get the object from somewhere, it may not be clear how to legally call **return-storage**. One of the only ways to do it is as follows:

```
(defun func ()
  (let ((object (make-array 100)))
    ...
    (return-storage (prog1 object (setq object nil)))))
```

so that the variable **object** does not refer to the object when **return-storage** is called. Alternatively, you can free the object and get rid of all pointers to it while interrupts are turned off with **without-interrupts**.

You should only call this function if you know what you are doing; otherwise the garbage collector can get fatal errors. Be careful.

15.10 Locking Subprimitive

%store-conditional *pointer old new*

This is the basic locking primitive. *pointer* is a locative to a cell which is uninterruptibly read and written. If the contents of the cell is eq to *old*, then it is replaced by *new* and t is returned. Otherwise, nil is returned and the contents of the cell are not changed.

See also store-conditional, a higher-level function which provides type checking (page 688).

15.11 CADR I/O Device Subprimitives

The CADR processor has a 32-bit memory bus called the Xbus. In addition to main memory and TV screen memory, most I/O device registers are on this bus. There is also a Unibus compatible with the PDP-11. A map of Xbus and Unibus addresses can be found in SYS: DOC; UNADDR TEXT.

%unibus-read *address*

Returns as a fixnum the contents of the register at the specified Unibus address. You must specify a full 18-bit address. This is guaranteed to read the location only once. Since the Lisp Machine Unibus does not support byte operations, this always references a 16-bit word, and so *address* should normally be an even number.

%unibus-write *address data*

Writes the 16-bit number *data* at the specified Unibus address, exactly once.

%xbus-read *io-offset*

Returns the contents of the register at the specified Xbus address. *io-offset* is an offset into the I/O portion of Xbus physical address space. This is guaranteed to read the location exactly once. The returned value can be either a fixnum or a bignum.

%xbus-write *io-offset data*

Writes *data*, which can be a fixnum or a bignum, into the register at the specified Xbus address. *io-offset* is an offset into the I/O portion of Xbus physical address space. This is guaranteed to write the location exactly once.

sys:%xbus-write-sync *w-loc w-data delay sync-loc sync-mask sync-value*

Does (%xbus-write *w-loc w-data*), but first synchronizes to within about one microsecond of a certain condition. The synchronization is achieved by looping until

(= (logand (%xbus-read *sync-loc*) *sync-mask*) *sync-value*)

is false, then looping until it is true, then looping *delay* times. Thus the write happens a specified delay after the leading edge of the synchronization condition. The number of microseconds of delay is roughly one third of *delay*.

This primitive is used to alter the color TV screen's color map during vertical retrace.

15.12 Lambda I/O-Device Subprimitives

sys:%nubus-read *slot byte-address*

Returns the contents of a word read from the Nu bus. Addresses on the Nu bus are divided into an 8-bit slot number which identifies which physical board is being referenced and a 24-bit address within slot. The address is measured in bytes and therefore should be a multiple of 4. Which addresses are valid depends on the type of board plugged into the specified slot. If, for example, the board is a 512k main memory board, then the valid address range from 0 to 4 * (512k - 1). (Of course, main memory boards are normally accessed through the virtual memory mechanism.)

sys:%nubus-write *slot byte-address word*

Writes *word* into a word of the Nu bus, whose address is specified by *slot* and *byte-address* as described above.

sys:%nubus-physical-address *apparent-physical-page*

The valid portions of the Nu bus address space are not contiguous. Each board is allocated 16m bytes of address space, but no memory board actually provides 16m bytes of memory.

The Lisp Machine virtual memory system maps virtual addresses into a contiguous physical address space. On the Lambda, this contiguous address space is mapped a second time into the discontiguous Nu bus address space. Unlike the mapping of virtual addresses to physical ones, the second mapping is determined from the hardware configuration when the machine is booted and does not change during operation.

This function performs exactly that mapping. The argument is a physical page number (a physical address divided by sys:page-size). The argument is a "Nu bus page number"; multiplied by sys:page-size and then by four, it yields the Nu bus byte address of the beginning of that physical page.

See also sys:%physical-address, page 286.

15.13 Function-Calling Subprimitives

These subprimitives can be used (carefully!) to call a function with the number of arguments variable at run time. They only work in compiled code and are not defined in the interpreted Lisp environment. The preferred higher-level primitive is apply (page 47).

%open-call-block *function n-adi-pairs destination*

Starts a call to *function*. *n-adi-pairs* is the number of pairs of additional information words already %push'ed; normally this should be 0. *destination* is where to put the result; the useful values are 0 for the value to be ignored, 1 for the value to go onto the stack, 3 for the value to be the last argument to the previous open call block, and 2 for the value to be returned from this frame.

%push *value*
Pushes *value* onto the stack. Use this to push the arguments.

%activate-open-call-block
Causes the call to happen.

%pop
Pops the top value off of the stack and returns it as its value. Use this to recover the result from a call made by %open-call-block with a destination of 1.

%assure-pdl-room *n-words*
Call this before doing a sequence of %push's or %open-call-block's that will add *n-words* to the current frame. This subprimitive checks that the frame will not exceed the maximum legal frame size, which is 255 words including all overhead. This limit is dictated by the way stack frames are linked together. If the frame is going to exceed the legal limit, %assure-pdl-room signals an error.

15.14 Special-Binding Subprimitive

%bind *locative value*
bind *locative value*
Binds the cell pointed to by *locative* to *x*, in the caller's environment. This function is not defined in the interpreted Lisp environment; it only works from compiled code. Since it turns into an instruction, the "caller's environment" really means "the binding block for the compiled function that executed the %bind instruction". The preferred higher-level primitives that turn into this are let (page 31), let-if (page 32), and progv (page 32).

The binding is in effect for the scope of the innermost binding construct, such as prog or let—even one that binds no variables itself.

%bind is the preferred name; bind is an older name which will eventually be eliminated.

15.15 The Paging System

[Someday this may discuss how it works.]

sys:%disk-switches *Variable*
This variable contains bits that control various disk usage features.

Bit 0 (the least significant bit) enables read-compares after disk read operations. This causes a considerable slowdown, so it is rarely used.

Bit 1 enables read-compares after disk write operations.

Bit 2 enables the multiple page swap-out feature. When this is enabled, as it is by default, each time a page is swapped out, up to 16. contiguous pages are also written out to the disk if they have been modified. This greatly improves swapping performance.

Bit 3 controls the multiple page swap-in feature, which is also on by default. This feature causes pages to be swapped in in groups; each time a page is needed, several contiguous pages are swapped in in the same disk operation. The number of pages swapped in can be specified for each area using si:set-swap-recommendations-of-area.

si:set-swap-recommendations-of-area *area-number recommendation*
Specifies that pages of area *area-number* should be swapped in in groups of *recommendation* at a time. This recommendation is used only if the multiple page swap-in feature is enabled.

Generally, the more memory a machine has, the higher the swap recommendations should be to get optimum performance. The recommendations are set automatically according to the memory size when the machine is booted.

si:set-all-swap-recommendations *recommendation*
Specifies the swap-in recommendation of all areas at once.

si:wire-page *address* &optional (*wire-p* t)
If *wire-p* is t, the page containing *address* is *wired-down*; that is, it cannot be paged-out. If *wire-p* is nil, the page ceases to be wired-down.

si:unwire-page *address*
(si:unwire-page *address*) is the same as (si:wire-page *address* nil).

sys:page-in-structure *object*
Makes sure that the storage that represents *object* is in main memory. Any pages that have been swapped out to disk are read in, using as few disk operations as possible. Consecutive disk pages are transferred together, taking advantage of the full speed of the disk. If *object* is large, this is much faster than bringing the pages in one at a time on demand. The storage occupied by *object* is defined by the %find-structure-leader and %structure-total-size subprimitives.

sys:page-in-array *array* &optional *from to*
This is a version of sys:page-in-structure that can bring in a portion of an array. *from* and *to* are lists of subscripts; if they are shorter than the dimensionality of *array*, the remaining subscripts are assumed to be zero.

sys:page-in-pixel-array *array* &optional *from to*
Like sys:page-in-array except that the lists *from* and *to*, if present, are assumed to have their subscripts in the order horizontal, vertical, regardless of which of those two is actually the first axis of the array. See make-pixel-array, page 182.

sys:page-in-words *address n-words*
Any pages that have been swapped out to disk in the range of address space starting at *address* and continuing for *n-words* are read in with as few disk operations as possible.

sys:page-in-area *area-number*
sys:page-in-region *region-number*

All swapped-out pages of the specified region or area are brought into main memory.

sys:page-out-structure *object*
sys:page-out-array *array* &optional *from to*
sys:page-out-pixel-array *array* &optional *from to*
sys:page-out-words *address n-words*
sys:page-out-area *area-number*
sys:page-out-region *region-number*

These are similar to the above, except that they take pages out of main memory rather than bringing them in. Actually, they only mark the pages as having priority for replacement by others. Use these operations when you are done with a large object, to make the virtual memory system prefer reclaiming that object's memory over swapping something else out.

sys:%page-status *virtual-address*

If the page containing *virtual-address* is swapped out, or if it is part of one of the low-numbered fixed areas, this returns nil. Otherwise it returns the entire first word of the page hash table entry for the page.

The %%pht1- symbols in SYS: SYS; QCOM LISP are byte specifiers you can use with %logldb for decoding the value.

sys:%change-page-status *virtual-address swap-status access-status-and-meta-bits*

The page hash table entry for the page containing *virtual-address* is found and altered as specified. t is returned if it was found, nil if it was not (presumably the page is swapped out). *swap-status* and *access-status-and-meta-bits* can be nil if those fields are not to be changed. This doesn't make any error checks; you can really screw things up if you call it with the wrong arguments.

sys:%compute-page-hash *virtual-address*

Makes the hashing function for the page hash table available to the user.

sys:%physical-address *virtual-address*

Returns the physical address which *virtual-address* currently maps into. The value is unpredictable if the virtual page is not swapped in; therefore, this function should be used on wired pages, or you should do

```
(without-interrupts
  (%p-pointer virtual-address)      ;swap it in
  (sys:%physical-address virtual-address))
```

sys:%create-physical-page *physical-address*

This is used when adjusting the size of real memory available to the machine. It adds an entry for the page frame at *physical-address* to the page hash table, with virtual address -1, swap status flushable, and map status 120 (read only). This doesn't make error checks; you can really screw things up if you call it with the wrong arguments.

sys:%delete-physical-page *physical-address*

If there is a page in the page frame at *physical-address*, it is swapped out and its entry is deleted from the page hash table, making that page frame unavailable for swapping in of pages in the future. This doesn't make error checks; you can really screw things up if you call it with the wrong arguments.

sys:%disk-restore *high-16-bits low-16-bits*

Loads virtual memory from the partition named by the concatenation of the two 16-bit arguments, and starts executing it. The name 0 refers to the default load (the one the machine loads when it is started up). This is the primitive used by disk-restore (see page 806).

sys:%disk-save *physical-mem-size high-16-bits low-16-bits*

Copies virtual memory into the partition named by the concatenation of the two 16-bit arguments (0 means the default), then restarts the world, as if it had just been restored. The *physical-mem-size* argument should come from %sys-com-memory-size in system-communication-area. If *physical-mem-size* is negative, it is minus the memory size, and an incremental save is done. This is the primitive used by disk-save (see page 807).

si:set-memory-size *nwords*

Specifies the size of physical memory in words. The Lisp machine determines the actual amount of physical memory when it is booted, but with this function you can tell it to use less memory than is actually present. This may be useful for comparing performance based on the amount of memory.

15.16 Closure Subprimitives

These functions deal with things like what closures deal with: the distinction between internal and external value cells and control over how they work.

sys:%binding-instances *list-of-symbols*

This is the primitive that could be used by closure. First, if any of the symbols in *list-of-symbols* has no external value cell, a new external value cell is created for it, with the contents of the internal value cell. Then a list of locatives, twice as long as *list-of-symbols*, is created and returned. The elements are grouped in pairs: pointers to the internal and external value cells, respectively, of each of the symbols. closure could have been defined by:

```
(defun closure (variables function)
  (%make-pointer dtp-closure
    (cons function (sys:%binding-instances variables))))
```

sys:%using-binding-instances *instance-list*

This function is the primitive operation that invocation of closures could use. It takes a list such as sys:%binding-instances returns, and for each pair of elements in the list, it "adds" a binding to the current stack frame, in the same manner that the %bind function does. These bindings remain in effect until the frame returns or is unwound.

sys:%using-binding-instances checks for redundant bindings and ignores them. (A binding is redundant if the symbol is already bound to the desired external value cell.) This check avoids excessive growth of the special pdl in some cases and is also made by the microcode which invokes closures, entities, and instances.

Given a closure, closure-bindings extracts its list of binding instances, which you can then pass to sys:%using-binding-instances.

sys:%internal-value-cell *symbol*
Returns the contents of the internal value cell of *symbol*. dtp-one-q-forward pointers are considered invisible, as usual, but dtp-external-value-cell-pointers are *not*: this function can return a dtp-external-value-cell-pointer. Such pointers will be considered invisible as soon as they leave the "inside of the machine", meaning internal registers and the stack.

15.17 Distiguishing Processor Types

The MIT Lisp Machine system runs on two types of processors: the CADR and the Lambda. These are similar enough that there is no difference in compiled code for them, and no provision for compile-time conditionalization. However, obscure or internal I/O code sometimes needs to behave differently at run-time depending on the type of processor. This is possible through the use of these macros.

sys:processor-type-code *Variable*
This variable is 1 on a CADR processor or equivalent, 2 on a Lambda.

if-in-cadr *body...* *Macro*
Executes *body* only when executing on a CADR.

if-in-lambda *body* *Macro*
Executes *body* only when executing on a Lambda.

if-in-cadr-else-lambda *if-cadr-form else-body...* *Macro*
Executes *if-cadr-form* when executing on a CADR, executes *else-body* when executing on a Lambda.

```
(format t "~&Processor is a ~A.~%"
        (if-in-cadr-else-lambda "CADR" "Lambda"))
```

if-in-lambda-else-cadr *if-lambda-form else-body...* *Macro*
Executes *if-cadr-form* when executing on a Lambda executes *else-body* when executing on a CADR.

select-processor *clauses* *Macro*
Each clause consists of :cadr or :lambda followed by forms to execute when running on that kind of processor. Example:

```
(format t "~&Processor is a ~A.~%"
        (select-processor
          (:cadr "CADR")
          (:lambda "Lambda")))
```

15.18 Microcode Variables

The following variables' values actually reside in the scratchpad memory of the processor. They are put there by dtp-one-q-forward invisible pointers. The values of these variables are used by the microcode. Many of these variables are highly internal and you shouldn't expect to understand them.

%microcode-version-number *Variable*
This is the version number of the currently-loaded microcode, obtained from the version number of the microcode source file.

sys:%number-of-micro-entries *Variable*
Size of micro-code-entry-area and related areas.

default-cons-area is documented on page 297.

sys:number-cons-area *Variable*
The area number of the area where bignums, ratios, full-size floats and complexnums are consed. Normally this variable contains the value of sys:extra-pdl-area, which enables the "temporary storage" feature for numbers, saving garbage collection overhead.

current-stack-group and current-stack-group-resumer are documented on page 258.

sys:%current-stack-group-state *Variable*
The sg-state of the currently-running stack group.

sys:%current-stack-group-calling-args-pointer *Variable*
The argument list of the currently-running stack group.

sys:%current-stack-group-calling-args-number *Variable*
The number of arguments to the currently-running stack group.

sys:%trap-micro-pc *Variable*
The microcode address of the most recent error trap.

sys:%initial-fef *Variable*
The function that is called when the machine starts up. Normally this is the definition of si:lisp-top-level.

sys:%initial-stack-group *Variable*
The stack group in which the machine starts up. Normally this is the initial Lisp Listener window's process's stack group.

sys:%error-handler-stack-group *Constant*
The stack group that receives control when a microcode-detected error occurs. This stack group cleans up, signals the appropriate condition, or assigns a stack group to run the debugger on the erring stack group.

sys:%scheduler-stack-group *Constant*
The stack group that receives control when a sequence break occurs.

sys:%chaos-csr-address *Constant*
A fixnum, the virtual address that maps to the Unibus location of the Chaosnet interface.

%mar-low *Variable*
A fixnum, the inclusive lower bound of the region of virtual memory subject to the MAR feature (see section 30.13, page 750).

%mar-high *Variable*
A fixnum, the inclusive upper bound of the region of virtual memory subject to the MAR feature (see section 30.13, page 750).

sys:%inhibit-read-only *Variable*
If non-nil, you can write into read-only areas. This is used by **fasload**.

self is documented on page 420.

inhibit-scheduling-flag is documented on page 685.

inhibit-scavenging-flag *Variable*
If non-nil, the scavenger is turned off. The scavenger is the quasi-asynchronous portion of the garbage collector, which normally runs during consing operations.

sys:scavenger-ws-enable *Variable*
If this is nil, scavenging can compete for all of the physical memory of the machine. Otherwise, it should be a fixnum, which specifies how much physical memory the scavenger can use: page numbers as high as this number or higher are not available to it.

sys:%region-cons-alarm *Variable*
Increments whenever a new region is allocated.

sys:%page-cons-alarm *Variable*
Increments whenever a new page is allocated.

sys:%gc-flip-ready *Variable*

t while the scavenger is running, nil when there are no pointers to oldspace.

sys:%gc-generation-number *Variable*

A fixnum which is incremented whenever the garbage collector flips, converting one or more regions from newspace to oldspace. If this number has changed, the %pointer of an object may have changed.

sys:%disk-run-light *Constant*

A fixnum, the virtual address of the TV buffer location of the run-light which lights up when the disk is active. This plus 2 is the address of the run-light for the processor. This minus 2 is the address of the run-light for the garbage collector.

sys:%loaded-band *Variable*

A fixnum, the high 24 bits of the name of the disk partition from which virtual memory was booted. Used to create the greeting message.

sys:%disk-blocks-per-track *Variable*
sys:%disk-blocks-per-cylinder *Variable*

Configuration of the disk being used for paging. Don't change these!

sys:%disk-switches is documented on page 284.

sys:%qlaryh *Variable*

This is the last array to be called as a function, remembered for the sake of the function store.

sys:%qlaryl *Variable*

This is the index used the last time an array was called as a function, remembered for the sake of the function store.

sys:%mc-code-exit-vector *Variable*

This is a vector of pointers that microcompiled code uses to refer to quoted constants.

sys:currently-prepared-sheet *Variable*

Used for communication between the window system and the microcoded graphics primitives.

alphabetic-case-affects-string-comparison is documented on page 218.

sys:tail-recursion-flag is documented on page 55.

zunderflow is documented on page 137.

The next four have to do with implementing the metering system described in section 35.3, page 787.

sys:%meter-global-enable *Variable*
t if the metering system is turned on for all stack-groups.

sys:%meter-buffer-pointer *Variable*
A temporary buffer used by the metering system.

sys:%meter-disk-address *Variable*
Where the metering system writes its next block of results on the disk.

sys:%meter-disk-count *Variable*
The number of disk blocks remaining for recording of metering information.

sys:lexical-environment *Variable*
This is the static chain used in the implementation of lexical scoping of variable bindings in compiled code.

sys:amem-evcp-vector *Variable*
No longer used.

background-cons-area is documented on page 297.

sys:self-mapping-table is documented on page 442.

sys:processor-type-code is documented on page 288.

sys:a-memory-location-names *Constant*
A list of all of the above symbols (and any others added after this documentation was written).

15.19 Microcode Meters

Microcode meters are locations in the scratchpad memory which contain numbers. Most of them are used to count events of various sorts. They are accessible only through the functions read-meter and write-meter. They have nothing to do with the Lisp metering tools.

read-meter *name*
Returns the contents of the microcode meter named *name*, which can be a fixnum or a bignum. *name* must be one of the symbols listed below.

write-meter *name value*
Writes *value*, a fixnum or a bignum, into the microcode meter named *name*. *name* must be one of the symbols listed below.

The microcode meters are as follows:

sys:%count-chaos-transmit-aborts *Meter*
The number of times transmission on the Chaosnet was aborted, either by a collision or because the receiver was busy.

sys:%count-cons-work *Meter*
sys:%count-scavenger-work *Meter*
Internal state of the garbage collection algorithm.

sys:%tv-clock-rate *Meter*
The number of TV frames per clock sequence break. The default value is 67., which causes clock sequence breaks to happen about once per second.

sys:%count-first-level-map-reloads *Meter*
The number of times the first-level virtual-memory map was invalid and had to be reloaded from the page hash table.

sys:%count-second-level-map-reloads *Meter*
The number of times the second-level virtual-memory map was invalid and had to be reloaded from the page hash table.

sys:%count-meta-bits-map-reloads *Meter*
The number of times the virtual address map was reloaded to contain only "meta bits", not an actual physical address.

sys:%count-pdl-buffer-read-faults *Meter*
The number of read references to the pdl buffer that were virtual memory references that trapped.

sys:%count-pdl-buffer-write-faults *Meter*
The number of write references to the pdl buffer that were virtual memory references that trapped.

sys:%count-pdl-buffer-memory-faults *Meter*
The number of virtual memory references that trapped in case they should have gone to the pdl buffer, but turned out to be real memory references after all (and therefore were needlessly slowed down).

sys:%count-disk-page-reads *Meter*
The number of pages read from the disk.

sys:%count-disk-page-writes *Meter*
The number of pages written to the disk.

sys:%count-fresh-pages *Meter*
The number of fresh (newly-consed) pages created in core, which would have otherwise been read from the disk.

sys:%count-disk-page-read-operations *Meter*
The number of paging read operations; this can be smaller than the number of disk pages read when more than one page at a time is read.

sys:%count-disk-page-write-operations *Meter*
The number of paging write operations; this can be smaller than the number of disk pages written when more than one page at a time is written.

sys:%count-disk-prepages-used *Meter*
The number of times a page was used after being read in before it was needed.

sys:%count-disk-prepages-not-used *Meter*
The number of times a page was read in before it was needed, but got evicted before it was ever used.

sys:%count-disk-page-write-waits *Meter*
The number of times the machine waited for a page to finish being written out in order to evict the page.

sys:%count-disk-page-write-busys *Meter*
The number of times the machine waited for a page to finish being written out in order to do something else with the disk.

sys:%disk-wait-time *Meter*
The time spent waiting for the disk, in microseconds. This can be used to distinguish paging time from running time when measuring and optimizing the' performance of programs.

sys:%count-disk-errors *Meter*
The number of recoverable disk errors.

sys:%count-disk-recalibrates *Meter*
The number of times the disk seek mechanism was recalibrated, usually as part of error recovery.

sys:%count-disk-ecc-corrected-errors *Meter*
The number of disk errors that were corrected through the error correcting code.

sys:%count-disk-read-compare-differences *Meter*
The number of times a read compare was done, no disk error occurred, but the data on disk did not match the data in memory.

sys:%count-disk-read-compare-rereads *Meter*
The number of times a disk read was done over because after the read a read compare was done and did not succeed (either it got an error or the data on disk did not match the data in memory).

sys:%count-disk-read-compare-rewrites *Meter*

The number of times a disk write was done over because after the write a read compare was done and did not succeed (either it got an error or the data on disk did not match the data in memory).

sys:%disk-error-log-pointer *Meter*

Address of the next entry to be written in the disk error log. The function si:print-disk-error-log (see page 794) prints this log.

sys:%count-aged-pages *Meter*

The number of times the page ager set an age trap on a page, to determine whether it was being referenced.

sys:%count-age-flushed-pages *Meter*

The number of times the page ager saw that a page still had an age trap and hence made it "flushable", a candidate for eviction from main memory.

sys:%aging-depth *Meter*

A number from 0 to 3 that controls how long a page must remain unreferenced before it becomes a candidate for eviction from main memory.

sys:%count-findcore-steps *Meter*

The number of pages inspected by the page replacement algorithm.

sys:%count-findcore-emergencies *Meter*

The number of times no evictable page was found and extra aging had to be done.

sys:a-memory-counter-block-names *Constant*

A list of all of the above symbols (and any others added after this documentation was written).

15.20 Miscellaneous Subprimitives

sys:%halt

Stops the machine.

16. Areas

Storage in the Lisp Machine is divided into *areas*. Each area contains related objects, of any type. Areas are intended to give the user control over the paging behavior of the program, among other things. Putting frequently used data and rarely used data in different areas can cause the frequently used data to occupy fewer pages. For example, the system puts the debugging info alists of compiled functions in a special area so that the other list structure the functions point to will be more compact.

Whenever a new object is created the area to be used can optionally be specified. For example, instead of using `cons` you can use `cons-in-area` (see page 88). Object-creating functions which take keyword arguments generally accept a :area argument. You can also control which area is used by binding `default-cons-area` (see page 297); most functions that allocate storage use the value of this variable, by default, to specify the area to use.

There is a default 'working storage' area that collects those objects that the user has not chosen to control explicitly.

Areas also give the user a handle to control the garbage collector. Some areas can be declared to be *static*, which means that they change slowly and the garbage collector should not attempt to reclaim any space in them. This can eliminate a lot of useless copying. A static area can be explicitly garbage-collected at infrequent intervals when it is believed that that might be worthwhile.

Each area can potentially have a different storage discipline, a different paging algorithm, and even a different data representation. The microcode dispatches on attributes of the area at the appropriate times. The structure of the machine makes the performance cost of these features negligible; information about areas is stored in extra bits in the memory mapping hardware where it can be quickly dispatched on by the microcode; these dispatches usually have to be done anyway to make the garbage collector work and to implement invisible pointers. This feature is not currently used by the system, except for the list/structure distinction described below.

Each area has a name and a number. The name is a symbol whose value is the number. The number is an index into various internal tables. Normally the name is treated as a special variable, so the number is what is given as an argument to a function that takes an area as an argument. Thus, areas are not Lisp objects; you cannot pass an area itself as an argument to a function; you just pass its number. There is a maximum number of areas (set at cold-load generation time); you can only have that many areas before the various internal tables overflow. Currently (as this manual is written) the limit is 256 areas, of which 64 already exist when you start.

The storage of an area consists of one or more *regions*. Each region is a contiguous section of address space with certain homogeneous properties. The most important of these is the *data representation type*. A given region can only store one type. The two types that exist now are *list* and *structure*. A list is anything made out of conses (a closure for instance). A structure is anything made out of a block of memory with a header at the front; symbols, strings, arrays, instances, compiled functions, etc. Since lists and structures cannot be stored in the same region, they cannot be on the same page. It is necessary to know about this when using areas to increase locality of reference.

When you create an area, one region is created initially. When you try to allocate memory to hold an object in some area, the system tries to find a region that has the right data representation type to hold this object, and that has enough room for it to fit. If there isn't any such region, it makes a new one (or signals an error; see the :size option to make-area, below). The size of the new region is an attribute of the area (controllable by the :region-size option to make-area). If regions are too large, memory may get taken up by a region and never used. If regions are too small, the system may run out of regions because regions, like areas, are defined by internal tables that have a fixed size (set at cold-load generation time). Currently (as this manual is written) the limit is 256 regions, of which about 105 already exist when you start. (If you're wondering why the limit on regions isn't higher than the limit on areas, as it clearly ought to be, it's just because both limits have to be multiples of 256 for internal reasons, and 256 regions seem to be enough.)

16.1 Area Functions and Variables

default-cons-area *Variable*

The value of this variable is the number of the area in which objects are created by default. It is initially the number of working-storage-area. Giving nil where an area is required uses the value of default-cons-area. Note that to put objects into an area other than working-storage-area you can either bind this variable or use functions such as cons-in-area (see page 88) which take the area as an explicit argument.

background-cons-area *Variable*

The value of this variable is the number of a non-temporary area in which objects created as incidental side effects by system functions should be created. This area is used whenever an object is created that should never be in a temporary area, even if *default-cons-area* is a temporary area.

By default, this area is working-storage-area.

make-area &key *name size region-size representation gc read-only pdl room*

Creates a new area, whose name and attributes are specified by the keywords. You must specify a symbol as a name; the symbol is setq'ed to the area-number of the new area, and that number is also returned, so that you can use make-area as the initialization of a defvar.

Here are the meanings of the keywords:

name	A symbol that will be the name of the area. This item is required.
size	The maximum allowed size of the area, in words. Defaults to infinite. (Actually, the default is the largest positive fixnum; but the area is not limited to that size!) If the number of words allocated to the area reaches this size, attempting to cons an object in the area will signal an error.
region-size	The approximate size, in words, for regions within this area. The default is the area size if a :size argument was given, otherwise it is a suitable medium size. Note that if you specify :size and not :region-size, the

area will have exactly one region. When making an area that will grow very big, it is desirable to make the region size larger than the default region size to avoid creating very many regions and possibly overflowing the system's fixed-size region tables.

representation	The type of object to be contained in the area's initial region. The argument to this keyword can be :list, :structure, or a numeric code. :structure is the default. If you are only going to cons lists in your area, you should specify :list so you don't get a useless structure region.
gc	The type of garbage-collection to be employed. The choices are :dynamic (which is the default), :static, and :temporary. :static means that the area will not be copied by the garbage collector, and nothing in the area or pointed to by the area will ever be reclaimed, unless a garbage collection of this area is manually requested. :temporary is like :static, but in addition you are allowed to use si:reset-temporary-area on this area.
read-only	With an argument of t, causes the area to be made read-only. Defaults to nil. If an area is read-only, then any attempt to change anything in it (altering a data object in the area or creating a new object in the area) will signal an error unless sys:%inhibit-read-only (see page 290) is bound to a non-nil value.
pdl	With an argument of t, makes the area suitable for storing regular-pdls of stack-groups. This is a special attribute due to the pdl-buffer hardware. Defaults to nil. Areas for which this is nil may *not* be used to store regular-pdls. Areas for which this is t are relatively slow to access; all references to pages in the area will take page faults to check whether the referenced location is really in the pdl-buffer.
room	With an argument of t, adds this area to the list of areas that are displayed by default by the room function (see page 792).

Example:
```
(make-area :name 'foo-area
           :gc :dynamic
           :representation :list)
```

describe-area *area*

area may be the name or the number of an area. Various attributes of the area are printed.

area-list *Variable*

The value of area-list is a list of the names of all existing areas. This list shares storage with the internal area name table, so you should not change it.

%area-number *pointer*

Returns the number of the area to which *pointer* points, or nil if it does not point within any known area. The data-type of *pointer* is ignored.

%region-number *pointer*

Returns the number of the region to which *pointer* points, or nil if it does not point within any known region. The data-type of *pointer* is ignored. (This information is generally not very interesting to users; it is important only inside the system.)

area-name *number*

Given an area number, returns the name. This "function" is actually an array.

si:reset-temporary-area *area-number*

This very dangerous operation marks all the storage in area *area-number* as free and available for re-use. Any data in the area is lost and pointers to it become meaningless. In principle, this operation should only be used if you are sure there are no pointers into the area.

If the area was not defined as temporary, this function gets an error.

See also cons-in-area (page 88), list-in-area (page 93), and room (page 792).

16.2 Interesting Areas

This section lists the names of some of the areas and tells what they are for. Only the ones of the most interest to a user are listed; there are many others.

working-storage-area *Constant*

This is the normal value of default-cons-area. Most working data are consed in this area.

permanent-storage-area *Constant*

This area is to be used for permanent data, which will (almost) never become garbage. Unlike working-storage-area, the contents of this area are not continually copied by the garbage collector; it is a static area.

sys:extra-pdl-area *Constant*

The 'number consing area' in which floating point numbers, ratios and bignums are normally created. If a pointer to a number in this area is stored anywhere outside the machine registers and current stack, a copy of the number is made in working-storage-area and a pointer to the copy is stored instead. When sys:extra-pdl-area gets full, the all numbers pointed to by the registers and current stack are copied, and then nothing in the area can be in use any more, so it is marked as empty.

sys:p-n-string *Constant*
Print-names of symbols are stored in this area.

sys:nr-sym *Constant*
Contains most of the symbols in the Lisp world, except t and nil.

sys:resident-symbol-area *Constant*
Contains the symbols t and nil. nil is known to be at address zero.

sys:pkg-area *Constant*
This area contains packages, principally the hash tables with which intern keeps track of symbols.

macro-compiled-program *Constant*
Compiled functions (FEFs) are put here by the compiler and by fasload. So are the constants that they refer to.

sys:property-list-area *Constant*
This area holds the property lists of symbols.

16.3 Errors Pertaining to Areas

sys:area-overflow (error) *Condition*
This is signaled on an attempt to make an area bigger than its declared maximum size.

The condition instance supports the operations :area-name and :area-maximum-size.

sys:region-table-overflow (error) *Condition*
This is signaled if you run out of regions.

sys:virtual-memory-overflow (error) *Condition*
This is signaled if all of virtual memory is part of some region and an attempt is made to allocate a new region. There may be free space left in some regions in other areas, but there is no way to apply it to the area in which storage is to be allocated.

sys:cons-in-fixed-area (error) *Condition*
This is signaled if an attempt is made add a second region to a fixed area. The fixed areas are certain areas, created at system initialization, that are only allowed a single region, because their contents must be contiguous in virtual memory.

17. The Compiler

17.1 The Basic Operations of the Compiler

The purpose of the Lisp compiler is to convert Lisp functions into programs in the Lisp Machine's instruction set, so that they run more quickly and take up less storage. Compiled functions are represented in Lisp by FEFs (Function Entry Frames), which contain machine code as well as various other information. The printed representation of a FEF is

```
#<DTP-FEF-POINTER address name>
```

If you want to understand the output of the compiler, refer to chapter 31, page 752.

There are three ways to invoke the compiler from the Lisp Machine. First, you may have an interpreted function in the Lisp environment that you would like to compile. The function **compile** is used to do this. Second, you may have code in an editor buffer that you would like to compile. The Zmacs editor has commands to read code into Lisp and compile it. Third, you may have a program (a group of function definitions and other forms) written in a file on the file system. The function **compile-file** can translate this file into a *QFASL* file that describes the compiled functions and associated data. The QFASL file format is capable of representing an arbitrary collection of Lisp objects, including shared structure. The name derives from 'Q', a prefix once used to mean "for the Lisp Machine, not for Maclisp", and 'FASL', an abbreviation for "fast loading".

17.2 How to Invoke the Compiler

compile *function-spec* &optional *definition*

Compiles an individual interpreted function definition. If *definition* is supplied, it is the definition to be compiled. Otherwise, the current definition of *function-spec* is used. If *function-spec* is non-nil, the compiled function is stored as the definition of *function-spec*, and *function-spec* is returned. Otherwise, the compiled function object itself is returned. (However, it is preferable to use **compile-lambda** if your wish is to create a compiled function object without storing it anywhere.)

The compiled function object created by **compile** records the interpreted definition it was made from on its debugging info alist (see page 242). This is useful in two ways: the function **uncompile** can be used to reinstall the interpreted definition, and **compile** invoked again on the same *function-spec* can find the interpreted definition used before and compile it again. The latter is useful if you have changed some macros or subst functions which the definition refers to.

uncompile *function-spec*

If *function-spec* is defined as a compiled function that records the original definition that was compiled, then *function-spec* is redefined with that original definition. This undoes the effect of calling **compile** on *function-spec*.

compile-lambda *lambda-exp function-spec*
Returns a compiled function object produced by compiling *lambda-exp*. The function name recorded by the compiled function object is *function-spec*, but that function spec is not defined by compile-lambda. This function is preferable to compile with first argument nil in that it allows you to specify the name for the function to record internally.

compile-encapsulations *function-spec*
Compiles all encapsulations that function-spec currently has. Encapsulations (see section 11.9, page 244) include tracing, breakons and advice. Compiling tracing or breakons makes it possible (or at least more possible) to trace or breakon certain functions that are used in the evaluator. Compiling advice makes it less costly to advise functions that are used frequently.

Any encapsulation that is changed will cease to be compiled; thus, if you add or remove advice, you must do compile-encapsulations again if you wish the advice to be compiled again.

compile-encapsulations-flag *Variable*
If this is non-nil, all encapsulations that are created are compiled automatically.

compile-file *input-file* &key *output-file set-default-pathname package*
Compiles the file specified by *input-file*, a pathname or namestring. The format for files input to the compiler is described on section 17.3, page 303.

If *output-file* is specified, it is a pathname used for the compiled file. Otherwise, the ouptut file name is computed from the input file name by specifying :qfasl as the type component.

package, if non-nil specifies the package in which compilation should be performed. Normally the system knows, or asks interactively, and you need not supply this argument.

set-default-pathname, if non-nil, means that the defaults should be set to the input file's name. *set-default-pathname* defaults to t.

qc-file *filename* &optional *output-file load-flag in-core-flag package file-local-declarations dont-set-default-p read-then-process-flag*
An older, obsolete way of invoking the compiler on a file.

file-local-declarations is for compiling multiple files as if they were one. *dont-set-default-p* suppresses the changing of the default file name to *filename* that normally occurs. The *load-flag* and *in-core-flag* arguments were not fully implemented and should not be used. *read-then-process-flag* causes the entire file to be read and then the entire file to be compiled; this is no longer advantageous now that there is enough memory to avoid thrashing when forms are read and compiled one by one, and it prevents compile-time reader-macros defined in the file from working properly.

qc-file-load *filename* &optional *output-file load-flag in-core-flag package functions-defined file-local-declarations dont-set-default-p read-then-process-flag*

Compiles a file and then loads in the resulting QFASL file.

compiler:compiler-verbose *Variable*

If this variable is non-nil, the compiler prints the name of each function that it is about to compile.

compiler:peep-enable *Variable*

The peephole optimizer is used if this variable is non-nil. The only reason to set it to nil is if there is a suspicion of a bug in the optimizer.

See also the disassemble function (page 792), which lists the instructions of a compiled function in symbolic form.

17.3 Input to the Compiler

The purpose of compile-file is to take a file and produce a translated version which does the same thing as the original except that the functions are compiled. compile-file reads through the input file, processing the forms in it one by one. For each form, suitable binary output is sent to the QFASL file so that when the QFASL file is loaded the effect of that source form will be reproduced. The differences between source files and QFASL files are that QFASL files are in a compressed binary form, which reads much faster but cannot be edited, and that function definitions in QFASL files have been translated from Lisp forms to FEFs.

So, if the source contains a (defun ...) form at top level, then when the QFASL file is loaded the function will be defined as a compiled function. If the source file contains a form that is not of a type known specially to the compiler, then that form (encoded in QFASL format) is output "directly" into the QFASL file, so that when the QFASL file is loaded that form will be evaluated. Thus, if the source file contains (princ "Hello") at top level, then the compiler puts in the QFASL file instructions to create the list (princ "Hello") and then evaluate it.

The Lisp Machine editor Zmacs assumes that source files are formatted so that an open parenthesis at the left margin (that is, in column zero) indicates the beginning of a function definition or other top level list (with a few standard exceptions). The compiler assumes that you follow this indentation convention, enabling it to tell when a close-parenthesis is missing from one function as soon as the beginning of the next function is reached.

If the compiler finds an open parenthesis in column zero in the middle of a list, it invents enough close parentheses to close off the list that is in progress. A compiler warning is produced instead of an error. After that list has been processed, the open parenthesis is read again. The compilation of the list that was forcefully closed off is probably useless, but the compilation of the rest of the file is usually correct. You can read the source file into the editor to fix and recompile the function that was unbalanced.

A similar thing happens on end of file in the middle of a list, so that you get to see any warnings for the function that was unbalanced.

Certain special forms including eval-when, progn, local-declare, declare-flavor-instance-variables, and comment are customarily used around lists that start in column zero. These symbols have a non-nil si:may-surround-defun property that makes the compiler permit this. You can add such properties to other symbols if you want.

compiler:qc-file-check-indentation *Variable*

If nil, inhibits the compiler from checking for open-parentheses in column zero.

When a macro definition (macro and defmacro forms) is encountered at top level in the file being compiled, the macro definition is recorded for the rest of the compilation so that the macro thus defined can be used in the same file following its definition. This is in addition to writing the compiled macro definition into the QFASL file.

Flavor definitions (defflavor forms, see page 414) and global special declarations (made with proclaim, page 54, or with defvar, page 33) are likewise recorded for the rest of the compilation, as well as written into the QFASL file so that they will be recorded permanently when the file is loaded.

sys:file-local-declarations *Variable*

During file-to-file compilation, the value of this variable is a list of all declarations that are in effect for the rest of the file. Macro definitions, defdecl's, proclaim's and special declarations that come from defvars are all recorded on this list.

Package-defining and altering functions such as defpackage, in-package, export and use-package are executed by the compiler in the ordinary, permanent fashion. They are also written in the QFASL file so that the form is executed just the same when the file is loaded. If you load the file later in the same session, the package altering form is executed twice. This is normally harmless. require receives the same treatment.

You can control explicitly whether a form is evaluated by the compiler, and whether it is written into the QFASL file to be executed when the file is loaded, using the eval-when construct. You might want a form to be:

Put into the QFASL file (compiled, of course), or not.

Evaluated within the compiler, or not.

Evaluated if the source file loaded, or not.

An eval-when form looks like

(eval-when *times-list*
  form1 form2 ...)

The *times-list* may contain one or more of the symbols load, compile, or eval. If load is present, the *forms* are written into the QFASL file to be evaluated when the QFASL file is loaded (except that defun forms put the compiled definition into the QFASL file instead). If compile is present, the *forms* are evaluated in the compiler. If eval is present, the *forms* are evaluated when read into Lisp; this is because eval-when is defined as a special form in Lisp. (The compiler ignores eval in the *times-list*.) For example,

```
(eval-when (compile eval) (macro foo (x) (cadr x)))
```

would define foo as a macro in the compiler and when the file is read in interpreted, but not when the QFASL file is fasloaded.

eval-when (*time*...) *body*... *Special form*

When seen by the interpreter, if one of the *times* is the symbol **eval** then the *body* forms are evaluated; otherwise **eval-when** does nothing.

But when seen by the compiler, this special form does the special things described above.

Nested use of **eval-when** is permitted but its meaning is tricky. If an inner **eval-when** form appears in an ordinary context where a general form would be written into the QFASL file but not executed at compile time, then it behaves in the usual fashion: the *body* forms are written into the QFASL file if **load** is one of the *times*, and they are evaluated at compile time if **compile** is one of the *times*.

If the inner **eval-when** form appears in a context which says to evaluate at compile time only, then the *body* forms are evaluated if **eval** is one of the *times*.

If the inner **eval-when** appears in a context which says to write into the QFASL file and evaluate at compile time. the the *body* forms are written into the QFASL file if **load** is one of the *times*, and they are evaluated at compile time if either **compile** or **eval** is one of the *times*.

For the rest of this section, we will use lists such as are given to **eval-when**, e.g. **(load eval)**, **(load compile)**, etc., to describe when forms are evaluated.

If a form is not enclosed in an **eval-when**, then the times at which it is evaluated depend on the form. The following table summarizes at what times evaluation takes place for any given form seen at top level by the compiler.

(eval-when *times-list form* **...)**
times-list specifies when the *form*... should be performed.

(declare (special ...)) or **(declare (unspecial ...))**
The **special** or **unspecial** is performed at **(load compile)** time.

(declare *anything-else***)**
anything-else is performed only at **(compile)** time.

(proclaim ...) is performed at **(load compile eval)** time.

(special ...) or **(unspecial ...)**
(load compile eval)

(macro ...) or **(defmacro ...)** or **(defsubst ...)**
or **(defflavor ...)** or **(defstruct ...)**
(load eval). However, during file to file compilation, the definition is recorded temporarily and used for expanding calls to the macro, or macros defined by the **defstruct** for the rest of the file.

(comment ...) Ignored at all times.

(compiler-let ((*var val***) ...)** *body*...**)**
Processes the *body* in its normal fashion, but with the indicated variable bindings in effect. These variables will typically affect the operation of the compiler or of macros. See section 18.5.6, page 339.

(local-declare (*decl decl ...*) *body...*)
: Processes the *body* in its normal fashion, with the indicated declarations added to the front of the list which is the value of local-declarations.

(defun ...) or (defmethod ...) or (defselect ...)
: (load eval), but at load time what is processed is not this form itself, but the result of compiling it.

(require ...) or (in-package ...)
or various other package functions
: (load compile eval)

anything-else (load eval)

Sometimes a macro wants to return more than one form for the compiler top level to see (and to be evaluated). The following facility is provided for such macros. If a form

```
(progn form1 form2 ...)
```

is seen at the compiler top level, all of the *forms* are processed as if they had been at compiler top level. (Of course, in the interpreter they are all evaluated.)

To prevent an expression from being optimized by the compiler, surround it with a call to dont-optimize.

dont-optimize *form* *Special form*

In execution, this is equivalent to simply *form*. However, any source-level optimizations that the compiler would normally perform on the top level of *form* are not done.
Examples:

```
(dont-optimize (apply 'foo (list 'a 'b)))
```

actually makes a list and calls apply, rather than doing

```
(foo 'a 'b)

(dont-optimize (si:flavor-method-table flav))
```

actually calls si:flavor-method-table as a function, rather than substituting the definition of that defsubst.

dont-optimize can even be used around a defsubst inside of setf or locf, to prevent open-coding of the defsubst. In this case, a function will be created at load time to do the setting or return the location.

```
(setf (dont-optimize (zwei:buffer-package buffer))
      (pkg-find-package "foo"))
```

Subforms of *form*, such as arguments, are still optimized or open coded, unless additional dont-optimize's appear around them.

17.4 Compile-Time Properties of Symbols

When symbol properties are referred to during macro expansion, it is desirable for properties defined in a file to be "in effect" for the the rest of the file if the file is compiled. This does not happen if get and defprop are used, because the defprop will not be executed until the QFASL file is loaded. Instead, you can use getdecl and defdecl. These are normally the same as get and defprop, but during file-to-file compilation they also refer to and create declarations.

getdecl *symbol property*

This is a version of get that allows the properties of the *symbol* to be overridden by declarations.

If a declaration of the form (*property symbol value*) is in effect, getdecl returns *value*. Otherwise, getdecl returns the result of (get *symbol property*).

If you intend to create such declarations with proclaim or local use of declare, you must make sure that a declaration declaration is in effect for *property*. You can do this with (proclaim '(declaration *property*)).

getdecl is typically used in macro definitions. For example, the setf macro uses getdecl to get the properties which say how to store in the specified place. See page 340 for an example of a macro that uses getdecl.

putdecl *symbol property value*

Causes (getdecl *symbol property*) to return *value*.

putdecl usually simply does a putprop. But if executed at compile time during file-to-file compilation, it instead makes an entry on file-local-declarations of the form (*property symbol value*).

In either case, this stores *value* where getdecl can find it; but if putdecl is done during compilation, it affects only the rest of that compilation.

defdecl *symbol property value* *Special form*

When executed, this is like putdecl except that the arguments are not evaluated. It is usually the same as defprop except for the order of the arguments.

Unlike defprop, when defdecl is encountered during file-to-file compilation, a declaration is recorded which remains in effect for the rest of the compilation. (The defdecl form also goes into the QFASL file to be executed when the file is loaded). defprop would have no effect whatever at compile time.

defdecl is often useful as a part of the expansion of a macro. It is also useful as a top-level expression in a source file.

Example:
```
        (defdecl foo locf foo-location)
```
in a source file would allow (locf (foo *args*...)) to be used in the rest of that source file; and, once the file was loaded, by anyone.

Simple use defsetf expands into a defdecl.

17.5 Using Compiler Warnings

When the compiler prints warnings, it also records them in a data base, organized by file and by function within file. Old warnings for previous compilations of the same function are thrown away, so the data base contains only warnings that are still applicable. This data base can be used to visit, in the editor, the functions that got warnings. You can also save the data base and restore it later.

There are three editor commands that you can use to begin visiting the sites of the recorded warnings. They differ only in how they decide which files to look through:

Meta-X Edit Warnings
: For each file that has any warnings, asks whether to edit the warnings for that file.

Meta-X Edit File Warnings
: Reads the name of a file and then edits the warnings for that file.

Meta-X Edit System Warnings
: Reads the name of a system and then edits the warnings for all files in that system (see defsystem, page 660).

While the warnings are being edited, the warnings themselves appear in a small window at the top of the editor frame, and the code appears in a large window which occupies the rest of the editor frame.

As soon as you have finished specifying the file(s) or system to process, the editor proceeds to visit the code for the first warning. From then on, to move to the next warning, use the command Control-Shift-W. To move to the previous warning, use Meta-Shift-W. You can also switch to the warnings window with Control-X O or with the mouse, and move around in that buffer. When you use Control-Shift-W and there are no more warnings after the cursor, you return to single-window mode.

You can also insert the text of the warnings into any editor buffer:

Meta-X Insert File Warnings
: Reads the name of a file and inserts into the buffer after point the text for that file's warnings. The mark is left after the warnings, but the region is not turned on.

Meta-X Insert Warnings
: Inserts into the buffer after point the text for the warnings of all files that have warnings. The mark is left after the warnings, but the region is not turned on.

You can also dump the warnings data base into a file and reload it later. Then you can do Meta-X Edit Warnings again in the later session. You dump the warnings with si:dump-warnings and load the file again with load. In addition, make-system with the :batch option writes all the warnings into a file in this way.

si:dump-warnings *output-file-pathname* &rest *warnings-file-pathnames*

Writes the warnings for the files named in *warnings-file-pathnames* (a list of pathnames or strings) into a file named *output-file-pathname*.

compiler:warn-on-errors *Variable*

If this variable is non-nil, errors in reading code to be compiled, and errors in macro expansion within the compiler, produce only warnings; they do not enter the debugger. The variable is normally **t**.

The default setting is useful when you do not anticipate errors during compilation, because it allows the compilation to proceed past such errors. If you have walked away from the machine, you do not come back to find that your compilation stopped in the first file and did not finish.

If you find an inexplicable error in reading or macroexpansion, and wish to use the debugger to localize it, set **compiler:warn-on-errors** to **nil** and recompile.

17.5.1 Controlling Compiler Warnings

By controlling the compile-time values of the variables **run-in-maclisp-switch**, **obsolete-function-warning-switch**, and **inhibit-style-warning-switch** (explained above), you can enable or disable some of the warning messages of the compiler. The following special form is also useful:

inhibit-style-warnings *form* *Macro*

Prevents the compiler from performing style-checking on the top level of *form*. Style-checking is still done on the arguments of *form*. Both obsolete function warnings and won't-run-in-Maclisp warnings are done by means of the style-checking mechanism, so, for example,

```
        (setq bar (inhibit-style-warnings (value-cell-location foo)))
```

does not warn that **value-cell-location** will not work in Maclisp, but

```
        (inhibit-style-warnings (setq bar (value-cell-location foo)))
```

does warn, since **inhibit-style-warnings** applies only to the top level of the form inside it (in this case, to the **setq**).

Sometimes functions take arguments that they deliberately do not use. Normally the compiler warns you if your program binds a variable that it never references. In order to disable this warning for variables that you know you are not going to use, there are three things you can do.

The first thing is to name the variables **ignore** or **ignored**. The compiler does not complain if a variable by one of these names is not used. Furthermore, by special dispensation, it is all right to have more than one variable in a lambda-list that has one of these names.

Another thing you can do is write an **ignore** declaration. Example:

```
    (defun the-function (list fraz-name fraz-size)
      (declare (ignore fraz-size)))
```

This has the advantage that **arglist** (see page 242) will return a more meaningful argument list for the function, rather than returning something with **ignore**'s in it.

Finally, you can simply use the variable for effect (ignoring its value) at the front of the function. Example:

```
(defun the-function (list fraz-name fraz-size)
  fraz-size       ; This argument is not used.
  ...)
```

The following function is useful for requesting compiler warnings in certain esoteric cases. Normally, the compiler notices whenever any function *x* uses (calls) any other function *y*; it makes notes of all these uses, and then warns you at the end of the compilation if the function *y* got called but no definition of it has been seen. This usually does what you want, but sometimes there is no way the compiler can tell that a certain function is being used. Suppose that instead of *x*'s containing any forms that call *y*, *x* simply stores *y* away in a data structure somewhere, and someplace else in the program that data structure is accessed and funcall is done on it. There is no way that the compiler can see that this is going to happen, and so it can't notice the function usage, and so it can't create a warning message. In order to make such warnings happen, you can explicitly call the following function at compile-time.

compiler:function-referenced *what by*

what is a symbol that is being used as a function. *by* may be any function spec. compiler:function-referenced must be called at compile-time while a compilation is in progress. It tells the compiler that the function *what* is referenced by *by*. When the compilation is finished, if the function *what* has not been defined, the compiler issues a warning to the effect that *by* referred to the function *what*, which was never defined.

You can also tell the compiler about any function it should consider "defined":

compiler:compilation-define *function-spec*

function-spec is marked as "defined" for the sake of the compiler; future calls to this function will not produce warnings.

compiler:make-obsolete *function reason* *Macro*

This special form declares a function to be obsolete; code that calls it will get a compiler warning, under the control of obsolete-function-warning-switch. This is used by the compiler to mark as obsolete some Maclisp functions which exist in Zetalisp but should not be used in new programs. It can also be useful when maintaining a large system, as a reminder that a function has become obsolete and usage of it should be phased out. An example of an obsolete-function declaration is:

```
(compiler:make-obsolete create-mumblefrotz
        "use MUMBLIFY with the :FROTZ option instead")
```

17.5.2 Recording Warnings

The warnings data base is not just for compilation. It can record operations for any number of different operations on files or parts of files. Compilation is merely the only operation in the system that uses it.

Each operation about which warnings can be recorded should have a name, preferably in the keyword package. This symbol should have four properties that tell the system how to print out the operation name as various parts of speech. For compilation, the operation name is :compile and the properties are defined as follows:

```
(defprop :compile "compilation" si:name-as-action)
(defprop :compile "compiling" si:name-as-present-participle)
(defprop :compile "compiled" si:name-as-past-participle)
(defprop :compile "compiler" si:name-as-agent)
```

The warnings system considers that these operations are normally performed on files that are composed of named objects. Each warning is associated with a filename and then with an object within the file. It is also possible to record warnings about objects that are not within any file.

To tell the warnings system that you are starting to process all or part of a file, use the macro si:file-operation-with-warnings.

sys:file-operation-with-warnings *Macro*
(*generic-pathname operation-name whole-file-p*) *body...*

body is executed within a context set up so that warnings can be recorded for operation *operation-name* about the file specified by *generic-pathname* (see page 563).

In the case of compilation, this is done at the level of compile-file (actually, it is done in compiler:compile-stream).

whole-file-p should be non-nil if the entire contents of the file are to be processed inside the *body* if it finishes; this implies that any warnings left over from previous iterations of this operation on this file should be thrown away on exit. This is only relevant to objects that are not found in the file this time; the assumption is that the objects must have been deleted from the file and their warnings are no longer appropriate.

All three of the special arguments are specified as expressions that are evaluated.

Within the processing of a file, you must also announce when you are beginning to process an object:

sys:object-operation-with-warnings (*object-name location-function*) *body...Macro*

Executes *body* in a context set up so that warnings are recorded for the object named *object-name*, which can be a symbol or a list. Object names are compared with **equal**.

In the case of compilation, this macro goes around the processing of a single function.

location-function is either **nil** or a function that the editor uses to find the text of the object. Refer to the file SYS: ZWEI; POSS LISP for more details on this.

object-name and *location-function* are specified with expressions that are evaluated.

You can enter this macro recursively. If the inner invocation is for the same object as the outer one, it has no effect. Otherwise, warnings recorded in the inner invocation apply to the object specified therein.

Finally, when you detect exceptions, you must make the actual warnings:

sys:record-warning *type severity location-info format-string* &rest *args*
Records one warning for the object and file currently being processed. The text of the warning is specified by *format-string*and *args*, which are suitable arguments for **format**, but the warning is *not* printed when you call this function. Those arguments will be used to reprint the warning later.

sys:record-and-print-warning *type severity location-info format-string* &rest *args*
Records a warning and also prints it.

type is a symbol that identifies the specific cause of the warning. Types have meaning only as defined by a particular operation, and at present nothing makes much use of them. The system defines one type: si:premature-warnings-marker.

severity measures how important a warning this is, and the general causal classification. It should be a symbol in the keyword package. Several severities are defined, and should be used when appropriate, but nothing looks at them:

:implausible	This warning is about something that is not intrinsically wrong but is probably due to a mistake of some sort.
:impossible	This warning is about something that cannot have a meaning even if circumstances outside the text being processed are changed.
:probable-error	This is used to indicate something that is certainly an error but can be made correct by a change somewhere else; for example, calling a function with the wrong number of arguments.
:missing-declaration	This is used for warnings about free variables not declared special, and such. It means that the text was not actually incorrect, but something else that is supposed to accompany it was missing.
:obsolete	This warning is about something that you shouldn't use any more, but which still does work.
:very-obsolete	This is about something that doesn't even work any more.
:maclisp	This is for something that doesn't work in Maclisp.
:fatal	This indicates a problem so severe that no sense can be made of the object at all. It indicates that the presence or absence of other warnings is

not significant.

:error	There was a Lisp error in processing the object.

location-info is intended to be used to inform the editor of the precise location in the text of the cause of this warning. It is not defined as yet, and you should use nil.

If a warning is encountered while processing data that doesn't really have a name (such as forms in a source file that are not function definitions), you can record a warning even though you are not inside an invocation of sys:object-operation-with-warnings. This warning is known as a *premature warning* and it will be recorded with the next object that is processed; a message will be added so that the user can tell which warnings were premature.

Refer to the file SYS: SYS; QNEW LISP for more information on the warnings data base.

17.6 Compiler Source-Level Optimizers

The compiler stores optimizers for source code on property lists so as to make it easy for the user to add them. An optimizer can be used to transform code into an equivalent but more efficient form (for example, (eq *obj* nil) is transformed into (null *obj*), which can be compiled better). An optimizer can also be used to tell the compiler how to compile a special form. For example, in the interpreter do is a special form, implemented by a function which takes quoted arguments and calls eval. In the compiler, do is expanded in a macro-like way by an optimizer into equivalent Lisp code using prog, cond, and go, which the compiler understands.

The compiler finds the optimizers to apply to a form by looking for the compiler:optimizers property of the symbol that is the car of the form. The value of this property should be a list of optimizers, each of which must be a function of one argument. The compiler tries each optimizer in turn, passing the form to be optimized as the argument. An optimizer that returns the original form unchanged (eq to the argument) has "done nothing", and the next optimizer is tried. If the optimizer returns anything else, it has "done something", and the whole process starts over again.

Optimizers should not be used to define new language features, because they only take effect in the compiler; the interpreter (that is, the evaluator) doesn't know about optimizers. So an optimizer should not change the effect of a form; it should produce another form that does the same thing, possibly faster or with less memory or something. That is why they are called optimizers. In principle, the code ought to compile just as correctly if the optimizer is eliminated.

compiler:add-optimizer *function optimizer optimized-into...* *Macro*

Puts *optimizer* on *function*'s optimizers list if it isn't there already. *optimizer* is the name of an optimization function, and *function* is the name of the function calls which are to be processed. Neither is evaluated.

(compiler:add-optimizer *function optimizer optimize-into-1 optimize-into-2...*) also remembers *optimize-into-1*, etc., as names of functions which may be called in place of *function* as a result of the optimization. Then who-calls of *function* will also mention callers of *optimize-into-1*, etc.

compiler:defoptimizer *Macro*
function optimizer-name (optimizes-into...) lambda-list body...
Defines an optimizer and installs it. Equivalent to
```
(progn
  (defun optimizer-name lambda-list
    body...)
  (compiler:add-optimizer function optimizer-name
                                    optimizes-into...))
```

compiler:defcompiler-synonym *function for-function* *Macro*
Makes *function* a synonym for *for-function* in code being compiled. Example:
```
(compiler:defcompiler-synonym plus +)
```
is how the compiler is told how to compile **plus**.

17.7 Maclisp Compatibility

Certain programs are intended to be run both in Maclisp and in Zetalisp. Their source files need some special conventions. For example, all special declarations must be enclosed in declare's, so that the Maclisp compiler will see them. The main issue is that many functions and special forms of Zetalisp do not exist in Maclisp. It is suggested that you turn on run-in-maclisp-switch in such files, which will warn you about a lot of problems that your program may have if you try to run it in Maclisp.

The macro-character combination #+lispm causes the object that follows it to be visible only when compiling for Zetalisp. The combination #+maclisp causes the following object to be visible only when compiling for Maclisp. These work both on subexpressions of the objects in the file and at top level in the file. To conditionalize top-level objects, however, it is better to put the macros if-for-lispm and if-for-maclisp around them. The if-for-lispm macro turns off run-in-maclisp-switch within its object, preventing spurious warnings from the compiler. The #+lispm reader construct does not dare do this, since it can be used to conditionalize any object, not just a expression that will be evaluated.

To allow a file to detect what environment it is being compiled in, the following macros are provided:

if-for-lispm *form* *Macro*
If (if-for-lispm *form*) is seen at the top level of the compiler, *form* is passed to the compiler top level if the output of the compiler is a QFASL file intended for Zetalisp. If the Zetalisp interpreter sees this it evaluates *form* (the macro expands into *form*).

if-for-maclisp *form* *Macro*
If (if-for-maclisp *form*) is seen at the top level of the compiler, *form* is passed to the compiler top level if the output of the compiler is a FASL file intended for Maclisp (e.g. if the compiler is COMPLR). If the Zetalisp interpreter ignores this form entirely (the macro expands into nil).

if-for-maclisp-else-lispm *maclisp-form lispm-form* *Macro*

If (if-for-maclisp-else-lispm *form1 form2*) is seen at the top level of the compiler, *form1* is passed to the compiler top level if the output of the compiler is a FASL file intended for Maclisp; otherwise *form2* is passed to the compiler top level.

if-in-lispm *form* *Macro*

In Zetalisp, (if-in-lispm *form*) causes *form* to be evaluated; in Maclisp, *form* is ignored.

if-in-maclisp *form* *Macro*

In Maclisp, (if-in-maclisp *form*) causes *form* to be evaluated; in Zetalisp, *form* is ignored.

In order to make sure that those macros are defined when reading the file into the Maclisp compiler, you must make the file start with a prelude, which should look like:

```
(eval-when (compile)
  (cond ((not (status feature lispm))
         (load '|PS:<L.SYS2>CONDIT.LISP|))))
          ;; Or other suitable filename
```

This does nothing when you compile the program on the Lisp Machine. If you compile it with the Maclisp compiler, it loads in definitions of the above macros, so that they will be available to your program. The form (status feature lispm) is generally useful in other ways; it evaluates to t when evaluated on the Lisp Machine and to nil when evaluated in Maclisp.

There are some advertised variables whose compile-time values affect the operation of the compiler. Mostly these are for Maclisp compatibility features. You can set these variables by including in his file forms such as

```
(eval-when (compile) (setq open-code-map-switch t))
```

However, these variables seem not to be needed very often.

run-in-maclisp-switch *Variable*

If this variable is non-nil, the compiler tries to warn the user about any constructs that will not work in Maclisp. By no means all Lisp Machine system functions not built in to Maclisp cause warnings; only those that could not be written by the user in Maclisp (for example, make-array, value-cell-location, etc.). Also, lambda-list keywords such as &optional and initialized prog variables are be mentioned. This switch also inhibits the warnings for obsolete Maclisp functions. The default value of this variable is nil.

obsolete-function-warning-switch *Variable*

If this variable is non-nil, the compiler tries to warn the user whenever an obsolete Maclisp-compatibility function such as maknam or samepnamep is used. The default value is t.

allow-variables-in-function-position-switch *Variable*

If this variable is non-nil, the compiler allows the use of the name of a variable in function position to mean that the variable's value should be funcall'ed. This is for compatibility with old Maclisp programs. The default value of this variable is nil.

open-code-map-switch *Variable*

If this variable is non-nil, the compiler attempts to produce inline code for the mapping functions (mapc, mapcar, etc., but not mapatoms) if the function being mapped is an anonymous lambda-expression. The generated code is faster but larger. The default value is t.

If you want to turn off open coding of these functions, it is preferable to use (declare (notinline mapc mapcar ...)).

inhibit-style-warnings-switch *Variable*

If this variable is non-nil, all compiler style-checking is turned off. Style checking is used to issue obsolete function warnings, won't-run-in-Maclisp warnings, and other sorts of warnings. The default value is nil. See also the inhibit-style-warnings macro, which acts on one level only of an expression.

compiler-let ((*variable value*)...) *body...* *Macro*

Allows local rebinding of global switches that affect either compilation or the behavior of user-written macros. Its syntax is like that of let, and in the interpreter it is identical to let. When encountered in compiled code, the variables are bound around the compilation of *body* rather than around the execution at a later time of the compiled code for *body*. For example,
Example:
```
        (compiler-let ((open-code-map-switch nil))
          (mapc (function (lambda (x) ...)) foo))
```
prevents the compiler from open-coding the mapc.

The same results can be obtained more cleanly using declare. User-written macros can examine the declarations using getdecl.

The next three functions are primarily for Maclisp compatibility. In Maclisp, they are declarations, used within a declare at top level in the file.

***expr** *symbol...* *Special form*

Declares each *symbol* to be the name of a function. In addition it prevents these functions from appearing in the list of functions referenced but not defined, printed at the end of the compilation.

***lexpr** *symbol...* *Special form*

Declares each *symbol* to be the name of a function. In addition it prevents these functions from appearing in the list of functions referenced but not defined, printed at the end of the compilation.

***fexpr** *symbol...* *Special form*

Declares each *symbol* to be the name of a special form. In addition it prevents these names from appearing in the list of functions referenced but not defined, printed at the end of the compilation.

17.8 Putting Data in QFASL Files

It is possible to make a QFASL file containing data, rather than a compiled program. This can be useful to speed up loading of a data structure into the machine, as compared with reading in printed representations. Also, certain data structures such as arrays do not have a convenient printed representation as text, but can be saved in QFASL files. For example, the system stores fonts this way. Each font is in a QFASL file (on the SYS: FONTS; directory) that contains the data structures for that font. When the file is loaded, the symbol that is the name of the font gets set to the array that represents the font. Putting data into a QFASL file is often referred to as "*fasdumping* the data".

In compiled programs, the constants are saved in the QFASL file in this way. The compiler optimizes by making constants that are **equal** become **eq** when the file is loaded. This does not happen when you make a data file yourself; identity of objects is preserved. Note that when a QFASL file is loaded, objects that were **eq** when the file was written are still **eq**; this does not normally happen with text files.

The following types of objects can be represented in QFASL files: Symbols (uninterned or uninterned), numbers of all kinds, lists, strings, arrays of all kinds, named structures, instances, and FEFs.

:fasd-form *Operation on instances*

When an instance is fasdumped (put into a QFASL file), it is sent a **:fasd-form** message, which must return a Lisp form that, when evaluated, will recreate the equivalent of that instance. This is because instances are often part of a large data structure, and simply fasdumping all of the instance variables and making a new instance with those same values is unlikely to work. Instances remain **eq**; the **:fasd-form** message is only sent the first time a particular instance is encountered during writing of a QFASL file. If the instance does not accept the **:fasd-form** message, it cannot be fasdumped.

Loading a QFASL file in which a named structure has been fasdumped creates a new named structure with components identical to those of the one that was dumped. Then the **:fasd-fixup** operation is invoked, which gives the new structure the opportunity to correct its contents if they are not supposed to be just the same as what was dumped.

The meaning of a QFASL file is greatly affected by the package used for loading it. Therefore, the file itself says which package to use.

In **dump-forms-to-file**, you can specify the package to use by including a **:package** attribute in the *attribute-list* argument. For example, if that argument is the list **(:package "SI")** then the file is dumped and loaded in the **si** package. If the package is not specified in this way, **user** is used. The other fasdumping functions always use **user**.

dump-forms-to-file *filename forms-list* &optional *attribute-list*

Writes a QFASL file named *filename* which contains, in effect, the forms in *forms-list*. That is to say, when the file is loaded, its effect will be the same as evaluating those forms.

Example:
```
(dump-forms-to-file "foo" '((setq x 1) (setq y 2)))
(load "foo")
x => 1
y => 2
```

attribute-list is the file attribute list to store in the QFASL file. It is a list of alternating keywords and values, and corresponds to the `-*-` line of a source file. The most useful keyword in this context is :package, whose value in the attribute list specifies the package to be used both in dumping the forms and in loading the file. If no :package keyword is present, the file will be loaded in whatever package is current at the time.

compiler:fasd-symbol-value *filename symbol*

Writes a QFASL file named *filename* which contains the value of *symbol*. When the file is loaded, *symbol* will be setq'ed to the same value. *filename* is parsed and defaulted with the default pathname defaults. The file type defaults to :qfasl.

compiler:fasd-font *name*

Writes the font named *name* into a QFASL file with the appropriate name (on the SYS: FONTS; directory).

compiler:fasd-file-symbols-properties *filename symbols properties dump-values-p dump-functions-p new-symbol-function*

This is a way to dump a complex data structure into a QFASL file. The values, the function definitions, and some of the properties of certain symbols are put into the QFASL file in such a way that when the file is loaded the symbols will be setqed, fdefined, and putpropped appropriately. The user can control what happens to symbols discovered in the data structures being fasdumped.

filename is the name of the file to be written. It is defaulted with the default pathname defaults. The file type defaults to "QFASL".

symbols is a list of symbols to be processed. *properties* is a list of properties which are to be fasdumped if they are found on the symbols. *dump-values-p* and *dump-functions-p* control whether the values and function definitions are also dumped.

new-symbol-function is called whenever a new symbol is found in the structure being dumped. It can do nothing, or it can add the symbol to the list to be processed by calling compiler:fasd-symbol-push. The value returned by *new-symbol-function* is ignored.

17.9 Analyzing QFASL Files

QFASL files are composed of 16-bit nibbles. The first two nibbles in the file contain fixed values, which are there so the system can tell a proper QFASL file. The next nibble is the beginning of the first *group*. A group starts with a nibble that specifies an operation. It may be followed by other nibbles that are arguments.

Most of the groups in a QFASL file are there to construct objects when the file is loaded. These objects are recorded in the *fasl-table*. Each time an object is constructed, it is assigned the next sequential index in the fasl-table. The indices are used by other groups later in the file, to refer back to objects already constructed.

To prevent the fasl-table from becoming too large, the QFASL file can be divided into *whacks*. The fasl-table is cleared out at the beginning of each whack.

The other groups in the QFASL file perform operations such as evaluating a list previously constructed or storing an object into a symbol's function cell or value cell.

If you are having trouble with a QFASL file and want to find out exactly what it does when it is loaded, you can use UNFASL to find out.

si:unfasl-print *input-file-name*
Prints on ***standard-output*** a description of the contents of the QFASL file *input-file-name*.

si:unfasl-file *input-file-name* &optional *output-file-name*
Writes a description of the contents of the QFASL file *input-file-name* into the output file. The output file type defaults to **:unfasl** and the rest of the pathname defaults from *input-file-name*.

18. Macros

18.1 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 (i.e. that cons) is called a *macro*. The cdr of the cons should be a function of two arguments. **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 ignore)
               (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 arguments the original form that **eval** was handed, and an environment data structure that this macro does not use. So it calls **(lambda (x ignore) (list 'car (cadr x)))** and the first argument is **(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 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 (form-arg env-arg)
            body)
```

name can be any function spec. *form-arg* and *env-arg* must be variables. *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 ignore)
        (list 'car (cadr x)))
```

Only sophisticated macros need to use value passed for the *env-arg*; this one does not need it, so the argument variable **ignore** is used for it. See page 324 for information on it.

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 ignore)
   (list 'plus '1 (cadr x)))
```

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

```
(macro increment (x ignore)
     (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 **setf** (page 36); but 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 present in a call to it; 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 a user wants some kind of control structure with a syntax that is not provided, he can translate it into some form that Lisp *does* know about.

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

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

To get this, he 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 ignore)
  (list* 'do
         (list (list (second x) (third x)
                     (list '1+ (second x))))
         (list (list '> (second x) (fourth x)))
         (cddddr x)))
```

for can now be used as if it were a built-in Lisp control construct.

18.2 Aids for Defining Macros

The main problem with the definition for the **for** macro is that it is verbose and clumsy. 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 the programmer must write things like (second x) and (cddddr x) to refer to the parts of the form he wants 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.

18.2.1 Defmacro

Instead of referring to the parts of our form by (second x) and such, we would like to give names to the various pieces of the form, and somehow have the (second 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)
      (list* 'do
             (list (list var lower (list '1+ var)))
             (list (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 two lists

```
    (var lower upper . body)
```

and

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

var is 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))). var, lower, upper, and body are then used by the body of the macro definition.

defmacro *Macro*

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

(defmacro *name pattern* . *body*)

name is the name of the macro to be defined; it can be any function spec (see section 11.2, page 223). Normally it is not useful to define anything but a symbol, since that is the only place that the evaluator looks for macro definitions. However, sometimes it is useful to define a :property function spec as a macro, when some part of the system (such as locf) will look for an expander function on a property.

The *pattern* may be anything made up out of symbols and conses. When the macro is called, *pattern* 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*. *body* is evaluated with these bindings in effect, and its result is returned to the evaluator as the expansion of the macro.

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.

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. Subpatterns of the lambda list pattern can also use &-keywords, a usage not allowed in functions.

&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 ambigous. The pattern must at least be enclosed in a singleton list. If *variable* is a pattern, *default* can be evaluated more than once. Example:

```
(defmacro foo (&optional ((x &optional y) '(a)))
  ...)
```

Here the first argument of foo is optional, and should be a list of one or two elements which become x and y. If foo is given no arguments, the list (a) is decomposed to get x and y, so that x's value is a and y's value is nil.

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.

When &key is used in a defmacro pattern, the keywords are decoded at macro expansion time. Therefore, they must be constants. Example:

```
(defmacro 11 (&key a b c)
  (list 'list a b c))

(11 :b 5 :c (car d))
  ==> (list nil 5 (car d))
```

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

defmacro implements a few 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 ordinary arguments and should be indented accordingly. Example:

```
(defmacro with-open-file
          ((streamvar filename &rest options)
           &body body)
  ...)
```

&whole causes the variable that follows it to be bound to the entire macro call, just as the *form-arg* variable in macro would be. &whole exists to make defmacro able to do anything that macro can be used for, for the sake of Common Lisp, in which defmacro is the primitive and macro does not exist. &whole is also useful in macrolet.

&environment causes the variable that follows it to be bound to the *local macros environment* of the macro call being expanded. This is useful if the code for expanding this macro needs to invoke macroexpand on subforms of the macro call. Then, to achieve correct interaction with macrolet, this local macros environment should be passed to macroexpand as its second argument.

&list-of *pattern* requires that the corresponding position of the form being translated must 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 don't 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)
                          '((*print-base* 10.)
                            (*print-radix* nil))))
      `((lambda (,@vars) (print ,x))
        ,@vals))

    (print-let foo)  ==>
    ((lambda (*print-base* *print-radix*)
       (print foo))
     10 nil)

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

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

18.2.2 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 backquote and comma make it easier to type forms that create new list structure. They allow you to create a list from a template including constant and variable parts.

The backquote facility is used by giving a backquote character (`), followed by a list or vector. If the comma character does not appear within the text for the list or vector, the backquote acts just like a single quote: it creates a form which, when evaluated, produces the list or vector specified. For example,

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

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 of 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))
        `#(a ,b) => #(a 1)
```

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

The list or vector following a backquote can be thought of as a template for some new data structure. The parts of it that are preceded by commas are forms that fill in slots in the template; everything else is just constant structure that appears as written 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 doesn't use the regular functions **cons**, **list**, and so forth, but uses special ones instead so that the grinder can recognize a form which 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, which 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)))
```

To demonstrate finally 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. One way to define **do-forever** would be for it to expand

(do-forever *form1 form2 form3*)

into

(tagbody
    a *form1*
      form2
      form3
      (go a))

You could define the macro by

```
(defmacro do-forever (&body body)
  `(tagbody
      a ,@body
        (go a)))
```

(This definition has the disadvantage of interfering with use of the **go** tag **a** to go from the body of the **do-forever** to a tag defined outside of it. A more robust implementation would construct a new tag each time, using **gensym**.)

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 modified destructively; 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 any possible expansion of this would make a new list. One possible expansion is
```
        (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 which expand into forms which, when evaluated, define macros. In such macros it is often useful to use nested backquote constructs. For example, here is a very simple version of **defstruct** (see page 374) which does not allow any options and only the simplest slot descriptors. Its invocation looks like:

```
        (defstruct (name)
            item1 item2 ...)
```
We would like this form to expand into
```
        (progn
         (defmacro item1 (x) `(aref ,x 0))
         (defmacro item2 (x) `(aref ,x 1))
         (defmacro item3 (x) `(aref ,x 2))
         (defmacro item4 (x) `(aref ,x 3))
         ...)
```

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 . ,(nreverse ans)))
               (push `(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 comma 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.

18.3 Local Macro Definitions

defmacro or **macro** defines a macro whose name has global scope; it can be used in any function anywhere (subject to separation of name spaces by packages). You can also make local macro definitions which are in effect only in one piece of code. This is done with **macrolet**. Like lexical variable bindings made by **let** or the local function definitions made by **flet**, **macrolet** macro definitions are in effect only for code contained lexically within the body of the **macrolet** construct.

macrolet (*local-macros...*) *body...* *Special form*

Executes *body* and returns the values of the last form in it, with local macro definitions in effect according to *local-macros*.

Each element of *local-macros* looks like the cdr of a **defmacro** form:

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

and it is interpreted just the same way. However, *name* is only thus defined for expressions appearing within *body*.

```
(macrolet ((ifnot (x y . z) '(if (not ,x) ,y . ,z)))
  (ifnot foo (print bar) (print t)))

 ==> (if (not foo) (print bar) (print t))
```

It is permissible for *name* to have a global definition also, as a macro or as a function. The global definition is shadowed within *body*.

```
(macrolet ((car (x) '(cdr (assq ,x '((a . ferrari)
                                      (b . ford))))))
  ...(print (car symbol))...)
```

makes **car** have an unusual meaning for its explicit use, but due to lexical scoping it has no effect on what happens if **print** calls **car**.

macrolet can also hide other local definitions made by **macrolet**, **flet** or **labels** (page 45).

18.4 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 is 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 works just as if it had been defined by defun. If foo is compiled, however, the squaring is substituted into it and it produces the same code as

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

square's definition would be

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

(The internal formats of substs are explained in section 11.5.1, page 230.)

A similar square could be defined as a macro, but the simple way

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

has a bug: it causes the argument to be computed twice. The simplest correct definition as a macro is

```
        (defmacro square (x)
          (once-only (x)
            `(* ,x ,x)))
```

See page 338 for information on once-only.

In general, anything that is implemented as a subst can be re-implemented as a macro, just by changing the defsubst to a defmacro and putting in the appropriate backquote and commas, using once-only or creating temporary variables to make sure the arguments are computed once and in the proper order. The disadvantage of macros is that they are not functions, and so cannot be applied to arguments. Also, the effort required to guarantee the order of evaluation is a disadvantage. 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.

The *lambda-list* of a subst may contain &optional and &rest, but no other lambda-list keywords. If there is a rest argument, it is replaced in the body with an explicit call to list:

```
        (defsubst append-to-foo (&rest args)
          (setq foo (append args foo)))

        (append-to-foo x y z)
```

expands to

```
        (setq foo (append (list x y z) foo))
```

Rest arguments in substs are most useful with apply. Because of an optimization, if

```
        (defsubst xhack (&rest indices)
          (apply 'xfun xarg1 indices))
```

has been done then

```
        (xhack a (car b))
```

is equivalent to

```
        (xfun xarg1 a (car b))
```
If xfun is itself a subst, it is expanded in turn.

When a defsubst is compiled, its list structure definition is kept around so that calls can still be open-coded by the compiler. But non-open-coded calls to the function run at the speed of compiled code. The interpreted definition is kept in the compiled definition's debugging info alist (see page 242). Undeclared free variables used in a defsubst being compiled do not get any warning, because this is a common practice that works properly with nonspecial variables when calls are open coded.

If you are using a defsubst from outside the program to which it belongs, you might sometimes be better off if it is not open-coded. The decrease in speed might not be significant, and you would have the advantage that you would not need to recompile your program if the definition is changed. You can prevent open-coding by putting dont-optimize around the call to the defsubst.
```
        (dont-optimize (xhack a (car b)))
```
See page 306.

Straightforward substitution of the arguments could cause arguments to be computed more than once, or in the wrong order. For instance, the functions
```
        (defsubst reverse-cons (x y) (cons y x))
        (defsubst in-order (a b c) (and (< a b) (< b c)))
```
would 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. In fact, a more complicated form of substitution (implemented by si:sublis-eval-once, page 348) is used so that local variables are introduced as necessary to prevent such problems.

Note that 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.

18.5 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 the first thing you should do is 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, you will find the macro much easier to write.

In general any macro that can be written as a substitutable function (see page 329) 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); and there are some subtleties that can occur in macro definitions that need not be worried about in substitutable functions. A macro can be a substitutable function only if it has exactly the semantics of a function, rather than of a special form. The macros we will see in this section are not semantically like functions; they must be written as macros.

18.5.1 Name Conflicts

One of the most common errors in writing macros is best illustrated by example. Suppose we wanted to write dolist (see page 74) as a macro that expanded into a do (see page 70). 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 example's 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 programmer 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 doesn't 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's 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 programmer 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, in a package which only you will write code in. This will probably work, but it is inelegant since there is no guarantee that the user won't just happen to choose the same name. The way to avoid the name conflict reliably is to use an uninterned symbol as the variable in the generated code. The function gensym (see page 133) 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. (This is how the expression would print; this text would not read in properly because a new uninterned symbol would be created by each use of #:.) 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 error-handler. These symbols are in the si package; as a result, a name conflict is possible only in code which uses variables in the si package. This would not normally happen in user code, which resides in other packages.

18.5.2 Block-Name Conflicts

A related problem occurs when you write a macro that expands into a prog or do (or anything equivalent) behind the user's back (unlike dolist, which is documented to be like do). Consider the error-restart special form (see page 724). Suppose we wanted to implement it as a macro that expands into a do-forever, which becomes a prog. Then the following (contrived) Lisp program would not behave correctly:

```
(dolist (a list)
  (error-restart ((sys:abort error) "Return from FOO.")
    (cond ((> a 10)
           (return 5))
          ((> a 4)
           (ferror 'lose "You lose."))))))
```

The problem is that the return would return from the error-restart instead of the prog.

There are two possible ways to avoid this. The best is to make the expanded code use only explicit block's with obscure or gensymmed block names, and never a prog or do.

The other is to give any prog or do the name t. t as a prog name is special; it causes the prog to generate only a block named t, omitting the usual block named nil which is normally generated as well. Because only blocks named nil affect return, the problem is avoided.

When error-restart's expansion is supposed to return from the prog named t, it uses return-from t.

Macros like dolist specifically should expand into an ordinary do, because the user expects to be able to exit them with return.

18.5.3 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 (see page 34) as a macro. defconst must do two things, declare the variable to be special and set the variable to its initial value. (Here we implement a simplified defconst that does only these two things, and doesn't 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:

```
    (proclaim '(special a))
    (setq a (+ 4 b))
```
However, because of the way that macros work, they only expand into 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 (proclaim '(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. The proclaim form and the setq form are evaluated. The compiler recognizes progn specially and treats each argument of the progn form as if it had been encountered at top level. Here is the macro definition:

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

Here is another example of a form that wants to expand into several things. We 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 character that should invoke the function in this subsystem: Suppose that in this subsystem, commands are always functions of no arguments, and characters are used to index a vector called dispatch-table to find the function to use. A typical call to define-command would look like:

```
    (define-command move-to-top #\meta-<
      (do () ((at-the-top-p))
        (move-up-one)))
```

Expanding into:

```
    (progn (setf (aref dispatch-table #\meta-<)
                 'move-to-top)
           (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 specified character to invoke this command. 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 setf 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 (page 305) for more discussion of the differences between 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 attach the code for a command character to the character itself. Macro expansion allows the function definitions and the command dispatch table to be made from the same source code.

18.5.4 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 a modified 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 ((*print-base* ,base-form))
      . ,body))
```

A typical use of this macro might look like:

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

which would expand into

```
(let ((*print-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 standard Zetalisp constructs that are similar to this macro; see **with-open-file** (page 580) and **with-input-from-string** (page 473), for example. The really interesting thing, of course, is that you can define your own such constructs for your 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 construct 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 constructs 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 construct is "do this stuff *with* the following things true". Another convention is that any "parameters" to the construct should appear in a list that is the first subform of the construct, and that the rest of the elements 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 construct. 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** construct (page 473), 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 indent the elements of the body (see page 324).

The other thing to keep in mind is that control can leave the construct either by the last form's returning, or by a non-local exit (go, return or throw). You should write the definition in such a way that everything is cleaned up appropriately no matter how control exits. In our with-output-in-base example, there is no problem, because non-local exits undo lambda-bindings. However, in even slightly more complicated cases, an unwind-protect form (see page 82) 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 (see page 126) expands

```
(using-resource (window menu-resource) body...)
```

into

```
(let ((window nil))
  (unwind-protect
      (progn (setq window
                   (allocate-resource 'menu-resource))
             body...)
    (and window
         (deallocate-resource 'menu-resource window))))
```

This way the allocated resource item is deallocated whenever control leaves the using-resource special form.

18.5.5 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 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)))
```
Once again, the expansion would print this way, but this text would not read in as a valid expression due to the inevitable problems of **#:**.

In general, when you define a new construct which contains one or more argument forms, you must be careful that the expansion evaluates the argument forms the proper number of times and in the proper order. There's nothing fundamentally wrong with multiple or out-of-order evalation if that is really what you want and if it is what you document your special form to do. But if this happens unexpectedly, it can make invocations fail to work as they appear they should.

once-only is a macro that can be used to avoid multiple evaluation. It is most easily explained by example. You would write **test** using once-only 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 or 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 *var-list body...* *Macro*

var-list is a list of variables. The *body* is 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 *body* in this new binding environment and, when they have been evaluated, it undoes the bindings. The result of the evaluation of the last form in *body* is presumed to be a Lisp form, typically the expansion of a macro. If all of the variables have 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 non-trivial forms.

The effect is that the program produced by evaluating the once-only form is coded in such a way that, each of the forms which was the value of one of the variables in *var-list* is evaluated only once, unless the form is such as to have no side effects. At the same time, no unnecessary temporary variables appear in the generated code, but the body of the once-only is not cluttered up with extraneous code to decide whether temporary variables are needed.

18.5.6 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 the binding of var was unmade when the expander function was done. By the time the collect form gets expanded, the binding is long gone. 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.

One way to write these macros is using **macrolet**:

```
(defmacro with-collection (&body body)
  (let ((var (gensym)))
    `(macrolet ((collect (argument)
                  `(push ,argument ,',var)))
       (let ((,var nil))
         ,@body
         (nreverse ,var)))))
```

Here with-collection expands into code which defines collect specially to know about which variable to collect into. ,', causes var's value to be substituted when the outer backquote, the one around the macrolet, is executed. argument, however, is substituted in when the inner backquote is executed, which happens when collect is expanded.

This technique has the interesting consequence that collect is defined only within the body of a with-collection. It would simply not be recognized elsewhere; or it could have another definition, for some other purpose, globally. This has both advantages and disadvantages.

Another technique is to communicate through local declarations. The code generated by with-collection can contain a local-declare. The expansion of collect can examine the declararion with getdecl to decide what to do. Here is the code:

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

(defmacro collect (argument)
  (let ((var ,(getdecl nil 'collection-var)))
    (unless var
      (ferror nil "COLLECT not within a WITH-COLLECTION"))
    `(push ,argument var)))
```

Another way, used before getdecl existed, was with compiler-let (see page 316). compiler-let is identical to let as far as the interpreter is concerned, so the macro continues to work in the interpreter with this change. 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's the right way to write these macros in this fashion:

```
(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*))
```

18.5.7 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, 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 page 305 for more details. Doing this means that at compile time the definition of the function is interpreted, not compiled, and hence runs 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, and also all functions that the macros call. It can be separately compiled and loaded up before compiling the main part of the program, which uses the macros. The *system* facility (see chapter 28, page 660) helps keep these various files straight, compiling and loading things in the right order.

18.6 Aids for Debugging Macros

mexp &optional *form*

mexp goes into a loop in which it reads forms and sequentially expands them, printing out the result of each expansion (using the grinder (see page 528) to improve readability). When the form itself has been expanded until it is no longer a macro call, macroexpand-all is used to expand all its subforms, and the result is printed if it is different from what preceded. This allows you to see what your macros are expanding into, without actually evaluating the result of the expansion.

If the form you type is an atom, mexp returns. Usually one simply uses Abort to exit it.

If the form you type is a list that not a macro call, nothing is printed. You are prompted immediately for another form.

If the argument *form* is given, it is expanded and printed as usual, and then mexp returns immediately.

If you type
```
        (mexp)
```
followed by
```
        (rest (first x))
```
then mexp will print
```
        (cdr (first x))
```
and then
```
        (cdr (car x))
```
You would then type Abort to exit mexp.

18.7 Displacing Macro Calls

Every time the the evaluator sees a macro form, it must call the macro to expand the form. This is time consuming. To speed things up, the expansion of the macro is recorded automatically by modifying the form using rplaca and rplacd so that it no longer appears to need expansion. If the same form is evaluated again, it can be processed straight away. This is done using the function displace.

A consequence of the evaluator's policy of displacing macro calls is that if you change the definition of a macro, the new definition does not take effect in any form that has already been displaced. An existing form which calls the macro will use the new definition only if the form has never been evaluated.

displace *form expansion*

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

(si:displaced *form expansion*)

When a form whose car is si:displaced is evaluated, the evaluator simply extracts the expansion and evaluates it. *old-form-copy* is a newly consed pair whose car and cdr are the same as the original car and cdr of the form; thus, it records the macro call which was expanded. grindef uses this information to print the code as it was, rather than as it has been expanded.

displace returns *expansion*.

The precise format of a displaced macro call may be changed in the future to facilitate the implementation of automatic reexpansion if the called macro changes.

18.8 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 that may contain macros.

macroexpand-1 *form* &optional *local-macros-environment*

If *form* is a macro form, this expands it (once) and returns the expanded form. Otherwise it just returns *form*. The second value is t if *form* has been expanded.

local-macros-environment is a data structure which specifies the local macro definitions (made by macrolet) to be used for this expansion in addition to the global macro definitions (made by defmacro and recorded in function cells of symbols). When macroexpand-1 is called by the evaluator, this argument comes from the evaluator's own data structures set up by any macrolet forms which *form* was found within. When macroexpand-1 is called by the compiler, this argument comes from data structures kept by the compiler in its handling of macrolet.

Sometimes macro definitions call macroexpand-1; in that case, if *form* was a subform of the macro call, a &environment argument in the macro definition can be used to obtain a value to pass as *local-macros-environment*. See page 324. setf is one example of a macro that needs to use &environment since it expands some of its subforms in deciding what code to expand into. See setf, page 36.

If *local-macros-environment* is omitted or nil, only global macro definitions are used.

macroexpand-1 expands defsubst function forms as well as macro forms.

macroexpand *form* &optional *local-macros-environment*

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*. The second value is t if one or more expansions have take place. Everything said about *local-macros-environment* under macroexpand-1 applies here too.

macroexpand expands defsubst function forms as well as macro forms.

macroexpand-all *form* &optional *local-macros-environment*

Expands all macro calls in *form*, including those which are its subforms, and returns the result. By contrast, macroexpand would not expand the subforms. This function knows the syntax of all Lisp special forms, so the result is completely accurate. Note, however, that quoted list structure within *form* is not altered; there is no way to know whether you intend such list structure to be code or to be used in constructing code.

macroexpand-hook *Variable*

The value is a function which is used by **macroexpand-1** to invoke the expander function of a macro. It receives arguments just like **funcall**: the expander function, and the arguments for it.

In fact, the default value of this variable *is* **funcall**. The variable exists so that the user can set it to some other function, which performs the **funcall** and possibly other associated record-keeping.

macroexpand-hook is not used when a macro is expanded by the interpreter.

18.9 Definitions of Macros

The definition of a macro is a list whose car is the symbol **macro**. The cdr of the list is the macro's *expander function*. This expander function contains the code written in the **defmacro** or other construct which was used to define the macro. It may be a **lambda** expression, or it may be a compiled function object (FEF). Expanding the macro is done by invoking the expander function.

When an expander function is called, it receives two arguments: the macro call to be expanded, and the local macros environment. If the expansion is being done by **macroexpand-1** then the local macros environment passed is the one that was given to **macroexpand-1**. In a macro defined with **defmacro**, the local macros environment can be accessed by writing an **&environment** parameter (see page 324).

Expander functions used to be given only one argument. For compatibility, it is useful to define expander functions so that the second argument is optional; **defmacro** does so. In addition, old macro definitions still work, because **macroexpand-1** actually checks the number of arguments which the expander function is ready to receive, and passes only one argument if the expander function expects only one. This is done using **call** (see page 48).

macro-function *function-spec*

If *function-spec* is defined as a macro, then this returns its expander-function: the function which should be called, with a macro call as its sole argument, to produce the macro expansion. For certain special forms, **macro-function** returns the "alternate macro definition" (see below). Otherwise, **macro-function** returns **nil**.

Since a definition as a macro is really a list of the form (**macro** . *expander-function*), you can get the expander function using (**cdr** (**fdefinition** *function-spec*)). But it is cleaner to use **macro-function**.

```
        (setf (macro-function function-spec) expander)
```
is permitted, and is equivalent to
```
        (fdefine function-spec (cons 'macro expander))
```

Certain constructs which Common Lisp specifies as macros are actually implemented as special forms (**cond**, for example). These special forms have "alternate macro definitions" which are the definitions they might have if they were implemented as macros. This is so that the caller of **macro-function**, if it is a portable Common Lisp program, need not know about any special forms except the standard Common Lisp ones in order to make deductions about all valid Common Lisp programs. It can instead regard as a macro any symbol on which macro-function returns a non-nil value, and treat that value as the macro expander function.

The alternate macro definition of a symbol such as cond is not actually its function definition. It exists only for macro-function to return. The existence of alternate macro definitions means that macro-function is not useful for testing whether a symbol really is defined as a macro.

18.10 Extending setf and locf

This section would logically belong within section 3.2, page 35, but it is too advanced to go there. It is placed in this chapter because it deals with concepts related to macro-expansion.

There are three ways to tell the system how to setf a function: simple defsetf when it is trivial, general defsetf which handles most other cases, and define-setf-method which provides the utmost generality.

defsetf *Macro*

The simple way to use defsetf is useful when there is a setting function which does all the work of storing a value into the appropriate place and has the proper calling conventions.

```
(defsetf function setting-function)
```

says that the way to store into (*function args...*) is to do (*setting-function args... new-value*). For example,

```
(defsetf car sys:setcar)
```

is the way setf of car is defined. Its meaning is that (setf (car *x*) *y*) should expand into (sys:setcar *x y*). (setcar is like rplaca except that setcar returns its second argument).

The more general form of defsetf is used when there is no setting function with exactly the right calling sequence. Thus,

```
(defsetf function (function-args...) (value-arg) body...)
```

tells setf how to store into (*function args...*) by providing something like a macro defininition to expand into code to do the storing. *body* computes the code; the last form in *body* returns a suitable expression. *function-args* should be a lambda list, which can have optional and rest args. *body* can substitute the values of the variables in this lambda list, to refer to the arguments in the form being setf'ed. Likewise, it can substitute in *value-arg* to refer to the value to be stored.

In fact, the *function-args* and *value-arg* are not actually the subforms of the form being setfd and the value to be stored; they are gensyms. After the *body* returns, the corresponding expressions may be substituted for the gensyms, or the gensyms may remain as local variables with a suitable let provided to bind them. This is how setf ensures a correct order of evaluation.

Example:

```
(defsetf car (list) (value) `(sys:setcar ,list ,value))
```

is how one could define the setf'ing of car using the general form of defsetf. The simple form of **defsetf** can be regarded as an abbreviation for something like this.

Since **setf** automatically expands macros, if you define a macro whose expansion is usable in **setf** then the macro is usable there also. Sometimes this is not desirable. For example, the accessor subst for a slot in a **defstruct** structure probably expands into **aref**, but if the slot is declared :read-only this should not be allowed. It is prevented by means of a **defsetf** like this:

```
        (defsetf accessor-function)
```

This means that **setf** is explicitly prohibited on that function.

define-setf-method *function* (*function-args*...) (*value-arg*) *body*... *Macro*

Defines how to do **setf** on *place*'s starting with *function*, with more power and generality than **defsetf** provides, but more complexity of use.

The **define-setf-method** form receives its arguments almost like an analogous **defsetf**. However, the values it receives are the actual subforms, and the actual form for the value, rather than gensyms which stand for them. The *function-args* are the actual subforms of the place to be **setf**'ed, and the full power of **defmacro** arglists can be used to match against it. *value-arg* is the actual form used as the second argument to **setf**.

body is once again evaluated, but it does not return an expression to do the storing. Instead, it returns five values which contain sufficient information to enable anyone to examine and modify the contents of the place. This information tells the caller which subforms of the place need to be evaluated, and how to use them to examine or set the value of the place. (Generally the function-args arglist is arranged to make each arg get one subform.) A temporary variable must be found or made (usually with **gensym**) for each of them. Another temporary variable should be made to correspond to the value to be stored.

Then the five values to be returned are:

0 A list of the temporary variables for the subforms of the place.

1 A list of the subforms that they correspond to.

2 A list of the temporary variables for the values to be stored. Currently there can only be one value to be stored, so there is only one variable in this list, always.

3 A form to do the storing. This form refers to some or all of the temporary variables listed in value 1.

4 A form to get the value of the place. **setf** does not need to do this, but **push** and **incf** do. This too should refer only to the temporary variables. No expression of contained it it should be a subexpression of the place being stored in.

This information is everything that the macro (**setf** or something more complicated) needs to know to decide what to do.

Example:

```
      (define-setf-method car (function-spec)
        (let ((tempvars (list (gensym)))
              (tempargs (list (list-form)))
              (storevar (gensym)))
          (values tempvars tempargs (list storevar)
                  `(sys:setcar ,(first tempvars) ,storevar)
                  `(car ,(first tempvars)))))
```

is how one could define the setf'ing of car using define-setf-method. This definition is equivalent to the other two definitions using the simpler techniques.

get-setf-method *form*

Invokes the **setf** method for form (which must be a list) and returns the five values produced by the body of the **define-setf-method** for the symbol which is the car of form. The meanings of these five values are given immediately above. If the way to **setf** that symbol was defined with **defsetf** you still get five values, which you can interpret in the same ways; thus, **defsetf** is effectively an abbreviation for a suitable **define-setf-method**.

There are two ways to use **get-setf-method**. One is in a macro which, like **setf** or **incf** or **push**, wants to store into a place. The other is in a **define-setf-method** for something like **ldb**, which is **setf** by setting one of its arguments. You would append your new tempvars and tempargs to the ones you got from **get-setf-method** to get the combined lists which you return. The forms returned by the **get-setf-method** you would stick into the forms you return.

An example of a macro which uses **get-setf-method** is **pushnew**. (The real **pushnew** is a little hairier than this, to handle the *test*, *test-not* and *key* arguments).

```
      (defmacro pushnew (value place)
        (multiple-value-bind
            (tempvars tempargs storevars storeform refform)
            (get-setf-method place)
          (si:sublis-eval-once
            (cons `(-val- . ,value) (pairlis tempvars tempargs))
            `(if (memq -val- ,refform)
                 ,refform
               ,(sublis (list (cons (car storevars)
                                    `(cons -val- ,refform)))
                        storeform))
            t t)))
```

An example of a **define-setf-method** that uses **get-setf-method** is that for **ldb**:

```
(define-setf-method ldb (bytespec int)
  (multiple-value-bind
        (temps vals stores store-form access-form)
      (get-setf-method int)
    (let ((btemp (gensym))
          (store (gensym))
          (itemp (first stores)))
      (values (cons btemp temps)
              (cons bytespec vals)
              (list store)
              `(progn
                 ,(sublis
                    (list (cons itemp
                                `(dpb ,store ,btemp
                                      ,access-form)))
                    store-form)
                ,store)
              `(ldb ,btemp ,access-form)))))
```

What this says is that the way to **setf** (**ldb** *byte* (**foo**)) is computed based on the way to **setf** (**foo**).

si:sublis-eval-once *alist form* &optional *reuse-tempvars sequential-flag*

Replaces temporary variables in *form* with corresponding values according to *alist*, but generates local variables when necessary to make sure that the corresponding values are evaluated exactly once and in same order that they appear in *alist*. (This complication is skipped when the values are constant). *alist* should be a list of elements (*tempvar* . *value*). The result is a form equivalent to

```
'(let ,(mapcar #'(lambda (elt) (list (car elt) (cdr elt)))
               alist)
   ,form)
```

but it usually contains fewer temporary variables and executes faster.

If *reuse-tempvars* is non-nil, the temporary variables which appear as the cars of the elements of *alist* are allowed to appear in the resulting form. Otherwise, none of them appears in the resulting form, and if any local variables turn out to be needed, they are made afresh with **gensym**. *reuse-tempvars* should be used only when it is guaranteed that none of the temporary variables in *alist* is referred to by any of the values to be substituted; as, when the temporary variables have been freshly made with **gensym**.

If *sequential-flag* is non-nil, then the value substituted for a temporary variable is allowed to refer to the temporary variables preceding it in alist. **setf** and similar macros should all use this option.

define-modify-macro *macro-name* (*lambda-list...*) *combiner-function [doc-string]*

Is a quick way to define **setf**'ing macros which resemble **incf**. For example, here is how **incf** is defined:

```
(define-modify-macro incf (&optional (delta 1)) +
   "Increment PLACE's value by DELTA.")
```

lambda-list describes any arguments the macro accepts, but not first argument, which is always the place to be examined and modified. The old value of this place, and any additional arguments such as **delta** in the case of **incf**, are combined using the *combiner-function* (in this case, +) to get the new value which is stored back in the place.

deflocf *Macro*

Defines how to perform **locf** on a generalized variable. There are two forms of usage, analogous to those of **defsetf**.

(deflocf *function locating-function*)

says that the way to get the location of (*function args...*) is to do (*locating-function args...*). For example,

```
(deflocf car sys:car-location)
```

could be used to define **locf** on **car** forms. is the way **setf** of **car** is defined. Its meaning is that **(locf (car** *x***))** should expand into **(sys:car-location** *x***)**.

The more general form of **deflocf** is used when there is no locating function with exactly the right calling sequence. Thus,

(deflocf *function* (*function-args...*) *body...*)

tells **locf** how to locate (*function args...*) by providing something like a macro defininition to expand into code to do the locating. *body* computes the code; the last form in *body* returns a suitable expression. *function-args* should be a lambda list, which can have optional and rest args. *body* can substitute the values of the variables in this lambda list, to refer to the arguments in the form being **locf**'ed.

Example:

```
(deflocf car (list) `(sys:car-location ,list))
```

is how one could define the **locf**'ing of **car** using the general form of **deflocf**. The simple form of **deflocf** can be regarded as an abbreviation for something like this.

(deflocf *function*)

says that **locf** should not be allowed on forms starting with *function*. This is useful only when *function* is defined as a macro or subst, for then **locf**'s normal action is to expand the macro call and try again. In other cases there is no way to **locf** a function unless you define one, so you can simply refrain from defining any way.

19. The LOOP Iteration Macro

19.1 Introduction

loop is a Lisp macro that provides a programmable iteration facility. The same loop module operates compatibly in Zetalisp, Maclisp (PDP-10 and Multics), and NIL, and a moderately compatible package is under development for the MDL programming environment. loop was inspired by the FOR facility of CLISP in InterLisp; however, it is not compatible and differs in several details.

The general approach is that a form introduced by the word loop generates a single program loop, into which a large variety of features can be incorporated. The loop consists of some initialization (*prologue*) code, a body that may be executed several times, and some exit (*epilogue*) code. Variables may be declared local to the loop. The special features of loop are concerned with loop variables, deciding when to end the iteration, putting user-written code into the loop, returning a value from the construct, and iterating a variable through various real or virtual sets of values.

The loop form consists of a series of clauses, each introduced by a "keyword" symbol. These symbols are keywords from loop's point of view; they are not keywords in the usual sense (symbols in the keyword package). loop ignores the package when it compares a symbol against the known keywords.

Forms appearing in or implied by the clauses of a loop form are classed as those to be executed as initialization code, body code, and/or exit code; within each part of the template filled in by loop, they are executed strictly in the order implied by the original composition. Thus, just as in ordinary Lisp code, side-effects may be used, and one piece of code may depend on following another for its proper operation. This is the principal philosophic difference from InterLisp's FOR facility.

Note that loop forms are intended to look like stylized English rather than Lisp code. There is a notably low density of parentheses, and many of the keywords are accepted in several synonymous forms to allow writing of more euphonious and grammatical English. Some find this notation verbose and distasteful, while others find it flexible and convenient. The former are invited to stick to do.

Here are some examples to illustrate the use of loop.

```
(defun print-elements-of-list (list-of-elements)
    (loop for element in list-of-elements
          do (print element)))
```
prints each element in its argument, which should be a list. It returns nil.

```
(bar (defun gather-alist-entries (list-of-pairs)
        (loop for pair in list-of-pairs
              collect (car pair)))
```
takes an association list and returns a list of the keys; that is, (gather-alist-entries '((foo 1 2) (bar 259) (baz))) returns (foo bar baz).

```
(defun extract-interesting-numbers (start-value end-value)
   (loop for number from start-value to end-value
         when (interesting-p number) collect number))
```
takes two arguments, which should be integers, and returns a list of all the numbers in that range (inclusive) which satisfy the predicate interesting-p.

```
(defun find-maximum-element (an-array)
   (loop for i from 0 below (array-dimension-n 1 an-array)
         maximize (aref an-array i)))
```
returns the maximum of the elements of its argument, a one-dimensional array.

```
(defun my-remove (object list)
   (loop for element in list
         unless (equal object element) collect element))
```
is like the standard function remove, except that it copies the entire list.

```
(defun find-frob (list)
   (loop for element in list
         when (frobp element) return element
         finally (ferror nil "No frob found in the list ~S"
                             list)))
```
returns the first element of its list argument which satisfies the predicate frobp. If none is found, an error is signaled.

Common Lisp defines loop as equivalent to do-forever: it is used with a body consisting only of forms to be evaluated until a nonlocal exit happens. This is incompatible with the traditional loop macro which this chapter is about. However, it is possible to tell which meaning of loop the programmer intended: in the traditional loop macro, it must be a symbol, while in the Common Lisp loop it is useless to use a symbol there. Therefore, if the first argument of a loop form is not a symbol, it is treated as a Common Lisp loop.

19.2 Clauses

Internally, loop constructs a prog which includes variable bindings, pre-iteration (initialization) code, post-iteration (exit) code, the body of the iteration, and stepping of variables of iteration to their next values (which happens on every iteration after the body is executed).

A *clause* consists of a keyword symbol and any Lisp forms and keywords that it deals with. For example,
```
    (loop for x in l do (print x)),
```
contains two clauses, `for x in l` and `do (print x)`. Certain of the parts of the clause will be described as being *expressions*, e.g. (print x) in the above. An expression can be a single Lisp

form, or a series of forms implicitly collected with progn. An expression is terminated by the next following atom, which is taken to be a keyword. This syntax allows only the first form in an expression to be atomic, but makes misspelled keywords more easily detectable.

loop uses print-name equality to compare keywords so that loop forms may be written without package prefixes; in Lisp implementations that do not have packages, eq is used for comparison.

Bindings and iteration variable steppings may be performed either sequentially or in parallel. This affects how the stepping of one iteration variable may depend on the value of another. The syntax for distinguishing the two will be described with the corresponding clauses. When a set of variables are to be bound in parallel, all of the initial values are computed and then all the bindings are established. Subsequent bindings will be performed inside of that binding environment. When the same variables are stepped, all the new values are computed and then the variables are set.

19.2.1 Iteration-Driving Clauses

These clauses all create a *variable of iteration*, which is bound locally to the loop and takes on a new value on each successive iteration. Note that if more than one iteration-driving clause is used in the same loop, several variables are created that all step together through their values; when any of the iterations terminates, the entire loop terminates. Nested iterations are not generated; for those, you need a second loop form in the body of the loop. In order not to produce strange interactions, iteration-driving clauses are required to precede any clauses that produce body code: that is, all except those that produce prologue or epilogue code (initially and finally), bindings (with), the named clause, and the iteration termination clauses (while and until).

Clauses which drive the iteration may be arranged to perform their testing and stepping either in series or in parallel. They are by default grouped in series, which allows the stepping computation of one clause to use the just-computed values of the iteration variables of previous clauses. They may be made to step in parallel, as is the case with the do special form, by "joining" the iteration clauses with the keyword and. The form this typically takes is something like

```
    (loop ... for x = (f) and for y = init then (g x) ...)
```

which sets x to (f) on every iteration, and binds y to the value of *init* for the first iteration, and on every iteration thereafter sets it to (g x), where x still has the value from the *previous* iteration. Thus, if the calls to f and g are not order-dependent, this would be best written as

```
    (loop ... for y = init then (g x) for x = (f) ...)
```

because, as a general rule, parallel stepping has more overhead than sequential stepping. Similarly, the example

```
    (loop for sublist on some-list
          and for previous = 'undefined then sublist
          ...)
```

which is equivalent to the do construct

```
    (do ((sublist some-list (cdr sublist))
         (previous 'undefined sublist))
        ((null sublist) ...)
      ...)
```
in terms of stepping, would be better written as
```
    (loop for previous = 'undefined then sublist
          for sublist on some-list
          ...)
```

When iteration-driving clauses are joined with **and**, if the token following the **and** is not a keyword that introduces an iteration driving clause, it is assumed to be the same as the keyword that introduced the most recent clause; thus, the above example showing parallel stepping could have been written as
```
    (loop for sublist on some-list
          and previous = 'undefined then sublist
          ...)
```

The order of evaluation in iteration-driving clauses is as follows: those expressions that are only evaluated once are evaluated in order at the beginning of the form, during the variable-binding phase, while those expressions that are evaluated each time around the loop are evaluated in order in the body.

One common and simple iteration-driving clause is **repeat**:

repeat *expression*
: Evaluates *expression* (during the variable binding phase), and causes the **loop** to iterate that many times. *expression* is expected to evaluate to an integer. If *expression* evaluates to a zero or negative result, the body code will not be executed.

All remaining iteration-driving clauses are subdispatches of the keyword **for**, which is synonomous with **as**. In all of them a *variable of iteration* is specified. Note that, in general, if an iteration-driving clause implicitly supplies an endtest, the value of this iteration variable is undefined as the loop is exited (i.e., when the epilogue code is run). This is discussed in more detail in section 19.5.

Here are all of the varieties of **for** clauses. Optional parts are enclosed in curly brackets.

for *var* **in** *expr1* {**by** *expr2*}
: Iterates over each of the elements in the list *expr1*. If the **by** subclause is present, *expr2* is evaluated once on entry to the loop to supply the function to be used to fetch successive sublists, instead of **cdr**.

for *var* **on** *expr1* {**by** *expr2*}
: Like the previous **for** format, except that *var* is set to successive sublists of the list instead of successive elements. Note that **loop** uses a **null** rather than an **atom** test to implement both this and the preceding clause.

for *var* = *expr*
: On each iteration, *expr* is evaluated and *var* is set to the result.

for *var* = *expr1* **then** *expr2*
: *var* is bound to *expr1* when the loop is entered, and set to *expr2* (re-evaluated) at all

but the first iteration. Since *expr1* is evaluated during the binding phase, it cannot reference other iteration variables set before it: for that, use the following:

for *var* first *expr1* then *expr2*

Sets *var* to *expr1* on the first iteration, and to *expr2* (re-evaluated) on each succeeding iteration. The evaluation of both expressions is performed *inside* of the loop binding environment, before the loop body. This allows the first value of *var* to come from the first value of some other iteration variable, allowing such constructs as

```
        (loop for term in poly
              for ans first (car term)
                      then (gcd ans (car term))
              finally (return ans))
```

for *var* from *expr1* {to *expr2*} {by *expr3*}

This performs numeric iteration. *var* is initialized to *expr1*, and on each succeeding iteration is incremented by *expr3* (default 1). If the to phrase is given, the iteration terminates when *var* becomes greater than *expr2*. Each of the expressions is evaluated only once, and the to and by phrases may be written in either order. Alternative keywords may be used in place of to; this choice controls the direction of stepping and the step at which the loop terminates. downto instead of to says that *var* is decremented by the step value, and the endtest is adjusted accordingly. If below is used instead of to, or above instead of downto, the iteration terminates before *expr2* is reached, rather than after. Note that the to variant appropriate for the direction of stepping must be used for the endtest to be formed correctly; i.e. the code will not work if *expr3* is negative or zero. If no limit-specifying clause is given, then the direction of the stepping may be specified as decreasing by using downfrom instead of from. upfrom may also be used instead of from; it forces the stepping direction to be increasing.

for *var* being *expr* and its *path* ...
for *var* being {each|the} *path* ...

Provides a user-definable iteration facility. *path* names the manner in which the iteration is to be performed. The ellipsis indicates where various path dependent preposition/expression pairs may appear. See the section on Iteration Paths (page 363) for complete documentation.

19.2.2 Bindings

The with keyword may be used to establish initial bindings, that is, variables that are local to the loop but are only set once, rather than on each iteration. The with clause looks like:

with *var1* { = *expr1*}
    {and *var2* { = *expr2*}}...

If no *expr* is given, the variable is initialized to nil.

with bindings linked by and are performed in parallel; those not linked are performed sequentially. That is,

```
    (loop with a = (foo) and b = (bar) and c
          ...)
```

binds the variables like

```
(let ((a (foo)) (b (bar)) c) ...)
```
whereas
```
(loop with a = (foo) with b = (bar a) with c ...)
```
binds the variables like
```
(let ((a (foo)))
  (let ((b (bar)))
    (let (c) ...)))
```
All *expr*'s in **with** clauses are evaluated in the order they are written, in lambda expressions surrounding the generated **prog**. The **loop** expression
```
(loop with a = xa and b = xb
      with c = xc
      for d = xd then (f d)
        and e = xe then (g e d)
      for p in xp
      with q = xq
      ...)
```
produces the following binding contour, where **t1** is a **loop**-generated temporary:
```
(let ((a xa) (b xb))
  (let ((c xc))
    (let ((d xd) (e xe))
      (let ((p nil) (t1 xp))
        (let ((q xq))
          ...)))))
```
Because all expressions in **with** clauses are evaluated during the variable binding phase, they are best placed near the front of the **loop** form for stylistic reasons.

For binding more than one variable with no particular initialization, one may use the construct

with *variable-list* {**and** ...}

as in
```
with (i j k t1 t2) ...
```
These are cases of *destructuring* which **loop** handles specially; destructuring and data type keywords are discussed in section 19.4.

19.2.3 Entrance and Exit

initially *expression*

Puts *expression* into the *prologue* of the iteration. It will be evaluated before any other initialization code except for initial bindings. For the sake of good style, the **initially** clause should therefore be placed after any **with** clauses but before the main body of the loop.

finally *expression*

Puts *expression* into the *epilogue* of the loop, which is evaluated when the iteration terminates (other than by an explicit **return**). For stylistic reasons, then, this clause should appear last in the loop body. Note that certain clauses may generate code which terminates the iteration without running the epilogue code; this behavior is noted with those clauses. Most notable of these are those described in the section

19.2.7, Aggregated Boolean Tests. This clause may be used to cause the loop to return values in a non-standard way:

```
(loop for n in l
      sum n into the-sum
      count t into the-count
      finally (return (quotient the-sum the-count)))
```

19.2.4 Side Effects

do *expression*
doing *expression*

expression is evaluated each time through the loop, as shown in the **print-elements-of-list** example on page 350.

19.2.5 Values

The following clauses accumulate a return value for the iteration in some manner. The general form is

type-of-collection expr {into *var*}

where *type-of-collection* is a **loop** keyword, and *expr* is the thing being accumulated somehow. If no **into** is specified, then the accumulation will be returned when the **loop** terminates. If there is an **into**, then when the epilogue of the **loop** is reached, *var* (a variable automatically bound locally in the loop) will have been set to the accumulated result and may be used by the epilogue code. In this way, a user may accumulate and somehow pass back multiple values from a single **loop**, or use them during the loop. It is safe to reference these variables during the loop, but they should not be modified until the epilogue code of the loop is reached. For example,

```
(loop for x in list
      collect (foo x) into foo-list
      collect (bar x) into bar-list
      collect (baz x) into baz-list
    finally (return (list foo-list bar-list baz-list)))
```

has the same effect as

```
(do ((#:g0001 list (cdr #:g0001))
     (x) (foo-list) (bar-list) (baz-list))
    ((null #:g0001)
     (list (nreverse foo-list)
           (nreverse bar-list)
           (nreverse baz-list)))
   (setq x (car #:g0001))
   (setq foo-list (cons (foo x) foo-list))
   (setq bar-list (cons (bar x) bar-list))
   (setq baz-list (cons (baz x) baz-list)))
```

except that **loop** arranges to form the lists in the correct order, obviating the **nreverses** at the end, and allowing the lists to be examined during the computation. (This is how the expression would print; this text would not read in properly because a new uninterned symbol would be created by each use of **#:**.)

collect *expr* {into *var*}
collecting ...

Causes the values of *expr* on each iteration to be collected into a list.

nconc *expr* {into *var*}
nconcing ...
append ...
appending ...

Like collect, but the results are nconc'ed or append'ed together as appropriate.

```
(loop for i from 1 to 3
      nconc (list i (* i i)))
   => (1 1 2 4 3 9)
```

count *expr* {into *var*}
counting ...

If *expr* evaluates non-nil, a counter is incremented.

sum *expr* {into *var*}
summing ...

Evaluates *expr* on each iteration and accumulates the sum of all the values.

maximize *expr* {into *var*}
minimize ...

Computes the maximum (or minimum) of *expr* over all iterations.
Note that if the loop iterates zero times, or if conditionalization prevents the code of this clause from being executed, the result will be meaningless.

Not only may there be multiple accumulations in a loop, but a single accumulation may come from multiple places *within the same* loop *form*. Obviously, the types of the collection must be compatible. collect, nconc, and append may all be mixed, as may sum and count, and maximize and minimize. For example,

```
(loop for x in '(a b c) for y in '((1 2) (3 4) (5 6))
      collect x
      append y)
  => (a 1 2 b 3 4 c 5 6)
```

The following computes the average of the entries in the list *list-of-frobs*:

```
(loop for x in list-of-frobs
      count t into count-var
      sum x into sum-var
     finally (return (cli:// sum-var count-var)))
```

19.2.6 Endtests

The following clauses may be used to provide additional control over when the iteration gets terminated, possibly causing exit code (due to **finally**) to be performed and possibly returning a value (e.g., from **collect**).

while *expr*
: If *expr* evaluates to **nil**, the loop is exited, performing exit code (if any) and returning any accumulated value. The test is placed in the body of the loop where it is written. It may appear between sequential **for** clauses.

until *expr*
: Identical to **while (not** *expr***)**.

This may be needed, for example, to step through a strange data structure, as in

```
(loop until (top-of-concept-tree? concept)
      for concept = expr then (superior-concept concept)
      ...)
```

Note that the placement of the **until** clause before the **for** clause is valid in this case because of the definition of this particular variant of **for**, which *binds* **concept** to its first value rather than setting it from inside the **loop**.

The following may also be of use in terminating the iteration:

loop-finish *Macro*

(loop-finish) causes the iteration to terminate "normally", like implicit termination by an iteration-driving clause, or by the use of **while** or **until**—the epilogue code (if any) will be run, and any implicitly collected result will be returned as the value of the **loop**. For example,

```
(loop for x in '(1 2 3 4 5 6)
      collect x
      do (cond ((= x 4) (loop-finish))))
 => (1 2 3 4)
```

This particular example would be better written as **until (= x 4)** in place of the **do** clause.

19.2.7 Aggregated Boolean Tests

All of these clauses perform some test and may immediately terminate the iteration depending on the result of that test.

always *expr*
: Causes the loop to return **t** if *expr* always evaluates non-null. If *expr* evaluates to **nil** the loop immediately returns **nil**, without running the epilogue code (if any, as specified with the **finally** clause); otherwise, **t** will be returned when the loop finishes, after the epilogue code has been run.

never *expr*
: Causes the loop to return **t** if *expr* never evaluates non-null. This is equivalent to **always (not** *expr***)**.

thereis *expr*
: If *expr* evaluates non-nil, then the iteration is terminated, and that value is returned without running the epilogue code.

19.2.8 Conditionalization

These clauses may be used to "conditionalize" the following clause. They may precede any of the side-effecting or value-producing clauses, such as do, collect, always, or return.

when *expr*
if *expr*
: If *expr* evaluates to nil, the following clause will be skipped, otherwise not.

unless *expr*
: This is equivalent to when (not *expr*)).

Multiple conditionalization clauses may appear in sequence. If one test fails, then any following tests in the immediate sequence, as well as the clause being conditionalized, are skipped.

Multiple clauses may be conditionalized under the same test by joining them with and, as in

```
(loop for i from a to b
      when (zerop (remainder i 3))
        collect i and do (print i))
```

which returns a list of all multiples of 3 from a to b (inclusive) and prints them as they are being collected.

If-then-else conditionals may be written using the else keyword, as in

```
(loop for i from a to b
      when (oddp i)
        collect i into odd-numbers
      else collect i into even-numbers)
```

Multiple clauses may appear in an else-phrase, using and to join them in the same way as above.

Conditionals may be nested. For example,

```
(loop for i from a to b
      when (zerop (remainder i 3))
        do (print i)
        and when (zerop (remainder i 2))
              collect i)
```

returns a list of all multiples of 6 from a to b, and prints all multiples of 3 from a to b.

When else is used with nested conditionals, the "dangling else" ambiguity is resolved by matching the else with the innermost when not already matched with an else. Here is a complicated example.

```
(loop for x in l
      when (atom x)
        when (memq x *distinguished-symbols*)
          do (process1 x)
        else do (process2 x)
      else when (memq (car x) *special-prefixes*)
             collect (process3 (car x) (cdr x))
             and do (memoize x)
           else do (process4 x))
```

Useful with the conditionalization clauses is the **return** clause, which causes an explicit return of its argument as the value of the iteration, bypassing any epilogue code. That is,

   when *expr1* **return** *expr2*

is equivalent to

   when *expr1* **do (return** *expr2***)**

Conditionalization of one of the "aggregated boolean value" clauses simply causes the test that would cause the iteration to terminate early not to be performed unless the condition succeeds. For example,

```
(loop for x in l
      when (significant-p x)
        do (print x) (princ "is significant.")
        and thereis (extra-special-significant-p x))
```

does not make the **extra-special-significant-p** check unless the **significant-p** check succeeds.

The format of a conditionalized clause is typically something like

   when *expr1 keyword expr2*

If *expr2* is the keyword **it**, then a variable is generated to hold the value of *expr1*, and that variable gets substituted for *expr2*. Thus, the composition

   when *expr* **return it**

is equivalent to the clause

   thereis *expr*

and one may collect all non-null values in an iteration by saying

   when *expression* **collect it**

If multiple clauses are joined with **and**, the **it** keyword may only be used in the first. If multiple **when**s, **unless**es, and/or **if**s occur in sequence, the value substituted for **it** will be that of the last test performed. The **it** keyword is not recognized in an **else**-phrase.

19.2.9 Miscellaneous Other Clauses

named *name*

Defines a **block** named *name* around the code for the **loop**, so that one may use **return-from** to return explicitly out of this particular **loop**. This is obsolete now that **block** exists; it is cleaner to write (**block** *name* ...) around the **loop**.

Note that every **loop** generates a **block** named **nil**, so the function **return** can always be used to exit the innermost **loop** (assuming no other construct generating a **block nil** intervenes).

return *expression*

Immediately returns the value of *expression* as the value of the loop, without running the epilogue code. This is most useful with some sort of conditionalization, as discussed in the previous section. Unlike most of the other clauses, **return** is not considered to "generate body code", so it is allowed to occur between iteration clauses, as in

```
(loop for entry in list
      when (not (numberp entry))
        return (ferror ...)
      as frob = (times entry 2)
      ...)
```

Although **ferror** is called only for effect, **return** is used so that it can be called from that point in the **loop**.

If one instead desires the loop to have some return value when it finishes normally, one may place a call to the **return** function in the epilogue (with the **finally** clause, page 355).

19.3 Loop Synonyms

define-loop-macro *keyword* *Macro*

May be used to make *keyword*, a **loop** keyword (such as **for**), into a Lisp macro that may introduce a **loop** form. For example, after evaluating

```
(define-loop-macro for),
```

one may now write an iteration as

```
(for i from 1 below n do ...)
```

This facility exists primarily for diehard users of a predecessor of **loop**. Its unconstrained use is not recommended, as it tends to decrease the transportability of the code and needlessly uses up a function name.

19.4 Destructuring

Destructuring provides one with the ability to "simultaneously" assign or bind multiple variables to components of some data structure. Typically this is used with list structure. For example,

```
(loop with (foo . bar) = '(a b c) ...)
```

has the effect of binding **foo** to **a** and **bar** to **(b c)**.

loop's destructuring support is intended to parallel and perhaps augment that provided by the host Lisp implementation, with a goal of minimally providing destructuring over list structure patterns. Thus, in Lisp implementations with no system destructuring support at all, one may still use list-structure patterns as **loop** iteration variables and in **with** bindings.

One may specify the data types of the components of a pattern by using a corresponding pattern of the data type keywords in place of a single data type keyword. This syntax remains unambiguous because wherever a data type keyword is possible, a **loop** keyword is the only other

possibility. Thus, if one wants to do
```
        (loop for x in l
              as i = (car x)
              and j = (cadr x)
              and k = (cddr x)
              ...)
```
and no reference to x is needed, one may instead write
```
        (loop for (i j . k) in l ...)
```
To allow some abbreviation of the data type pattern, an atomic component of the data type pattern is considered to state that all components of the corresponding part of the variable pattern are of that type. That is, the previous form could be written as
```
        (loop for (i j . k) in l ...)

        (defun map-over-properties (fn symbol)
           (loop for (propname propval) on (plist symbol) by 'cddr
                 do (funcall fn symbol propname propval)))
```
maps *fn* over the properties on *symbol*, giving it arguments of the symbol, the property name, and the value of that property.

19.5 The Iteration Framework

This section describes the way **loop** constructs iterations. It is necessary if you will be writing your own iteration paths, and may be useful in clarifying what **loop** does with its input.

loop considers the act of *stepping* to have four possible parts. Each iteration-driving clause has some or all of these four parts, which are executed in this order:

pre-step-endtest
: This is an endtest which determines if it is safe to step to the next value of the iteration variable.

steps
: Variables that get stepped. This is internally manipulated as a list of the form (*var1 val1 var2 val2* ...); all of those variables are stepped in parallel, meaning that all of the *val*s are evaluated before any of the *var*s are set.

post-step-endtest
: Sometimes you can't see if you are done until you step to the next value; that is, the endtest is a function of the stepped-to value.

pseudo-steps
: Other things that need to be stepped. This is typically used for internal variables that are more conveniently stepped here, or to set up iteration variables that are functions of some internal variable(s) actually driving the iteration. This is a list like *steps*, but the variables in it do not get stepped in parallel.

The above alone is actually insufficient in just about all the iteration-driving clauses that **loop** handles. What is missing is that in most cases the stepping and testing for the first time through the loop is different from that of all other times. So, what **loop** deals with is two four-tuples as above; one for the first iteration, and one for the rest. The first may be thought of as describing code that immediately precedes the loop in the **prog**, and the second following the body code—in fact, loop does just this, but severely perturbs it in order to reduce code duplication. Two lists of forms are constructed in parallel: one is the first-iteration endtests and steps, the other the remaining-iterations endtests and steps. These lists have dummy entries in them so that identical expressions will appear in the same position in both. When loop is done parsing all of the clauses, these lists get merged back together such that corresponding identical expressions in both lists are not duplicated unless they are "simple" and it is worth doing.

Thus, one *may* get some duplicated code if one has multiple iterations. Alternatively, loop may decide to use and test a flag variable that indicates whether one iteration has been performed. In general, sequential iterations have less overhead than parallel iterations, both from the inherent overhead of stepping multiple variables in parallel, and from the standpoint of potential code duplication.

One other point that must be noted about parallel stepping is that although the user iteration variables are guaranteed to be stepped in parallel, the placement of the endtest for any particular iteration may be either before or after the stepping. A notable case of this is

```
(loop for i from 1 to 3 and dummy = (print 'foo)
      collect i)
   => (1 2 3)
```

but prints foo *four* times. Certain other constructs, such as for *var* on, may or may not do this depending on the particular construction.

This problem also means that it may not be safe to examine an iteration variable in the epilogue of the loop form. As a general rule, if an iteration-driving clause implicitly supplies an endtest, then one cannot know the state of the iteration variable when the loop terminates. Although one can guess on the basis of whether the iteration variable itself holds the data upon which the endtest is based, that guess *may* be wrong. Thus,

```
(loop for subl on expr
      ...
      finally (f subl))
```

is incorrect, but

```
(loop as frob = expr while (g frob)
      ...
      finally (f frob))
```

is safe because the endtest is explicitly dissociated from the stepping.

19.6 Iteration Paths

Iteration paths provide a mechanism for user extension of iteration-driving clauses. The interface is constrained so that the definition of a path need not depend on much of the internals of loop. The typical form of an iteration path is

```
for var being {each|the} path {preposition1 expr1}...
```

path is an atomic symbol which is defined as a loop path function.
Any number of preposition/expression pairs may be present; the prepositions allowable for any particular path are defined by that path. For example,

```
(loop for x being the array-elements of my-array from 1 to 10
      ...)
```

To enhance readability, paths are usually defined in both the singular and plural forms; this

particular example could have been written as
```
        (loop for x being each array-element of my-array from 1 to 10
              ...)
```

Another format, which is not so generally applicable, is
```
        for var being expr0 and its path {preposition1 expr1}...
```
In this format, *var* takes on the value of *expr0* the first time through the loop. Support for this format is usually limited to paths for which the next value is obtained by operating on the previous value. Thus, we can hypothesize the **cdrs** path, such that
```
        (loop for x being the cdrs of '(a b c . d) collect x)
         => ((b c . d) (c . d) d)
```
but
```
        (loop for x being '(a b c . d) and its cdrs collect x)
         => ((a b c . d) (b c . d) (c . d) d)
```
To satisfy the anthropomorphic among you, **his**, **her**, or **their** may be substituted for the **its** keyword, as may **each**. Egocentricity is not condoned. Some example uses of iteration paths are shown in section 19.6.1.

Very often, iteration paths step internal variables which the user does not specify, such as an index into some data-structure. Although in most cases the user does not wish to be concerned with such low-level matters, it is occasionally useful to have a handle on such things. **loop** provides an additional syntax with which one may provide a variable name to be used as an "internal" variable by an iteration path, with the **using** "prepositional phrase". The **using** phrase is placed with the other phrases associated with the path, and contains any number of keyword/variable-name pairs:
```
        (loop for x being the array-elements of a using (index i)
              ...)
```
which says that the variable i should be used to hold the index of the array being stepped through. The particular keywords which may be used are defined by the iteration path; the **index** keyword is recognized by all **loop** sequence paths (section 19.6.1.3). Note that any individual **using** phrase applies to only one path; it is parsed along with the "prepositional phrases". It is an error if the path does not call for a variable using that keyword.

By special dispensation, if a *path* is not recognized, then the **default-loop-path** path will be invoked upon a syntactic transformation of the original input. Essentially, the **loop** fragment
```
        for var being frob
```
is taken as if it were
```
        for var being default-loop-path in frob
```
and
```
        for var being expr and its frob ...
```
is taken as if it were
```
        for var being expr and its default-loop-path in frob
```
Thus, this "undefined path hook" only works if the **default-loop-path** path is defined. Obviously, the use of this "hook" is competitive, since only one such hook may be in use, and the potential for syntactic ambiguity exists if *frob* is the name of a defined iteration path. This feature is not for casual use; it is intended for use by large systems that wish to use a special syntax for some feature they provide.

19.6.1 Pre-Defined Paths

loop comes with two pre-defined iteration path functions; one implements a mapatoms-like iteration path facility and the other is used for defining iteration paths for stepping through sequences.

19.6.1.1 The Interned-Symbols Path

The interned-symbols iteration path is like a mapatoms for loop.
```
        (loop for sym being interned-symbols ...)
```
iterates over all of the symbols in the current package and its superiors.
This is the same set of symbols over which mapatoms iterates, although not necessarily in the same order. The particular package to look in may be specified as in
```
        (loop for sym being the interned-symbols in package ...)
```
which is like giving a second argument to mapatoms.

You can restrict the iteration to the symbols directly present in the specified package, excluding inherited symbols, using the local-interned-symbols path:
```
        (loop for sym being the local-interned-symbols  {in package}
              ...)
```

Example:
```
        (defun my-apropos (sub-string &optional (pkg package))
            (loop for x being the interned-symbols in pkg
                  when (string-search sub-string x)
                    when (or (boundp x) (fboundp x) (plist x))
                      do (print-interesting-info x)))
```
In the Zetalisp and NIL implementations of loop, a package specified with the in preposition may be anything acceptable to the pkg-find-package function. The code generated by this path will contain calls to internal loop functions, with the effect that it will be transparent to changes to the implementation of packages. In the Maclisp implementation, the obarray *must* be an array pointer, *not* a symbol with an array property.

19.6.1.2 The Hash-Elements Path

The hash-elements path provides an effect like that of the function maphash. It can find all the occupied entries in a hash table.
```
        (loop for value being the hash-elements of hash-table ...)
```
iterates over all the occupied entries in *hash-table*. Each time, *value* is the value stored in the entry. To examine the keys of the entries as well, write
```
        (loop for value being the hash-elements of hash-table
              with-key keysym ...)
```
and then keysym's value each will be the hash key that corresponds to *value*.

19.6.1.3 Sequence Iteration

One very common form of iteration is done over the elements of some object that is accessible by means of an integer index. **loop** defines an iteration path function for doing this in a general way and provides a simple interface to allow users to define iteration paths for various kinds of "indexable" data.

define-loop-sequence-path *Macro*
path-name-or-names fetch-fun size-fun &optional *sequence-type default-var-type*

path-name-or-names is either an atomic path name or list of path names. *fetch-fun* is a function of two arguments, the sequence and the index of the item to be fetched. (Indexing is assumed to be zero-origined.) *size-fun* is a function of one argument, the sequence; it should return the number of elements in the sequence. *sequence-type* is the name of the data-type of the sequence, and *default-var-type* the name of the data-type of the elements of the sequence. These are applicable to use of **loop** in other Lisp systems; on the Lisp Machine they might as well be omitted.

The Zetalisp implementation of **loop** utilizes the Zetalisp array manipulation primitives to define both **array-element** and **array-elements** as iteration paths:

```
(define-loop-sequence-path (array-element array-elements)
    aref array-active-length)
```

Then, the **loop** clause

```
        for var being the array-elements of array
```

will step *var* over the elements of *array*, starting from element 0. The sequence path function also accepts **in** as a synonym for **of**.

The range and stepping of the iteration may be specified with the use of all of the same keywords which are accepted by the **loop** arithmetic stepper (**for** *var* **from** ...); they are **by**, **to**, **downto**, **from**, **downfrom**, **below**, and **above**, and are interpreted in the same manner. Thus,

```
        (loop for var being the array-elements of array
                  from 1 by 2
              ...)
```

steps *var* over all of the odd elements of *array*, and

```
        (loop for var being the array-elements of array
                  downto 0
              ...)
```

steps in reverse order.

```
        (define-loop-sequence-path (vector-elements vector-element)
            vref vector-length notype notype)
```

is how the **vector-elements** iteration path can be defined in NIL (which it is). One can then do such things as

```
        (defun cons-a-lot (item &restv other-items)
            (and other-items
                 (loop for x being the vector-elements of other-items
                       collect (cons item x))))
```

All such sequence iteration paths allow one to specify the variable to be used as the index variable, by use of the **index** keyword with the **using** prepositional phrase, as described (with an example) on page 364.

19.6.2 Defining Paths

This section and the next may not be of interest to those not interested in defining their own iteration paths.

In addition to the code which defines the iteration (section 19.5), a **loop** iteration clause (e.g. a **for** or **as** clause) produces variables to be bound and pre-iteration (*prologue*) code. This breakdown allows a user-interface to **loop** which does not have to depend on or know about the internals of **loop**. To complete this separation, the iteration path mechanism parses the clause before giving it to the user function that will return those items. A function to generate code for a path may be declared to loop with the **define-loop-path** function:

define-loop-path *pathname-or-names path-function list-of-allowable-prepositions* &rest *data* — *Macro*

This defines *path-function* to be the handler for the path(s) *path-or-names*, which may be either a symbol or a list of symbols. Such a handler should follow the conventions described below. The *datum-i* are optional; they are passed in to *path-function* as a list.

The handler will be called with the following arguments:

path-name
: The name of the path that caused the path function to be invoked.

variable
: The "iteration variable".

data-type
: The data type supplied with the iteration variable, or **nil** if none was supplied. This is a facility of the **loop** intended for other Lisp systems in which declaring the type of a variable produces more efficient code. It is not documented in this manual since it is never useful on the Lisp Machine.

prepositional-phrases
: This is a list with entries of the form *(preposition expression)*, in the order in which they were collected. This may also include some supplied implicitly (e.g. an **of** phrase when the iteration is inclusive, and an **in** phrase for the **default-loop-path** path); the ordering will show the order of evaluation that should be followed for the expressions.

inclusive?
: This is **t** if *variable* should have the starting point of the path as its value on the first iteration (by virtue of being specified with syntax like **for** *var* **being** *expr* **and its** *path*), **nil** otherwise. When **t**, *expr* will appear in *prepositional-phrases* with the **of** preposition; for example, **for x being foo and its cdrs** gets *prepositional-phrases* of **((of foo))**.

allowed-prepositions
: This is the list of allowable prepositions declared for the path that caused the path

function to be invoked. It and *data* (immediately below) may be used by the path function such that a single function may handle similar paths.

data This is the list of "data" declared for the path that caused the path function to be invoked. It may, for instance, contain a canonicalized path, or a set of functions or flags to aid the path function in determining what to do. In this way, the same path function may be able to handle different paths.

The handler should return a list of either six or ten elements:

variable-bindings
This is a list of variables that need to be bound. The entries in it may be of the form *variable* or (*variable expression*).
Note that it is the responsibility of the handler to make sure the iteration variable gets bound. All of these variables will be bound in parallel; if initialization of one depends on others, it should be done with a **setq** in the *prologue-forms*. Returning only the variable without any initialization expression is not allowed if the variable is a destructuring pattern.

prologue-forms
This is a list of forms that should be included in the **loop** prologue.

the four items of the iteration specification
These are the four items described in section 19.5, page 362: *pre-step-endtest*, *steps*, *post-step-endtest*, and *pseudo-steps*.

another four items of iteration specification
If these four items are given, they apply to the first iteration, and the previous four apply to all succeeding iterations; otherwise, the previous four apply to *all* iterations.

Here are the routines that are used by **loop** to compare keywords for equality. In all cases, a *token* may be any Lisp object, but a *keyword* is expected to be an atomic symbol. In certain implementations these functions may be implemented as macros.

si:loop-tequal *token keyword*
This is the **loop** token comparison function. *token* is any Lisp object; *keyword* is the keyword it is to be compared against. It returns **t** if they represent the same token, comparing in a manner appropriate for the implementation.

si:loop-tmember *token keyword-list*
The **member** variant of **si:loop-tequal**.

si:loop-tassoc *token keyword-alist*
The **assoc** variant of **si:loop-tequal**.

If an iteration path function desires to make an internal variable accessible to the user, it should call the following function instead of **gensym**:

si:loop-named-variable *keyword*

This should only be called from within an iteration path function. If *keyword* has been specified in a **using** phrase for this path, the corresponding variable is returned; otherwise, **gensym** is called and that new symbol returned. Within a given path function, this routine should only be called once for any given keyword.

If the user specifies a **using** preposition containing any keywords for which the path function does not call **si:loop-named-variable**, **loop** will inform the user of his error.

19.6.2.1 Path Definition Example

Here is an example function that defines the **string-characters** iteration path. This path steps a variable through all of the characters of a string. It accepts the format

```
(loop for var being the string-characters of str ...)
```

The function is defined to handle the path by

```
    (define-loop-path string-characters string-chars-path (of))
```

Here is the function:

```
    (defun string-chars-path (path-name variable data-type
                              prep-phrases inclusive?
                              allowed-prepositions data
                              &aux (bindings nil)
                                   (prologue nil)
                                   (string-var (gensym))
                                   (index-var (gensym))
                                   (size-var (gensym)))
      allowed-prepositions data ; unused variables
      data-type
      ; To iterate over the characters of a string, we need
      ; to save the string, save the size of the string,
      ; step an index variable through that range, setting
      ; the user's variable to the character at that index.
      ; We support exactly one "preposition", which is required,
      ; so this check suffices:
      (cond ((null prep-phrases)
             (ferror nil "OF missing in ~S iteration path of ~S"
                     path-name variable)))
      ; We do not support "inclusive" iteration:
      (cond ((not (null inclusive?))
             (ferror nil
               "Inclusive stepping not supported in ~S path ~
                of ~S (prep phrases = ~:S)"
               path-name variable prep-phrases)))
      ; Set up the bindings
      (setq bindings (list (list variable nil)
                           (list string-var (cadar prep-phrases))
                           (list index-var 0)
                           (list size-var 0)))
      ; Now set the size variable
      (setq prologue (list '(setq ,size-var (string-length
                                               ,string-var))))
      ; and return the appropriate stuff, explained below.
      (list bindings prologue
            '(= ,index-var ,size-var)
            nil nil
            (list variable '(aref ,string-var ,index-var)
                  index-var '(1+ ,index-var))))
```

The first element of the returned list is the bindings. The second is a list of forms to be placed in the *prologue*. The remaining elements specify how the iteration is to be performed. This example is a particularly simple case, for two reasons: the actual "variable of iteration", index-var, is purely internal (being gensymmed), and the stepping of it (1+) is such that it may be performed safely without an endtest. Thus index-var may be stepped immediately after the setting of the user's variable, causing the iteration specification for the first iteration to be identical to the iteration specification for all remaining iterations. This is advantageous from the standpoint of the optimizations **loop** is able to perform, although it is frequently not possible due to the semantics of the iteration (e.g., **for** *var* **first** *expr1* **then** *expr2*) or to subtleties of the stepping. It is safe for this path to step the user's variable in the *pseudo-steps* (the fourth item of an iteration specification) rather than the "real" steps (the second), because the step value can have no dependencies on any other (user) iteration variables. Using the pseudo-steps generally results in some efficiency gains.

If one desired the index variable in the above definition to be user-accessible through the **using** phrase feature with the **index** keyword, the function would need to be changed in two ways. First, **index-var** should be bound to **(si:loop-named-variable 'index)** instead of **(gensym)**. Secondly, the efficiency hack of stepping the index variable ahead of the iteration variable must not be done. This is effected by changing the last form to be

```
(list bindings prologue
      nil
      (list index-var '(1+ ,index-var))
      '(= ,index-var ,size-var)
      (list variable '(aref ,string-var ,index-var))
      nil
      nil
      '(= ,index-var ,size-var)
      (list variable '(aref ,string-var ,index-var)))
```

Note that although the second **'(= ,index-var ,size-var)** could have been placed earlier (where the second **nil** is), it is best for it to match up with the equivalent test in the first iteration specification grouping.

20. Defstruct

defstruct provides a facility in Lisp for creating and using aggregate datatypes with named elements. These are like structures in PL/I, or records in Pascal. In the last two chapters we saw how to use macros to extend the control structures of Lisp; here we see how they can be used to extend Lisp's data structures as well.

20.1 Introduction to Structure Macros

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 which 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 which defines a structure. For the space ship example above, we might define the structure by saying:

```
(defstruct (ship)
  "Represents a space 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 which, 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 also defines make-ship to be a macro or function (you can specify which one) that can create a ship object. So (setq s (make-ship)) makes a new ship, and sets s to it. This is called the *constructor*, 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 (see page 36), as follows (for example):

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

Here s is bound to a ship, and after the evaluation of the setf form, the ship-x-position of that ship is 100. Another way to change the contents of a structure is to use the alterant macro, which is described later, in section 20.4.3, page 387.

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; by default, it chooses 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-position (ship)
  (aref ship 0))
```

The constructor form (make-ship) performs (make-array 5), which makes an array of the appropriate size to be a ship. Note that a program which 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 section 11.5.1, page 230). 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 (setf (aref s 4) m), 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>
```

(The describe-defstruct function is explained more fully on page 376.)

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. Packages are this sort of an object, and so are the strings which Zmacs uses for storing lines of text.

The constructor function or macro allows you to specify values for slots in the new structure. defstruct allows you to specify default initial values for slots; whenever a structure is constructed and no value is specified for a slot, the slot's default initial value is stored in it.

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 one installed in Maclisp!)

Note that there is another version of defstruct used in Common Lisp programs, which is slightly incompatible. See section 20.8, page 393.

20.2 How to Use Defstruct

defstruct *Macro*

A call to defstruct looks like:

```
(defstruct (name options...)
  [doc-string]
  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 and 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 listed in the next section, or a list, whose car should be one of the option names and the rest of which should be arguments to the option. Some options have arguments that default; others require that arguments be given explicitly.

doc-string is a string which is recorded as the documentation of *name* as a structure. It can be accessed via (documentation '*name* 'structure). It is not required.

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

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

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 is defined with the name *slot-name* (but see the :conc-name option, page 379). Form (2) is similar, but allows a default initialization for the slot. Initialization is explained further on page 385. Form (3) lets you pack several slots into a single element of the physical underlying structure, using the byte field feature of defstruct, which is explained on page 387.

Forms (2) and (3) allow *slot-options* which are alternating keywords and values (unevaluated). These slot option keywords are defined:

:read-only *flag*	If *flag* is non-nil, this specifies that this slot should not be changed in an existing structure. setf will not be allowed on the slot's accessor.
:type *type-spec*	Declares that the contents of this slot must be of type *type-spec*. The Lisp machine compiler does not use this information, but sometimes it enables defstruct to deduce that it can pack the structure into less space by using a specialized array type.
:documentation *documentation-string*	Makes *documentation-string* the documentation for the slot's accessor function. It also goes in the si:defstruct-slot-description-documentation for this slot in the defstruct-description structure.

Here is an eggsample of using slot options:

```
(defstruct (eggsample :named :conc-name)
  (yolk 'a
   :documentation "First thing you need in an eggsample.")
  (grade 3)
  (albumen nil :read-only t))

(documentation 'eggsample-yolk 'function)
=> "First thing you need in an eggsample."

(setf (eggsample-albumen (make-eggsample)) 'eggsistential)
>>ERROR: SETF is forbidden on EGGSAMPLE-ALBUMEN.
While in the function ...
```

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 which 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 example above, all the names started with ship-. The :conc-name option can be used to provide such prefixes automatically (see page 379). Similarly, the conventional name for the constructor in the example above was **make-ship**, and the conventional name for the alterant macro (see section 20.4.3, page 387) was **alter-ship**.

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

describe-defstruct *instance* &optional *name*

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 don't have to provide *name*, since it is just the named structure symbol of *instance*. Normally the **describe** function (see page 791) 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 page 390 for more information about named structures.

20.3 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 (mod 256)))
                (:make-array (:leader-length 3))
                :conc-name
                (:size-macro foo))
  a b)
```

:type
The :type option specifies what kind of Lisp object to use 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 in traditional programs, or :vector in Common Lisp programs. You can define your own types; this is explained in section 20.10, page 396.

:list Uses a list.

:named-list
Like :list, but the first element of the list holds the symbol that is the name of the structure and so is not used as a component.

:array
:typed-array
:vector
These are all synonymous. They use an array, storing components in the body of the array.

:named-array
Like :array, but makes the array a named structure (see page 390) using the name of the structure as the named structure symbol. Element 0 of

the array holds the named structure symbol and so is not used to hold a component of the structure.

:named-typed-array
:named-vector
These two synonyms are like **:named-array** but the array always has a leader and the named structure symbol is stored there. As a result, it is possible to use the **:subtype** option to specify a restricted array type, such as **art-8b**.

:phony-named-vector
This is what you get in Common Lisp if you say **(:type :vector)** and **:named**.

:array-leader
Use an array, storing components in the leader of the array. (See the **:make-array** option, described below.)

:named-array-leader
Like **:array-leader**, but makes the array a named structure (see page 390) using the name of the structure as the named structure symbol. Element 1 of the leader holds the named structure symbol and so is not used to hold a component of the structure.

: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**.

:named-fixnum-array
:named-flonum-array
Like **:fixnum-array** or **:flonum-array** but also a named structure, with a leader to hold the named structure symbol.

: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 (see page 387); it lets you store a bunch of small numbers within fields of a fixnum, giving the fields names.

:grouped-array
This is described in section 20.6, page 389.

The argument of **:type** may also have the form (*type subtype*). This is equivalent to specifying *type* for the **:type** option and *subtype* for the **:subtype** option. For example, **(:type (:array (mod 16.)))** specifies an array of four-bit bytes.

:subtype For structures which are arrays, **:subtype** permits the array type to be specified. It requires an argument, which must be either an array type name such as **art-4b** or a type specifier restricting the elements of the array. In other words, it should be a suitable value for either the *type* or the *element-type* argument to make-array.

If no :subtype option is specified but a :type slot option is given for every slot, defstruct may deduce a subtype automatically to make the structure more compact.

See section 2.3, page 14 for more information on type specifiers.

:constructor
: Specifies how to make a constructor for the structure. In the simplest use, there is one argument, which specifies the name to give to the standard keyword-argument constructor. 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. More complicated usage is explained in section 20.4.1, page 385.

:alterant
: Takes one argument, which specifies the name of the alterant macro. If the argument 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. Use of the alterant macro is explained in section 20.4.3, page 387.

In Common Lisp programs, the default for :alterant is nil; no alterant is defined. In traditional programs, the default is alter-*name*.

:predicate
: Causes defstruct to generate a predicate to recognize instances of the structure. Naturally it only works for "named" types. The argument to the :predicate option is the name of the predicate. If the option is present without an argument, then the name is formed by concatenating '-p' to the end of the name symbol of the structure. If the option is not present, then no predicate is generated. Example:

```
(defstruct (foo :named :predicate)
  a
  b)
```

defines a single argument function, foo-p, that is true only of instances of this structure.

The defaulting of the :predicate option is different (and complicated) in Common Lisp programs. See section 20.8, page 393.

:copier
: Causes defstruct to generate a single argument function that can copy instances of this structure. Its argument is the name of the copying function. If the option is present without an argument, then the name is formed by concatenating 'copy-' with the name of the structure. Example:

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

Generates a function approximately like:

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

: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 the accessor is used with no argument, it evaluates the default-pointer form to find a structure and accesses the appropriate component of that structure. 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 string to be used as the prefix, or a symbol whose pname is to be used. If :conc-name is specified without an argument, the prefix is the name of the structure followed by a hyphen. If the argument is **nil** or "", 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.

In Common Lisp programs, the default for :conc-name, when this option is not specified, is the structure name followed by a hyphen. For traditional programs, the default is **nil**.

The keywords recognized by the constructor and alterant are the slot names, not the accessor names, transfered into the keyword package. 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 Builds 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 :named :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 :named (:include person)
                      :conc-name)
   helmet-size
   (favorite-beverage 'tang))
```

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 **:include** option inserts the slots of the included structure at the front of the list of slots for this structure. That is, an **astronaut** has 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, such as **person-name**, can be used also on **astronaut**'s. New accessor functions are generated for these slots in the **astronaut** structure as if they were defined afresh; their names start with **astronaut-** instead of **person-**. In fact, the functions **person-age** and **astronaut-age** receive identical definitions.

Since the structures are named structures, recognizable by **typep**, **subtypep** considers **astronaut** a subtype of **person**, and **typep** considers any **astronaut** to be of type **person**.

The following examples illustrate how you can construct and use **astronaut** structures:

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

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

(typep x 'astronaut) => t
(typep x 'person) => t
```

Note that the :conc-name option was *not* inherited from the included structure; it is present for :astronaut only because it was specified explicitly in the definition. Similarly, the :default-pointer and :but-first options are not inherited from the :include'd structure.

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

```
(:include name new-descriptor-1 ... new-descriptor-n)
```

Each *new-slot-descriptor* is just like the slot descriptors used for defining new slots, except that byte fields are not allowed. The default initialization specified in *new-slot-descriptor*, or the absence of one, overrides what was specified in the included structure type (person). Any slot option values specified in *new-slot-descriptor* also override the values given in the included structure's definition. Any inherited slots for which no *new-slot-descriptor* is given, and any slot options not explicitly overridden, are inherited.

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

```
(defstruct (astronaut :conc-name
                      (:include person
                        (age 45. :documentation
                          "The ASTRONAUT's age in years.")))
   helmet-size
   (favorite-beverage 'tang))
```

If the :read-only option is specified as nil when t would have been inherited, an error is signaled.

: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.

The :named option works quite differently in Common Lisp programs; see section 20.8, page 393.

: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.

The argument to the :make-array option should be a list of alternating keyword symbols for the make-array function (see page 167), and forms whose values are the arguments to those keywords. For example, (:make-array (:area 'permanent-storage-area)) would request that the array be allocated in a particular area. 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. If you use the :initial-element make-array option, it initializes all the slots, but defstruct's own initializations are done afterward. If a subtype has been specified to or deduced by defstruct, it overrides any :type keyword in the :make-array argument.

Constructors for structures implemented as arrays recognize the keyword argument :make-array. Attributes supplied therein overide 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 may only be used with the default style of constructor that takes keyword arguments.

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 Used for structures of type :grouped-array to control the number of repetitions of the structure to be allocated by the constructor. (See section 20.6, page 389.) The constructor also accepts a keyword argument :times to override the value given in the defstruct. If :times appears in neither the invocation of the constructor nor as a defstruct option, the constructor allocates only one instance of the structure.

:size-macro Defines a special macro to expand into the size of this structure. 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 (for example, the length of the array or list). The argument of the :size-macro option is the name to be used for the macro. If this option is present without an argument, then the name of the structure is concatenated with '-size' to produce the macro name.

Example:
```
(defstruct (foo :conc-name :size-macro)
  a b)
(macroexpand '(foo-size)) => 2
```

:size-symbol Like :size-macro but defines a global variable rather than a macro. The size of the structure is the variable's value. Use of :size-macro is considered cleaner.

: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), which is the number of slots you want **defstruct** to skip. To make use of this option requires that you 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 is an accessor from some other structure, and it is never expected that this structure will be found outside of 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

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**, etc.). If the argument is **nil** then the accessors are really macros.

:callable-constructors

Controls whether constructors are really functions, and therefore "callable", or macros. An argument of **t** makes them functions; **nil** makes them macros. The default is **t** in Common Lisp programs, **nil** in traditional programs. See section 20.4.1, page 385 for more information.

: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 must look inside the **defstruct-description** structure himself (see page 394).

:print Controls the printed representation of his structure in a way independent of the Lisp dialect in use. Here is an example:

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

Of course, this only works if you use some named type, so that the system can recognize examples of this structure automatically.

The arguments to the :print option are arguments to the format function (except for the stream of course!). They are evaluated in an environment where the name symbol of the structure (foo in this case) is bound to the instance of the structure to be printed.

This works by generating a defselect that creates a named structure handler. Do not use the :print option if you define a named structure handler yourself, as they will conflict.

:print-function

is the Common Lisp version of the :print option. Its argument is a function to print a structure of this type, and it is called with three arguments: the structure to be printed, the stream to print it on, and the current printing depth (which should be compared with *print-level* to decide when to cut off recursion and print '#'). The function is expected to observe the values of the various printer-control variables such as *print-escape* (see page 514). Example:

```
(defstruct (bar :named :conc-name
                (:print-function
                  (lambda (struct stream depth)
                    depth ; unused
                    (sys:printing-random-object
                        (struct stream :type)
                      (format stream "with zap ~S"
                              (bar-zap struct))))))
  "The famous BAR structure."
  (zap 'yow)
  random-slot)

(make-bar) => #<BAR with zap YOW>
```

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 isn't found among those listed above, it should be a valid defstruct-keyword for the type of structure being defined, and the option should be of the form (*option-name value*). If so, the option is treated just like (:property *option-name value*). That is, the **defstruct** is given an *option-name* property of *value*.

This provides a primitive way for you to define your own options to **defstruct**, particularly in connection with user-defined types (see section 20.10, page 396). Several of the options listed above are actually implemented using this mechanism. They include :times, :subtype and :make-array.

The valid defstruct-keywords of a type are in a list in the **defstruct-keywords** slot of the **defstruct-type-description** structure for *type*.

20.4 Using the Constructor and Alterant

After you have defined a new structure with **defstruct**, you can create instances of this structure using the constructor, and you can alter the values of its slots using the alterant macro. By default, traditional **defstruct** defines both the constructor and the alterant, forming their names by concatenating 'make-' and 'alter-', respectively, onto the name of the structure. The **defstruct** for Common Lisp programs defines no alterant by default. 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 don't want the macro created at all by passing **nil** as the argument.

20.4.1 Constructors

A call to a constructor, in general, has the form
(*name-of-constructor keyword-1 value-1 keyword-2 value-2* ...)
Each *keyword* is a keyword (a symbol in the **keyword** package) whose name matches one of the slots of the structure, or one of a few specially recognized keywords.

The name of the constructor is specified by the **:constructor** option, which can also specify a documentation string for it:

```
(:constructor name-of-constructor [doc-string])
```

If a *keyword* matches the name of a slot (*not* the name of an accessor), then the corresponding *value* is used to initialize that slot of the new structure. Any slots whose values are not specified in this way are initialized to the values of the default initial value forms specified in the **defstruct**. If no default initial value was specified either for a slot, that slot's initial value is undefined. You should always specify the initialization, either in the **defstruct** or in the constructor invocation, if you care about the initial value of the slot.

Constructors may be macros or functions. They are functions if the **:callable-constructors** option to **defstruct** is non-nil. By default, they are functions in Common Lisp programs and macros in traditional programs.

Constructor macros allow the slot name (in its own package) to be used instead of a keyword. Constructor functions do not, as they are ordinary functions defined using **&key**. Old code using slot names not in the keyword package should be converted.

The default initial value forms are evaluated (if needed) each time a structure is constructed, so that if (gensym) is used as a default initial value form then a new symbol is generated for each structure. The order of evaluation of the default initial value forms is unpredictable. When the constructor is a macro, the order of evaluation of the keyword argument forms it is given is also unpredictable.

The two special keyword arguments recognized by constructors are :make-array and :times. :make-array should be used only for structures which are represented as arrays, and :times only for :grouped-array structures. If one of these arguments is given, then it overrides the :make-array option or the :times option (see page 381) specified in the defstruct. For example:
For example,

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

User-defined types of structures can define their own special constructor keywords.

20.4.2 By-Position Constructors

If the :constructor option is given as (:constructor *name arglist* [*doc-string*]), then instead of making a keyword driven constructor, defstruct defines a positional constructor, taking arguments whose meaning is determined by the argument's position rather than by a keyword. The *arglist* is used to describe what arguments the constructor should accept. 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))
  "Make a FOO, with positional arguments")
```

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 isn't any 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 isn't any 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, even if a default value was specified in its slot-description. 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 override completely 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. These may include both keyword and positional constructors. If there are multiple keyword constructors, they all behave the same, differing only in the name. It is important to have a keyword constructor because otherwise the #S reader construct cannot work.

Note that positional constructors may be macros or functions, just like keyword constructors, and based on the same criterion: they are functions if the :callable-constructors option to defstruct is non-nil. By default, they are functions in Common Lisp programs and macros in traditional programs. If the positional constructor is a macro, then the actual order of evaluation of its arguments is unpredictable.

Also note that you cannot specify the :make-array or :times information in a positional constructor.

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

20.5 Byte Fields

The byte field feature of defstruct allows you to specify that several slots of your structure are bytes (see section 7.9, page 155) in an integer stored in one element of the structure. For example, suppose we had the following structure:

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

This works correctly but 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 (byte 10. 10.) 617.)
   (exchange (byte 10. 0)))
  line-number)
```

The expressions **(byte ...)** calculate 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 (byte 10. 10.) (caddr pbe))
(exchange pbe)  ==> (ldb (byte 10. 0) (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 (byte 10. 0) 631808.)
          7788.)

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

==> ((lambda (#:g0530)
        (setf (nth 2 #:g0530)
              (dpb ac (byte 10. 10.)
                      (dpb ex (byte 10. 0) (nth 2 #:g0530)))))
     pbe)
```

(This is how the expression would print; this text would not read in properly because a new uninterned symbol would be created by each use of **#:**.)

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 don't need to worry about the details of how the alterant macro does its work.

If the byte specifier is **nil**, then the accessor is 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 is 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.

Constructors (both functions and macros) initialize words divided into byte fields as if they were deposited in 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 invocation.

4) Initializations for the byte fields given in the constructor invocation.

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 is unpredictable.

20.6 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 is taken from the **:times** keyword of the constructor or the argument to the **:times** option to **defstruct**.

20.7 Named Structures

The *named structure* feature provides a very simple form of user-defined data type. (Flavors are another, more powerful, facility for defining data types, but they are more expensive in simple cases. See chapter 21, page 401.) A named structure is actually an array, containing elements and optionally a leader. The difference between a named structure and an ordinary array is that the named structure also contains an explicit slot to hold its ostensible data type. This data type is a symbol, any symbol the programmer cares to use. In traditional programs, named structures are normally defined using **defstruct** with the **:named** option. In Common Lisp programs, **defstruct** defines a named structure by default. Individual named structures are made with the constructors defined by **defstruct**.

The data type symbol of a named structure is also called the *named structure symbol*. It is stored in array element 0 if the structure has no leader. If there is a leader, the type symbol is stored in array leader element 1 (recall that element 0 is reserved for the fill pointer). If a numeric-type array is to be a named structure, it must have a leader, since a symbol cannot be an element of a numeric array.

Named structure are *recognizable*; that is, if you define a named structure called **foo**, you can always tell whether an object is a **foo** structure. No array created in the normal fashion, no matter what components it has, will be mistaken for a genuine **foo**.

Named structures can recognized by **typep**. Specify **foo**, the named structure name, as the second argument, and the object to be tested as the first argument. **type-of** of an ordinary array returns **array**, but **type-of** of a named structure returns the explicitly recorded data type symbol.

```
(defstruct (foo :named) a b)
(type-of (make-foo)) => foo
(typep (make-foo) 'foo) => t
```

Named structures of other types which include **foo** are also recognized as **foo**'s by **typep**. For example, using the previously-given definitions of **person** and **astronaut**, then

```
(typep (make-astronaut) 'person) => t
```

because the type **person** was explicitly included by the **defstruct** for **astronaut**. Indirect includes count also:

```
(defstruct (mission-specialist :named
             (:include astronaut))
  ...)

(typep (make-mission-specialist) 'person) => t
(subtypep 'person 'mission-specialist) => t
```

It should be emphasized that the named structure *is* an array. All the usual array functions, such as **aref** and **array-dimension**, can be used on it. If it is one-dimensional (as is usually the case) then the named structure is a vector and the generic sequence functions can be used on it.

```
(typep (make-astronaut) 'array) => t
(arrayp (make-astronaut)) => t
(array-rank (make-astronaut)) => 1
```

Because named structure data types are recognizable, they can define generic operations and say how to handle them. A few such operations are defined by the system and are invoked automatically from well-defined places. For example **print** automatically invokes the **:print-self** operation if you give it a named structure. Thus, each type of named structure can define how it should print. The standardly defined named structure operations are listed below. You can also define new named structure operations and invoke them by calling the named structure as a function just as you would invoke a flavor instance.

Operations on a named structure are all handled by a single function, which is found as the **named-structure-invoke** property of the structure type symbol. It is OK for a named structure type to have no handler function. Then invocation of any operation on the named structure returns **nil**, and system routines such as **print** take default actions.

If a handler function exists, it is given these arguments:

operation The name of the operation being invoked; usually a keyword.

structure The named structure which is being operated on.

additional-arguments...
Any other arguments which were passed when the operation was invoked. The handler function should have a rest parameter so it can accept any number of arguments.

The handler function should return **nil** if it does not recognize the *operation*. These are the named structure operations used by the system at present:

:which-operations
Should return a list of the names of the operations the function handles. Every handler function must handle this operation, and every opertation that the function handles should be included in this list.

:print-self Should output the printed representation of the named structure to a stream. The additional arguments are the stream to output to, the current depth in list-structure, and the current value of ***print-escape***. If **:print-self** is not in the value returned by **:which-operations**, or if there is no handler function, **print** uses **#s** syntax.

:describe Is invoked by **describe** and should output a description of the structure to ***standard-output***. If there is no handler function or **:describe** is not in its **:which-operations** list, **describe** prints the names and values of the structure's fields as defined in the **defstruct**.

:sxhash Is invoked by **sxhash** and should return a hash code to use as the value of **sxhash** for this structure. It is often useful to call **sxhash** on some (perhaps all) of the components of the structure and combine the results in some way.

There is one additional argument to this operation: a flag saying whether it is permissible to use the structure's address in forming the hash code. For some kinds of structure, there may be no way to generate a good hash code except to use the address. If the flag is nil, they must simply do the best they can, even if it means always returning zero.

It is permissible to return nil for :sxhash. Then sxhash produces a hash code in its default fashion.

:fasd-fixup Is invoked by fasload on a named structure that has been created from data in a QFASL file. The purpose of the operation is to give the structure a chance to "clean itself up" if, in order to be valid, it needs to have contents that are not exactly identical to those that were dumped. For example, readtables push themselves onto the list si:*all-readtables* so that they can be found by name.

For most kinds of structures it is acceptable not to define this operation at all (so that it returns nil).

Example handler function:

```
(defun (:property person named-structure-invoke)
       (op self &rest args)
  (selectq op
    (:which-operations '(:print-self :describe))
    (:describe
      (format (car args)
              "This is a ~D-year-old person"
              (person-age self)))
    (:print-self
      (if *print-escape*
          (si:printing-random-object (self (car args) :type)
            (princ (person-name self) (car args)))
        (princ (person-name self) (car args))))))
```

or

```
(defselect ((:property person named-structure-handler)
            ignore)
  (:print-self (self stream ignore &optional ignore)
    (if *print-escape*
        (si:printing-random-object (self stream :type)
          (princ (person-name self) stream))
      (princ (person-name self) stream)))
  (:describe (self)
    (format *standard-output*
            "This is a ~D-year-old person"
            (person-age self))))
```

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. This simple example could have been done even more simply with the :print-function option.

It is often convenient to define a handler function with **defselect**; but you must be careful. **defselect** by default defines the function to signal an error if it is called with a first argument that is not recognized. A handler function should return **nil** and get no error. To avoid the problem, specify **ignore** as the default handler when you write the **defselect**. See page 236.

Note that the handler function of a named structure type is *not* inherited by other named structure types that include it. For example, the above definition of a handler for **person** has no effect at all on the **astronaut** structure. If you need such inheritance, you must use flavors rather than named structures (see chapter 21, page 401).

The following functions operate on named structures.

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

make-array-into-named-structure *array*
Marks *array* as a named structure and returns it. This is used by **make-array** when creating named structures. You should not normally call it explicitly.

named-structure-invoke *operation structure* &rest *args*
Invokes a named structure operation on *structure*. *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.

If the structure type has no **named-structure-invoke** property, **nil** is returned. By convention, **nil** is also returned by the handler if it does not recognize **operation**.

(send *structure operation args...***)** has the same effect, by calling **named-structure-invoke**.

See also the **:named-structure-symbol** keyword to **make-array**, page 167.

20.8 Common Lisp Defstruct

cli:defstruct *Macro*
The version of **defstruct** used in Common Lisp programs differs from the traditional **defstruct** in the defaults for a few options and the meanings of a few of them.

The **:conc-name** option defaults to the structure type name followed by a hyphen in **cli:defstruct**. In traditional **defstruct** it defaults to **nil**.

The **:callable-constructors** option defaults to **t** in **cli:defstruct**, so that the constructor is a function. Traditionally, it defaults to **nil**.

The **:alterant** option defaults to **nil** in **cli:defstruct**, so that no alterant is defined. Traditionally, an alterant is defined by default with the name **alter-***name*.

The :type option defaults to :named-vector in cli:defstruct. This makes a named structure, and you may specify how to print it. The :predicate option defaults to t in this case.

If the :type option is specified in cli:defstruct, you never get a named structure. You get either a plain list or a plain vector. There is no type-testing predicate, and you may not request one. You may not say how to print the structure, either.

If you specify the :named option along with :type, you still *do not* get a named structure. You get a plain list or a plain vector in which the structure name happens to be stored. The type is either :named-list or :phony-named-vector. The :predicate option defaults to nil, but you may specify t yourself. However, any randomly created list or vector with the structure name stored in the right place will satisfy the predicate thus defined. typep cannot recognize these phony named structures, and you may not specify how to print them (they do not understand named-structure-invoke.)

20.9 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 (see page 791) and the Inspector do, your program should work by examining these structures. The information in this section is also necessary for anyone who is thinking of defining his own structure types.

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

This is a simplified version of the way the si:defstruct-description structure is defined. It omits some slots whose meaning is not worth documenting here. (The actual definition is in the system-internals package.)

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

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

The size slot contains the total number of slots 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 :property option, page 383).

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 (with other slots that are omitted here), also in the si package:

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

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 a number 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 page 397.

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 his defstruct form. If there is no initialization code for this slot then the init-code slot contains a canonical object which can be obtained (for comparison using eq) as the result of (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).

20.10 Extensions to Defstruct

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

si:defstruct-define-type *Macro*

Is used for teaching defstruct about new types.

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

```
(si: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.

The semantics of si:defstruct-define-type is the subject of the rest of this section.

20.10.1 Example

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

```
(si: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 page 394). The third variable and the :list keyword will be explained later.

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.

20.10.2 Options to si:defstruct-define-type

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

:cons
: Specifies the code 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*). Additional kinds may be provided in the future.

  The symbol *description* will be bound to the instance of the defstruct-description structure (see page 394) that defstruct maintains for this particular 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 that wasn't 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 on using the constructor, section 20.4.1, page 385. You specify the list of acceptable keywords with the :cons-keywords option (see page 398).

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

:ref
: Specifies the code 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 page 395).

The symbol *description* will be bound to the instance of the **si: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 page 379). **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 **si:defstruct-define-type**.

:overhead Declares 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. For example, named arrays have an overhead of one, and named array leaders an overhead of two, but named typed arrays have no overhead since the structure type symbol is stored in the array leader whilst the actual data specifying the values of the slots is stored in the array proper.

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 page 382).

:named Controls the use of the **:named** option to **defstruct**. With no argument, the **:named** option means that this type is one which records the structure type name somehow (not necessarily by using an actual 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 the user specifies this type and **:named** as well. For example, the definition of the **:list** type contains **(:named :named-list)**, saying that a **defstruct** that specifies **(:list :named)** really uses type **:named-list**.

:cons-keywords

Defines additional constructor keywords for this type of structure. Using these keywords, one may specify additional information about a structure at the time it is created ("consed") using one of its constructor functions or macros. (The **:times** constructor keyword for structures of type **:grouped-array** is an example.) The syntax is: **(:cons-keywords** *keyword-1* ... *keyword-n***)** where each *keyword* is a symbol that the constructor conser expects to find in the *keywords* alist (explained above).

:keywords **:keywords** is an old name for the **:cons-keywords option.**

:defstruct-keywords

Defines additional **defstruct** options allowed for this type of structure. (The **:subtype** option for structures of type **:array** is an example.) These options take effect at the time the structure is defined using **defstruct**, and thus affect all structures of a particular type (unless overridden in some way.) In contrast, the **:cons-keywords** options affect the creation of individual structures of a particular type.

The syntax is: (:defstruct-keywords *keyword-1* ... *keyword-n*) where each *keyword* is a keyword that defstruct will recognize as an option. defstruct puts such options, with their values, in the property-alist slot of the defstuct-description structure (defined above)

:predicate Tells defstruct how to produce predicates for a particular type (for the :predicate option to defstruct). Its syntax is:

```
(:predicate (description name)
  body...)
```

The variable *description* is bound to the defstruct-description structure maintained for the structure for which a predicate is being generated. The variable *name* is bound to the symbol that is to be defined as a predicate. *body* is a piece of code to compute the defining form for the predicate. A typical use of this option might look like:

```
(:predicate (description name)
  `(defun ,name (x)
     (and (frobbozp x)
          (eq (frobbozref x 0)
              ',(si:defstruct-description-name
                  description)))))
```

:copier defstruct knows how to generate a copier function using the constructor and reference code that must be provided with any new defstruct type. Nevertheless it is sometimes desirable to specify a specific method of copying a particular defstruct type. The :copier option to si:defstruct-define-type allow this to be done:

```
(:copier (description name)
  body)
```

As with the :predicate option, *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. For example:

```
(:copier (description name)
  `(fdefine ',name 'copy-frobboz))
```

:defstruct The :defstruct option to si:defstruct-define-type allows the user 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 don't want to actually output any additional code, then you should be careful to return **nil** from the code in this option.

defstruct will cause the *body* forms to be evaluated as early as possible in the parsing of a structure definition, and cause the returned forms to be evalutated as late as possible in the macro-expansion of the **defstuct** forms. This is so that *body* can rehack arguments, signal errors, and the like before many of **defstruct's** internal forms are executed, while enabling it to return code which will modify or extend the default forms produced by a vanilla **defstruct**.

21. Objects, Message Passing, and Flavors

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.

21.1 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 which "piece of text" the window is associated with.

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, for each type, a set of operations that can be performed on any object of the type.

The new types may exist only in the programmer's mind. For example, it is possible to think of a disembodied property list as an abstract data type on which certain operations such as **get** and **putprop** are defined. This type can be instantiated with **(cons nil nil)** (that is, by evaluating this form you can create a new disembodied property list); the operations are invoked through functions defined just for that purpose. The fact that disembodied property lists are really implemented as lists, indistinguishable from any other lists, does not invalidate this point of view. However, such conceptual data types cannot be distinguished automatically by the system; one cannot ask "is this object a disembodied property list, as opposed to an ordinary list".

The **defstruct** for **ship** early in chapter 20 defines another conceptual type. **defstruct** automatically defines some operations on this object, the operations to access its elements. We could define other functions that did useful things with **ship**'s, 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 disembodied property list case, the Lisp object is a list of pairs; 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-x-position** *ship***) 5.0)** alters it.

We have now seen the essence of object-oriented programming. A conceptual object is modeled 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.

21.2 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, they help and encourage 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 **putprop**'s 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 :conc-name)
  x-position
  y-position
  x-velocity
  y-velocity
  mass)

(defun ship-speed (ship)
  (sqrt (+ (^ (ship-x-velocity ship) 2)
           (^ (ship-y-velocity ship) 2))))

(defun ship-direction (ship)
  (atan2 (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 :conc-name)
  x-position
  y-position
  speed
  direction
  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 uses; 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 **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 that 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 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. Another reason is the traditional MIT AI Lab philosophy that opposes "fascist" restrictions which impose on the user "for his own good". The user himself should decide what is good for him.

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)**.

Often a few abstract operation functions are simple enough that it is desirable 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, substs, 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 :ordered-instance-variables feature is used; see page 427, where this problem is explained further. If you don't use the :ordered-instance-variables feature, you don't have to worry about this.

21.3 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 ship's 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 needs to know the ship's attributes, and must therefore 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 accepts integers, floats, ratios and complex numbers, and perform an appropriate kind of addition, 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 ship's, meteor's, 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.

Another terminology for the use of such generic operations has emerged from the Smalltalk language: performing a generic operation is called *sending a message*. The message consists of an operation name (a symbol) and arguments. The objects in the program are thought of as little people, who get sent messages and respond with answers (returned values). In the example above, the objects are sent x-position messages, to which they respond with their *x* position.

Sending a message is a way of invoking a function without specifying which function is to be called. Instead, the data determines the function to use. The caller specifies an operation name and an object; that is, it said what operation to perform, and what object to perform it on. The function to invoke is found from this information.

The two data used to figure out which function to call are the *type* of the object, and the *name* of the operation. 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 operation is data which are passed as arguments to the function, so the operation 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 operation on 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 consisting of the operation x-position and 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.

21.4 Generic Operations in Lisp

How do we implement message passing in Lisp? Our convention is that objects that receive messages are always *functional* objects (that is, you can apply them to arguments). A message is sent to an object by calling that object as a function, passing the operation name as the first argument and the arguments of the message as the rest of the arguments. Operation names are represented by symbols; normally these symbols are in the keyword package (see chapter 27, page 636), 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:

```
(send my-ship :x-position)
```

To set the ship's *x* position to 3.0, we send it a message like this:

```
(send 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 an operation name. The object must consider this operation name, find the function which is the method for that operation, and invoke that function.

send *object operation* &rest *arguments*
Sends *object* a message with operation and arguments as specified. Currently send is identical to funcall, but preferable when a message is being sent, just for clarity.

There are vague ideas of making *send* different from *funcall* if *object* is a symbol, list, number, or other object that does not normally handle messages when funcalled, but the meaning of this is not completely clear.

lexpr-send *object operation* &rest *arguments*
Currently lexpr-send is the same as *apply*.

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. But an ordinary function will not do. We need something that can store 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 chapter 12, page 250). 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. (Actually, rather than using closures and a selectq, you would probably use entities (section 12.4, page 255) and defselect (page 236).)

While using closures (or entities) 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 (and entities) 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 are very easy to do with flavors. The rest of this chapter describes flavors.

21.5 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) ()
  (atan2 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 operation is a symbol with the same name as the instance variable, but interned on the keyword package. Thus, methods are created to handle the operations :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 the operation :speed. The second subform is the lambda-list, and the rest is the body of the function that handles the :speed operation. The body can refer to or set any instance variables of the flavor, just like variables bound by a containing let. When any instance of the ship flavor is invoked with a first argument of :direction, the body of the second defmethod is 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 the arguments passed to cli:atan are the the velocity components of this particular ship. The result of cli:atan becomes the value returned by the :direction operation.

Now we have seen how to create a new abstract type: a new flavor. Every instance of this flavor has 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 returns an object whose printed representation is #<SHIP 13731210>. (Of course, the value of the magic number will vary; it is just the object address in octal.) The argument to make-instance is 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 operation :set for first argument :x-position, :y-position, and so on; each such method takes one additional argument and sets the corresponding instance variable to that value. It also generates methods for the operations :set-x-position, :set-y-position and so on; each of these takes one argument and sets the corresponding variable.

Another option we can add to the defflavor is :inittable-instance-variables, which allows us to initialize the values of the instance variables when an instance is first created. :inittable-instance-variables does not create any methods; instead, it makes *initialization keywords* named :x-position, :y-position, etc., that can be used as init-option arguments to make-instance to initialize the corresponding instance variables. The list of init options is sometimes called the *init-plist* because it is 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
  :inittable-instance-variables)
```

All we have to do is evaluate this new defflavor, and the existing flavor definition is updated and now includes the new methods and initialization options. In fact, the instance we generated a while ago now accepts the new operations! We can set the mass of the ship we created by evaluating

```
(send my-ship :set-mass 3.0)
```

or

```
(send my-ship :set :mass 3.0)
```

and the mass instance variable of my-ship is properly set to 3.0. Whether you use :set-mass or the general operation :set is a matter of style; :set is used by the expansion of (setf (send my-ship :mass) 3.0).

If you want to play around 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:          void
        Y-POSITION:          void
        X-VELOCITY:          void
        Y-VELOCITY:          void
        MASS:                3.0
```

Now that the instance variables are *inittable*, 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:          void
        Y-VELOCITY:          void
        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
  :inittable-instance-variables)

(setq another-ship (make-instance 'ship :x-position 3.4))
 => #<SHIP 14563643>

(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:               void
```

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 void.

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.

21.6 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 defmethod's 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 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)
   :inittable-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 still implement the :x-velocity and :speed operations, with the same meaning as before. However, the :engine-power operation is also understood (and returns 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 non-terminal nodes *before* the subtrees they head, ignoring any flavor that has been encountered previously somewhere else in the tree. 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 does not cause a non-terminating 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 has an instance variable named foo, and any methods that refer to foo 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; 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 operation supported by the flavor. This function is constructed out of all the methods for that operation 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 an operation by providing your own method. Your method will be called, and foo's will never be called.

Simple overriding is often useful; for example, if you want to make a new flavor bar that is just like foo except that it reacts completely differently to a few operations. 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 operation, 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.

daemon methods 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 operation, 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 run before any of those in **foo**, and **bar**'s after-daemons 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 make this a bit more clear, let's consider a simple example that is easy to play with: the **:print-self** method. The Lisp printer (i.e. the **print** function; see section 23.1, page 506) prints instances of flavors by sending them **:print-self** messages. The first argument to the **:print-self** operation 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** operation, then that method completely takes over the job of printing; **vanilla-flavor**'s method will not be called at all. However, if we give **ship** a before-daemon method for the **:print-self** operation, then it will get invoked before the **vanilla-flavor** method, 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 section 21.11, page 433. **vanilla-flavor** and what it does for you are also explained later; see section 21.10, page 432.

21.7 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 is given an **si:flavor** property which is the internal data-structure containing the details of the flavor.

(type-of *obj***)**, where *obj* is an instance of the flavor named *flavor-name*, returns 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*, etc. 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 is 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 remains void.

flav1, *flav2*, etc. 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*, etc. 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 section 21.8, page 424.

all-flavor-names *Variable*

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 operation for instances of a particular flavor, is defined by a form such as

```
        (defmethod (flavor-name method-type operation) lambda-list
          form1 form2...)
```

flavor-name is a symbol which is the name of the flavor which is to receive the method. *operation* is a keyword symbol which names the operation to be handled. *method-type* is a keyword symbol for the type of method; it is omitted when you are defining a primary method. For some method-types, additional information is expected. It comes after *operation*.

The meaning of *method-type* depends on what style of method combination is declared for this operation. For instance, if :daemon combination (the default style) is in use, method types **:before** and **:after** are allowed. See section 21.11, page 433 for 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 operation name itself, is automatically handled and so is not included in the lambda-list. Note that methods may not have unevaluated (**&quote**) arguments; that is, they must be functions, not special forms. *form1*, *form2*, etc. are the function body; the value of the last form is returned.

The variant form

```
        (defmethod (flavor-name operation) function)
```

where *function* is a symbol, says that *flavor-name*'s method for *operation* is *function*, a symbol which names a function. That function must take appropriate arguments; the first argument is the operation. When the function is called, **self** will be bound.

If you redefine a method that is already defined, the old definition is replaced by the new one. Given a flavor, an operation name, and a method type, there can only be one function (with the exception of :case methods; see page 437), so if you define a **:before** daemon method for the **foo** flavor to handle the :bar operation, then you replace the previous before-daemon; however, you do not affect the primary method or methods of any other type, operation or flavor.

The function spec for a method (see section 11.2, page 225) looks like:

```
(:method flavor-name operation)   or
(:method flavor-name method-type operation)   or
(:method flavor-name method-type operation suboperation)
```

This is useful to know if you want to trace (page 738), breakon (page 741) or advise (page 742) a method, or if you want to poke around at the method function itself, e.g. disassemble it (see page 792).

make-instance *flavor-name init-option1 value1 init-option2 value2...*

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. An :init message 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 (however, init keywords that simply initialize instance variables, and the corresponding values, may be absent when the :init methods are called). make-instance is an easy-to-call interface to instantiate-flavor, below.

If :allow-other-keys is used as an init keyword with a non-nil value, this error check is suppressed. Then unrecognized keywords are simply ignored. Example:

```
(make-instance 'foo :lose 5 :allow-other-keys t)
```

specifies the init keyword :lose, but prevents an error should the keyword not be handled.

instantiate-flavor *flavor-name init-plist* &optional *send-init-message-p return-unhandled-keywords area*

This is an extended version of make-instance, giving you more features. Note that it takes the *init-plist* as an individual argument, rather than taking a rest argument of init options and values.

The *init-plist* argument must be a disembodied property list; locf of a rest argument is satisfactory. Beware! This property list can be modified; the properties from the default init plist are putprop'ed on if not already present, and some :init methods do explicit putprop's onto the *init-plist*.

In the event that :init methods remprop properties already on the *init-plist* (as opposed to simply doing get and putprop), then the *init-plist* is rplacd'ed. This means that the actual supplied list of options is modified. It also means that locf of a rest argument does not work; the caller of instantiate-flavor must copy its rest argument (e.g. with copylist); this is because rplacd is not allowed on stack lists.

Do not use nil as the *init-plist* argument. This would mean to use the properties of the symbol nil as the init options. If your goal is to have no init options, you must provide a property list containing no properties, such as the list (nil).

Here is the sequence of actions by which instantiate-flavor creates a new instance:

First, the specified flavor's instantiation flavor function (page 429), if it exists, is called to determine which flavor should actually be instantiated. If there is no instantiation flavor function, the specified flavor is instantiated.

If the flavor's method hash-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 or even invoke the compiler, but it happens only once for each time you define or redefine a particular flavor.

Next, the instance itself is created. If the *area* argument is specified, it is the number of an area in which to cons the instance; otherwise the flavor's instance area function is called to choose an area if there is one; otherwise, default-cons-area is used. See page 429.

Then the initial values of the instance variables are computed. If an instance variable is declared inittable, and a keyword with the same spelling as its name appears in *init-plist*, the property for that keyword is used as the initial value.

Otherwise, if the default init plist specifies such a property, it is evaluated and the value is used. Otherwise, if the flavor definition specifies a default initialization form, it is evaluated and the value is used. The initialization form may not refer to any instance variables. It can find the new instance in self but should not invoke any operations on it and should not refer directly to any instance variables. It can get at instance variables using accessor macros created by the :outside-accessible-instance-variables option (page 427) or the function symeval-in-instance (page 423).

If an instance variable does not get initialized either of these ways it is left void; an :init method may initialize it (see below).

All remaining keywords and values specified in the :default-init-plist option to defflavor, that do not initialize instance variables and are not overridden by anything explicitly specified in *init-plist* are then merged into *init-plist* using putprop. The default init plist of the instantiated flavor is considered first, followed by those of all the component flavors in the standard order. See page 425.

Then keywords appearing in the *init-plist* but not defined with the :init-keywords option or the :inittable-instance-variables option for some component flavor are collected. If the :allow-other-keys option is specified with a non-nil value (either in the original *init-plist* argument or by some default init plist) then these *unhandled* keywords are ignored. If the *return-unhandled-keywords* argument is non-nil, a list of these keywords is returned as the second value of instantiate-flavor. Otherwise, an error is signaled if any unrecognized init keywords are present.

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.

The :init methods should not look on the *init-plist* for keywords that simply initialize instance variables (that is, keywords defined with :inittable-instance-variables rather than :init-keywords). The corresponding instance variables are already set up when the :init methods are called, and sometimes the keywords and their values may actually be missing from the *init-plist* if it is more efficient not to put them on. To avoid problems, always refer to the instance variables themselves rather than looking for the init keywords that initialize them.

:init *init-plist* *Operation on all flavor instances*

This operation is implemented on all flavor instances. Its purpose is to examine the init keywords and perform whatever initializations are appropriate. *init-plist* is the argument that was given to instantiate-flavor, and may be passed directly to get to examine the value of any particular init option.

The default definition of this operation does nothing. However, many flavors add :before and :after daemons to it.

instancep *object*

Returns t if object is an instance. This is equivalent to (typep *object* 'instance).

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 that expands into code that 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 operation on flavor bar, which takes two arguments, and you have a lock-frobboz special-form that knows how to lock the lock (presumably it generates an unwind-protect). lock-frobboz needs to see the first argument to the operation; 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 macro defined by defwrapper from knowing the exact implementation and allows several defwrapper's 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))
         (t (print "About to do :FOO")
            . ,body)))
```

Suppose you need a variable for communication among the daemons for a particular operation: 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 operation and used free by the methods.

```
        (defvar *communication*)
        (defwrapper (bar :foo) (ignore . body)
          `(let ((*communication* nil))
             . ,body))
```

Similarly you might want a wrapper that puts a **catch** around the processing of an operation so that any one of the methods could throw out in the event of an unexpected condition.

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 execute within that wrapper's context.

:around methods can do some of the same things that wrappers can. See page 439. If one flavor defines both a wrapper and an **:around** method for the same operation, the **:around** method is executed inside the wrapper.

By careful about inserting the body into an internal lambda-expression within the wrapper's code. Doing so interacts with the internals of the flavor system and requires knowledge of things not documented in the manual in order to work properly. It is much simpler to use an **:around** method instead.

undefmethod (*flavor* [*type*] *operation* [*suboperation*]) *Macro*

```
        (undefmethod (flavor :before :operation))
```

removes the method created by

```
        (defmethod (flavor :before :operation) (args) ...)
```

To remove a wrapper, use **undefmethod** with **:wrapper** as the method type.

undefmethod is simply an interface to **fundefine** (see page 241) that accepts the same syntax as **defmethod**.

If a file that used to contain a method definition is reloaded and if that method no longer seems to have a definition in the file, the user is asked whether to **undefmethod** that method. This may be important to enable the modified program to inherit the methods it is supposed to inherit. If the method in question has been redefined by some other file, this is not done, the assumption being that the definition was merely moved.

undefflavor *flavor*

Undefines flavor *flavor*. All methods of the flavor are lost. *flavor* and all flavors that depend on it are no longer valid to instantiate.

If instances of the discarded definition exist, they continue to use that definition.

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).

funcall-self *operation arguments...*
lexpr-funcall-self *operation arguments... list-of-arguments*

funcall-self is nearly equivalent to **funcall** with **self** as the first argument. **funcall-self** used to be faster, but now **funcall** of **self** is just as fast. Therefore, **funcall-self** is obsolete. It should be replaced with **funcall** or **send** of **self**.

Likewise, **lexpr-funcall-self** should be replaced with use of **lexpr-send** to **self**.

funcall-with-mapping-table *function mapping-table* &rest *arguments*

Applies *function* to *arguments* with **sys:self-mapping-table** bound to *mapping-table*. This is faster than binding the variable yourself and doing an ordinary **funcall**, because the system assumes that the mapping table you specify is the correct one for *function* to be run with. However, if you pass the wrong mapping table, incorrect execution will take place.

This function is used in the code for combined methods and is also useful for the user in **:around** methods (see page 439).

lexpr-funcall-with-mapping-table *function mapping-table* &rest *arguments*

Applies *function* to *arguments* using **lexpr-funcall**, with **sys:self-mapping-table** bound to *mapping-table*.

declare-flavor-instance-variables (*flavor*) *body...* *Macro*

Sometimes it is useful to have 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)
      (defun function args body...))
```

surrounds the function definition with a peculiar kind of declaration which makes the instance variables of flavor *flavor-name* accessible by name. Any kind of function definition is allowed; it does not have to use **defun** per se.

If you call such a function when **self**'s value is an instance whose flavor does not include *flavor-name* as a component, it is an error.

Cleaner than using **declare-flavor-instance-variables**, because it does not involve putting anything around the function definition, is using a local declaration. Put **(declare (:self-flavor** *flavorname***))** as the first expression in the body of the function. For example:

```
(defun foo (a b)
  (declare (:self-flavor myobject))
  (+ a (* b speed)))
```
(where **speed** is an instance variable of the flavor **myobject**) is equivalent to
```
(declare-flavor-instance-variables (myobject)
(defun foo (a b)
  (+ a (* b speed))))
```

with-self-variables-bound *body...* *Special form*

Within the body of this special form, all of **self**'s instance variables are bound as specials to the values inside **self**. (Normally this is true only of those instance variables that are specified in **:special-instance-variables** when **self**'s flavor was defined.) As a result, inside the body you can use **set**, **boundp** and **symeval**, etc., freely on the instance variables of **self**.

recompile-flavor *flavor-name* &optional *single-operation* (*use-old-combined-methods* **t**) (*do-dependents* **t**)

Updates the internal data of the flavor and any flavors that depend on it. If *single-operation* is supplied non-**nil**, only the methods for that operation 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 are used if possible. New ones are generated only 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 are regenerated unconditionally. If *do-dependents* is **nil**, only the specific flavor you specified is recompiled. Normally all flavors that depend on it are also recompiled.

recompile-flavor affects only flavors that have already been compiled. Typically this means it affects flavors that have been instantiated, but does not bother with mixins (see page 431).

si:*dont-recompile-flavors* *Variable*

If this variable is non-**nil**, automatic recompilation of combined methods is turned off.

If you wish to make several changes each of which will cause recompilation of the same combined methods, you can use this variable to speed things up by making the recompilations happen only once. Set the variable to **t**, make your changes, and then set the variable back to **nil**. Then use **recompile-flavor** to recompile whichever combined methods need it. For example:
```
(setq si:*dont-recompile-flavors* t)
(undefmethod (tv:sheet :after :bar))
(defmethod (tv:sheet :before :bar) ...)
(setq si:*dont-recompile-flavors* nil)
(recompile-flavor 'tv:sheet :bar)
```
tv:sheet has very many dependents; **recompile-flavor** even once takes painfully long. It's nice to avoid spending the time twice.

compile-flavor-methods *flavor...* *Macro*

The form (compile-flavor-methods *flavor-name-1 flavor-name-2*...), placed in a file to be compiled, directs the compiler to include the automatically-generated combined methods for the named flavors in the resulting QFASL file, provided all of the necessary flavor definitions have been made. Furthermore, all internal data structures needed to instantiate the flavor will be computed when the QFASL file is loaded rather than waiting until the first attempt to instantiate it.

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, since if the the compiler must be invoked at run time, the program will be slow the first time it is run. (The compiler must be called in any case 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 is never to 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. In this unusual case, you should use the :abstract-flavor option in defflavor (page 428).

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.

The methods used by compile-flavor-methods to form the combined methods that go in the QFASL file are all those present in the file being compiled and all those defined in the Lisp world.

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 operation*

Given an object and an operation, this returns the object's method for that operation, 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 Meta-X List Combined Methods editor command (page 444) to find out what it does.

This is related to the :handler function spec (see section 11.2, page 223).

It is preferable to use the generic operation :get-handler-for.

flavor-allows-init-keyword-p *flavor-name keyword*
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 that contributes the support of that keyword.

si:flavor-all-allowed-init-keywords *flavor-name*
Returns a list of all the init keywords that may be used in instantiating *flavor-name*.

symeval-in-instance *instance symbol* &optional *no-error-p*
Returns the value of the instance variable *symbol* inside *instance*. If there is no such instance variable, an error is signaled, unless *no-error-p* is non-nil in which case nil is returned.

set-in-instance *instance symbol value*
Sets the value of the instance variable *symbol* inside *instance* to *value*. If there is no such instance variable, an error is signaled.

locate-in-instance *instance symbol*
Returns a locative pointer to the cell inside *instance* which holds the value of the instance variable named *symbol*.

describe-flavor *flavor-name*
Prints 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*
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
(*function-spec pathname*)
where the function spec starts with :method and has a method type of :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.

sys:unclaimed-message (error) *Condition*
This condition is signaled whenever a flavor instance is sent a message whose operation it does not handle. The condition instance supports these operations:

:object	The flavor instance that received the message.
:operation	The operation that was not handled.
:arguments	The list of arguments to that operation

21.8 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 signaled. 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 operation name is the name of the variable, in the keyword package (i.e. it has 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 operation name as one of the automatically generated methods, the explicit definition overrides the automatic one.

:settable-instance-variables

Enables automatic generation of methods for setting the values of instance variables. The operation name is ':set-' followed by the name of the variable. All settable instance variables are also automatically made gettable and inittable. (See the note in the description of the :gettable-instance-variables option, above.)

In addition, :case methods are generated for the :set operation with suboperations taken from the names of the variables, so that :set can be used to set them.

:inittable-instance-variables

The instance variables listed as arguments, or all instance variables listed in this **defflavor** if the keyword is given alone, are made *inittable*. 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**.

:special-instance-variables

The instance variables listed as arguments, or all instance variables listed in this **defflavor** if the keyword is given alone, will be bound dynamically when handling messages. (By default, instance variables are bound lexically with the scope being the method.) You must do this to any instance variables that you wish to be accessible through **symeval**, **set**, **boundp** and **makunbound**, since they see only dynamic bindings.

This should also be done for any instance variables that are declared globally special. If you omit this, the flavor system does it for you automatically when you instantiate the flavor, and gives you a warning to remind you to fix the **defflavor**.

:init-keywords
: The arguments are declared to be valid keywords to use in **instantiate-flavor** when creating an instance of this flavor (or any flavor containing it). 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 inittable instance variables or elements of this list. If any is not recognized, an error is signaled. When you write a :init method that accepts some keywords, they should be listed in the :init-keywords option of the flavor.

If :allow-other-keys is used as an init keyword with a non-nil value, this error check is suppressed. Then unrecognized keywords are simply ignored.

: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 :allow-other-keys t)
```

prevents errors for unhandled init keywords in all instantiation of this flavor and other flavors that depend on it.

:required-init-keywords
: The arguments are init keywords which are to be required each time this flavor (or any flavor containing it) is instantiated. An error is signaled if any required init keyword is missing.

:required-instance-variables
: Declares that any flavor incorporating this one that 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 that 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 accepts responsibility for 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-methods
: The arguments are names of operations that 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 operations. Typically this option appears in the **defflavor** for a base flavor (see page 431). 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 to be detected when the flavor instantiated or when **compile-flavor-methods** is done, rather than when the missing operation is used.

:required-flavors

The arguments are names of flavors that 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 purpose of declaring a flavor to be required is to allow instance variables declared by that flavor to be accessed. It also provides error checking: an attempt to instantiate a flavor that does not include the required flavors as components signals 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))
```

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 signaled, 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 instantiated, 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 operations which all objects are supposed to understand. These include **:print-self**, **:describe**, **:which-operations**, and several other operations. See section 21.10, page 432.

If any component of a flavor specifies the **:no-vanilla-flavor** option, then **si:vanilla-flavor** is not included in that flavor. This option should not be used casually.

:default-handler

The argument is the name of a function that is to be called to handle any operation for which there is no method. Its arguments are the arguments of the **send** which invoked the operation, including the operation name as the first argument. Whatever values the default handler returns are the values of the operation.

Default handlers can be inherited from component flavors. If a flavor has no default handler, any operation for which there is no method signals a **sys:unclaimed-message** error.

: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 that are specially known about by microcode, and also 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**.

Removing any of the **:ordered-instance-variables**, or changing their positions in the list, requires that you recompile all methods that use any of the affected instance variables.

:outside-accessible-instance-variables

The arguments are instance variables which are to be accessible from outside of this flavor's methods. A macro (actually a subst) 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 chapter 20, page 372.)

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 must be computed at run time. This takes noticeable time, although less than actually sending a message would take. An error is signaled if the argument to the accessor macro is not an instance or is an instance that 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 compiles a call to the accessor macro into a subprimitive which simply accesses that variable's assigned slot by number. This subprimitive is only three or four 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.

: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$)** causes it to be named **get$***v* instead.

:alias-flavor

Marks this flavor as being an alias for another flavor. This flavor should have only one component, which is the flavor it is an alias for, and no instance variables or other options. No methods should be defined for it.

The effect of the **:alias-flavor** option is that an attempt to instantiate this flavor actually produces an instance of the other flavor. Without this option, it would make an instance of this flavor, which might behave identically to an instance of the other flavor. **:alias-flavor** eliminates the need for separate mapping tables, method tables, etc. for this flavor, which becomes truly just another name for its component flavor.

The alias flavor and its base flavor are also equivalent when used as an argument of **subtypep** or as the second argument of **typep**; however, if the alias status of a flavor is changed, you must recompile any code which uses it as the second argument to **typep** in order for such code to function.

:alias-flavor is mainly useful for changing a flavor's name gracefully.

:abstract-flavor

This option marks the flavor as one that is not supposed to be instantiated (that is, is supposed to be used only as a component of other flavors). An attempt to instantiate the flavor signals an error.

It is sometimes useful to do **compile-flavor-methods** on a flavor that is not going to be instantiated, if the combined methods for this flavor will be inherited and shared by many others. **:abstract-flavor** tells **compile-flavor-methods** not to complain about missing required flavors, methods or instance variables. Presumably the flavors that depend on this one and actually are instantiated will supply what is lacking.

:method-combination

Specifies the method combination style to be used for certain operations. Each argument to this option is a list (*style order operation1 operation2...*). *operation1*, *operation2*, etc. are names of operations whose methods are to be combined in the declared fashion. *style* is a keyword that specifies a style of combination; see section 21.11, page 433. *order* is a keyword whose interpretation is up to *style*; 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 operation. If no component specifies a style of method combination, then the default style is used, namely **:daemon**. If more than one component of a flavor specifies the combination style for a given operation, then they must agree on the specification, or else an error is signaled.

:instance-area-function

The argument is the name of a function to be used when this flavor is instantiated, to determine which area to create the new instance in. Use a function name rather than an explicit lambda expression.

```
(:instance-area-function function-name)
```

When the instance area function is called, it is given the init plist as an argument, and should return an area number or **nil** to use the default. Init keyword values can be accessed using **get** on the init plist.

Instance area functions can be inherited from component flavors. If a flavor does not have or inherit an instance area function, its instances are created in **default-cons-area**.

:instantiation-flavor-function

You can define a flavor **foo** so that, when you try to instantiate it, it calls a function to decide what flavor it should really instantiate (not necessarily **foo**). This is done by giving **foo** an instantiation flavor function:

```
(:instantiation-flavor-function function-name)
```

When (**make-instance 'foo** *keyword-args...*) is done, the instantiation flavor function is called with two arguments: the flavor name specified (**foo** in this case) and the init plist (the list of keyword args). It should return the name of the flavor that should actually be instantiated.

Note that the instantiation flavor function applies only to the flavor it is specified for. It is not inherited by dependent flavors.

:run-time-alternatives
:mixture

A run-time-alternative flavor defines a collection of similar flavors, all built on the same base flavor but having various mixins as well. Instantiation chooses a flavor of the collection at run time based on the init keywords specified, using an automatically generated instantiation flavor function.

A simple example would be
```
(defflavor foo () (basic-foo)
  (:run-time-alternatives
     (:big big-foo-mixin))
  (:init-keywords :big))
```

Then (make-instance 'foo :big t) makes an instance of a flavor whose components are big-foo-mixin as well as foo. But (make-instance 'foo) or (make-instance 'foo :big nil) makes an instance of foo itself. The clause (:big big-foo-mixin) in the :run-time-alternatives says to incorporate big-foo-mixin if :big's value is t, but not if it is nil.

There may be several clauses in the :run-time-alternatives. Each one is processed independently. Thus, two keywords :big and :wide could independently control two mixins, giving four possibilities.
```
(defflavor foo () (basic-foo)
  (:run-time-alternatives
     (:big big-foo-mixin)
     (:wide wide-foo-mixin))
  (:init-keywords :big))
```

It is possible to test for values other than t and nil. The clause
```
(:size (:big big-foo-mixin)
       (:small small-foo-mixin)
       (nil nil))
```
allows the value for the keyword :size to be :big, :small or nil (or omitted). If it is nil or omitted, no mixin is used (that's what the second nil means). If it is :big or :small, an appropriate mixin is used. This kind of clause is distinguished from the simpler kind by having a list as its second element. The values to check for can be anything, but eq is used to compare them.

The value of one keyword can control the interpretation of others by nesting clauses within clauses. If an alternative has more than two elements, the additional elements are subclauses which are considered only if that alternative is selected. For example, the clause
```
(:etherial (t etherial-mixin)
           (nil nil
                (:size (:big big-foo-mixin)
                       (:small small-foo-mixin)
                       (nil nil))))
```
says to consider the :size keyword only if :etherial is nil.

:mixture is synonymous with :run-time-alternatives. It exists for compatibility with Symbolics systems.

:documentation
Specifies the documentation string for the flavor definition, which is made accessible

through (documentation *flavorname* 'flavor).

This documentation can be viewed with the describe-flavor function (see page 423) or the editor's Meta-X Describe Flavor command (see page 443).

Previously this option expected two arguments, a keyword and a string. The keyword was intended to classify the flavor as a base flavor, mixin or combination. But no way was found for this classification to serve a useful purpose. Keyword are still accepted but no longer recommended for use.

21.9 Flavor Families

The following organization conventions are recommended for programs that use flavors.

A *base flavor* is a flavor that defines a whole family of related flavors, all of which have that base flavor as a component. 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 operations. :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 should 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 may still define its own flavors, in a simple way. The facility provides a base flavor and a set of mixins: the caller can combine these in various ways depending on exactly what it wants, since the facility probably does 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 (see page 425) to select options of its component flavors and you can define one or two methods to customize it "just a little".

21.10 Vanilla Flavor

The operations 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 escape-p* *Operation*

The object should output its printed-representation to a stream. The printer sends this message when it encounters an instance or an entity. The arguments are the stream, the current depth in list-structure (for comparison with **prinlevel**), and whether escaping is enabled (a copy of the value of ***print-escape***; see page 514). **si: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 *Operation*

The object should describe itself, printing a description onto the ***standard-output*** stream. The **describe** function sends this message when it encounters an instance. **si:vanilla-flavor** outputs in a reasonable format the object, the name of its flavor, and the names and values of its instance-variables.

:set *keyword value* *Operation*

The object should set the internal value specified by *keyword* to the new value *value*. For flavor instances, the **:set** operation uses **:case** method combination, and a method is generated automatically to set each settable instance variable, with *keyword* being the variable's name as a keyword.

:which-operations *Operation*

The object should return a list of the operations it can handle. **si:vanilla-flavor** generates the list once per flavor and remembers it, minimizing consing and compute-time. If the set of operations handled is changed, this list is regenerated the next time someone asks for it.

:operation-handled-p *operation* *Operation*

operation is an operation name. The object should return **t** if it has a handler for the specified operation, **nil** if it does not.

:get-handler-for *operation* *Operation*

operation is an operation name. The object should return the method it uses to handle *operation*. If it has no handler for that operation, it should return **nil**. This is like the **get-handler-for** function (see page 422), but, of course, you can use it only on objects known to accept messages.

:send-if-handles *operation* &rest *arguments* *Operation*

operation is an operation name and *arguments* is a list of arguments for the operation. If the object handles the operation, it should send itself a message with that operation and arguments, and return whatever values that message returns. If it doesn't handle the operation it should just return **nil**.

:eval-inside-yourself *form* *Operation*

The argument is a form that 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 is modified. This is intended to be used mainly for debugging.

:funcall-inside-yourself *function* &rest *args* *Operation*

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 is modified. This is a way of allowing callers to provide actions to be performed in an environment set up by the instance.

:break *Operation*

break is called in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables.

21.11 Method Combination

When a flavor has or inherits more than one method for an operation, they must be called in a specific sequence. The flavor system creates a function called a *combined method* which calls all the user-specified methods in the proper order. Invocation of the operation actually calls the combined method, which is responsible for calling the others.

For example, if the flavor **foo** has components and methods as follows:

```
(defflavor foo () (foo-mixin foo-base))
(defflavor foo-mixin () (bar-mixin))

(defmethod (foo :before :hack) ...)
(defmethod (foo :after :hack) ...)

(defmethod (foo-mixin :before :hack) ...)
(defmethod (foo-mixin :after :hack) ...)

(defmethod (bar-mixin :before :hack) ...)
(defmethod (bar-mixin :hack) ...)

(defmethod (foo-base :hack) ...)
(defmethod (foo-base :after :hack) ...)
```

then the combined method generated looks like this (ignoring many important details not related to this issue):

```
(defmethod (foo :combined :hack) (&rest args)
  (apply #'(:method foo :before :hack) args)
  (apply #'(:method foo-mixin :before :hack) args)
  (apply #'(:method bar-mixin :before :hack) args)
  (multiple-value-prog1
      (apply #'(:method bar-mixin :hack) args)
    (apply #'(:method foo-base :after :hack) args)
    (apply #'(:method foo-mixin :after :hack) args)
    (apply #'(:method foo :after :hack) args)))
```

This example shows the default style of method combination, the one described in the introductory parts of this chapter, called :daemon combination. Each style of method combination defines which *method types* it allows, and what they mean. :daemon combination accepts method types :before and :after, in addition to *untyped* methods; then it creates a combined method which calls all the :before methods, only one of the untyped methods, and then all the :after methods, returning the value of the untyped method. The combined method is constructed by a function much like a macro's expander function, and the precise technique used to create the combined method is what gives :before and :after their meaning.

Note that the :before methods are called in the order foo, foo-mixin, bar-mixin and foo-base. (foo-base does not have a :before method, but if it had one that one would be last.) This is the standard ordering of the components of the flavor foo (see page 412); since it puts the base flavor last, it is called :base-flavor-last ordering. The :after methods are called in the opposite order, in which the base flavor comes first. This is called :base-flavor-first ordering.

Only one of the untyped methods is used; it is the one that comes first in :base-flavor-last ordering. An untyped method used in this way is called a *primary* method.

Other styles of method combination define their own method types and have their own ways of combining them. Use of another style of method combination is requested with the :method-combination option to defflavor (see page 429). Here is an example which uses :list method combination, a style of combination that allows :list methods and untyped methods:

```
(defflavor foo () (foo-mixin foo-base))
(defflavor foo-mixin () (bar-mixin))
(defflavor foo-base () ()
  (:method-combination (:list :base-flavor-last :win)))

(defmethod (foo :list :win) ...)
(defmethod (foo :win) ...)

(defmethod (foo-mixin :list :win) ...)

(defmethod (bar-mixin :list :win) ...)
(defmethod (bar-mixin :win) ...)

(defmethod (foo-base :win) ...)
```

yielding the combined method

```
(defmethod (foo :combined :win) (&rest args)
  (list
    (apply #'(:method foo :list :win) args)
    (apply #'(:method foo-mixin :list :win) args)
    (apply #'(:method bar-mixin :list :win) args)
    (apply #'(:method foo :win) args)
    (apply #'(:method bar-mixin :win) args)
    (apply #'(:method foo-base :win) args)))
```

The **:method-combination** option in the **defflavor** for **foo-base** causes **:list** method combination to be used for the **:win** operation on all flavors that have **foo-base** as a component, including **foo**. The result is a combined method which calls all the methods, including all the untyped methods rather than just one, and makes a list of the values they return. All the **:list** methods are called first, followed by all the untyped methods; and within each type, the **:base-flavor-last** ordering is used as specified. If the **:method-combination** option said **:base-flavor-first**, the relative order of the **:list** methods would be reversed, and so would the untyped methods, but the **:list** methods would still be called before the untyped ones. **:base-flavor-last** is more often right, since it means that **foo**'s own methods are called first and **si:vanilla-flavor**'s methods (if it has any) are called last.

A few specific method types, such as **:default** and **:around**, have standard meanings independent of the style of method combination, and can be used with any style. They are described in a table below.

Here are the standardly defined method combination styles.

:daemon The default style 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.

:daemon-with-or
Like the **:daemon** method combination style, except that the primary method is

wrapped in an :or special form with all :or methods. Multiple values can be returned from the primary method, but not from the :or methods (as in the **or** special form). This produces code like the following in combined methods:

```
(progn (foo-before-method)
       (multiple-value-prog1
         (or (foo-or-method)
             (foo-primary-method))
         (foo-after-method)))
```

This is useful primarily for flavors in which a mixin introduces an alternative to the primary method. Each :or method 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.

:daemon-with-and
: Like :daemon-with-or except that it combines :and methods in an **and** special form. The primary method is run only if all of the :and methods return non-nil values.

:daemon-with-override
: Like the :daemon method combination style, 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 run only if *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)))
```

:progn
: Calls all the methods inside a **progn** special form. Only untyped and :progn methods are allowed. The combined method calls all the :progn methods and then all the untyped methods. The result of the combined method is whatever the last of the methods returns.

:or
: Calls all the methods inside an **or** special form. This means that each of the methods is called in turn. Only untyped methods and :or methods are allowed; the :or methods are called first. 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 can return nil, and the next method gets a chance to try.

:and
: Calls all the methods inside an **and** special form. Only untyped methods and :and methods are allowed. The basic idea is much like :or; see above.

:append
: Calls all the methods and appends the values together. Only untyped methods and :append methods are allowed; the :append methods are called first.

:nconc — Calls all the methods and nconc's the values together. Only untyped methods and :nconc methods are allowed, etc.

:list — Calls all the methods and returns a list of their returned values. Only untyped methods and :list methods are allowed, etc.

:inverse-list — Calls each method with one argument; these arguments are successive elements of the list that is the sole argument to the operation. Returns no particular value. Only untyped methods and :inverse-list methods are allowed, etc.

If the result of a :list-combined operation is sent back with an :inverse-list-combined operation, with the same ordering and with corresponding method definitions, each component flavor receives the value that came from that flavor.

:pass-on — Calls each method on the values returned by the preceeding one. The values returned by the combined method are those of the outermost call. 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.

Only untyped methods and :pass-on methods are allowed. The :pass-on methods are called first.

:case — With :case method combination, the combined method automatically does a selectq dispatch on the first argument of the operation, known as the *suboperation*. Methods of type :case can be used, and each one specifies one suboperation that it applies to. If no :case method matches the suboperation, the primary method, if any, is called.

Example:

```
(defflavor foo (a b) ()
  (:method-combination (:case :base-flavor-last :win)))
```

This method handles (send a-foo :win :a):

```
(defmethod (foo :case :win :a) ()
  a)
```

This method handles (send a-foo :win :a*b):

```
(defmethod (foo :case :win :a*b) ()
  (* a b))
```

This method handles (send a-foo :win :something-else):

```
(defmethod (foo :win) (suboperation)
  (list 'something-random suboperation))
```

:case methods are unusual in that one flavor can have many :case methods for the same operation, as long as they are for different suboperations.

The suboperations :which-operations, :operation-handled-p, :send-if-handles and :get-handler-for are all handled automatically based on the collection of :case methods that are present.

Methods of type :or are also allowed. They are called just before the primary method, and if one of them returns a non-nil value, that is the value of the operation, and no more methods are called.

Here is a table of all the method types recognized by the standard styles 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	Used for the before-daemon and after-daemon methods used by :daemon 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 (see page 431) defines some default methods for certain of the operations understood by its family. When using the default kind of method combination these default methods are suppressed if another component provides a primary method.
:or :and	Used for :daemon-with-or and :daemon-with-and method combination. The :or methods are wrapped in an or, or the :and methods are wrapped in an and, together with the primary method, between the :before and :after methods.
:override	Allows the features of :or method combination to be used together with daemons. If you specify :daemon-with-override method combination, you may use :override methods. The :override methods are executed first, until one of them returns non-nil. If this happens, that method's value(s) are returned and no more methods are used. If all the :override methods return nil, the :before, primary and :after methods are executed as usual. 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 operation.
:or, :and, :progn, :list, :inverse-list, pass-on, :append, :nconc.	Each of these methods types is allowed in the method combination style of the same name. In those method combination styles, these typed methods work just like untyped ones, but all the typed methods are called before all the untyped ones.
:case	:case methods are used by :case method combination.

These method types can be used with any method combination style; they have standard meanings independent of the method combination style being used.

:around An :around method is able to control when, whether and how the remaining methods are executed. It is given a continuation that is a function that will execute the remaining methods, and has complete responsibility for calling it or not, and deciding what arguments to give it. For the simplest behavior, the arguments should be the operation name and operation arguments that the :around method itself received; but sometimes the whole purpose of the :around method is to modify the arguments before the remaining methods see them.

The :around method receives three special arguments before the arguments of the operation itself: the *continuation*, the *mapping-table*, and the *original-argument-list*. The last is a list of the operation name and operation arguments. The simplest way for the :around method to invoke the remaining methods is to do

```
(lexpr-funcall-with-mapping-table
  continuation mapping-table
  original-argument-list)
```

In general, the *continuation* should be called with either funcall-with-mapping-table or lexpr-funcall-with-mapping-table, providing the *continuation*, the *mapping-table*, and the operation name (which you know because it is the same as in the defmethod), followed by whatever arguments the remaining methods are supposed to see.

```
(defflavor foo-one-bigger-mixin () ())

(defmethod (foo-one-bigger-mixin :around :set-foo)
           (cont mt ignore new-foo)
  (funcall-with-mapping-table cont mt :set-foo
                              (1+ new-foo)))
```

is a mixin which modifies the :set-foo operation so that the value actually used in it is one greater than the value specified in the message.

:inverse-around
:inverse-around methods work like :around methods, but they are invoked at a different time and in a different order.

With :around methods, those of earlier flavor components components are invoked first, starting with the instantiated flavor itself, and those of earlier components are invoked within them. :inverse-around methods are invoked in the opposite order: si:vanilla-flavor would come first. Also, all :around methods and wrappers are invoked inside all the :inverse-around methods.

For example, the :inverse-around :init method for tv:sheet (a base flavor for all window flavors) is used to handle the init keywords :expose-p and :activate-p, which cannot be handled correctly until the window is entirely set up. They are handled in this method because it is guaranteed to be the first method invoked by the :init operation on any flavor of window (because no component of tv:sheet defines an :inverse-around method for this operation). All the rest of the work of making a new window valid takes place in this method's continuation; when the continuation returns, the window must be as valid as it will ever be, and it is ready to be exposed or activated.

:wrapper Used internally by **defwrapper**.

Note that if one flavor defines both a wrapper and an :around method for the same operation, the :around method is executed inside the wrapper.

: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 instantiated flavor is called, then :before daemons of successively more basic flavors 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 base flavor is called, followed by the :after daemons at successively less basic flavors.

Now, if there is no interaction among all these methods, if their actions are completely independent, then it doesn't matter whether you use a :before daemon or an :after daemon. There is 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 operation, 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 operation 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.

The problems become most difficult when you are creating a network of instances of various flavors that are supposed to point to each other. For example, suppose you have flavors for "buffers" and "streams", and each buffer should be accompanied by a stream. If you create the stream in the :before :init method for buffers, you can inform the stream of its corresponding buffer with an init keyword, but the stream may try sending messages back to the buffer, which is not yet ready to be used. If you create the stream in the :after :init method for buffers, there will be no problem with stream creation, but some other :after :init methods of other mixins may have run and made the assumption that there is to be no stream. The only way to guarantee success is to create the stream in a :before method and inform it of its associated buffer by sending it a message from the buffer's :after :init method. This scheme—creating associated objects in :before methods but linking them up in :after methods—often avoids problems, because all the various associated objects used by various mixins at least exist when it is time to make other objects point to them.

Since flavors are not hierarchically organized, the notion of levels of abstraction is not rigidly applicable. However, it remains a useful way of thinking about systems.

21.12 Implementation of Flavors

An object that is an instance of a flavor is implemented using the data type dtp-instance. The representation is a structure whose first word, tagged with dtp-instance-header, 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 that 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 operations, 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 operation. The microcode binds self to the object and binds those instance variables that are supposed to be special to the value cells in the instance. Then it passes on the operation and arguments to a funcallable hash table taken from the flavor-structure for this flavor.

When the funcallable hash table is called as a function, it hashes the first argument (the operation) to find a function to handle the operation and an array called a mapping table. The variable sys:self-mapping-table is bound to the mapping table, which tells the microcode how to access the lexical instance variables, those not defined to be special. Then the function is called. If there is only one method to be invoked, this function is that method; otherwise it is an automatically-generated function called the combined method (see page 413), which calls the appropriate methods in the right order. If there are wrappers, they are incorporated into this combined method.

The mapping table is an array whose elements correspond to the instance variables which can be accessed by the flavor to which the currently executing method belongs. Each element contains the position in self of that instance variable. This position varies with the other instance variables and component flavors of the flavor of self.

Each time the combined method calls another method, it sets up the mapping table required by that method—not in general the same one which the combined method itself uses. The mapping tables for the called methods are extracted from the array leader of the mapping table used by the combined method, which is kept in a local variable of the combined method's stack frame while sys:self-mapping-table is set to the mapping tables for the component methods.

sys:self-mapping-table *Variable*

Holds the current mapping table, which tells the running flavor method where in **self** to find each instance variable.

Ordered instance variables are referred to directly without going through the mapping table. This is a little faster, and reduces the amount of space needed for mapping tables. It is also the reason why compiled code contains the positions of the ordered instance variables and must be recompiled when they change.

21.12.1 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 may produce a warning that an instance variable was declared special (because the system did not realize it was an instance variable). If this happens, you should fix the problem and recompile.

The methods automatically generated by the **:gettable-instance-variables** and **:settable-instance-variables defflavor** options (see page 424) are generated at the time the **defflavor** is done.

The first time a flavor is instantiated, or when **compile-flavor-methods** is done, the system looks through all of the component flavors and gathers various information. At this point an error is signaled 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 defflavor** option, page 425). The combined methods (see page 413) are generated at this time also, unless they already exist.

After a flavor has been instantiated, it is possible to make changes to it. Such changes affect all existing instances if possible. This is described more fully immediately below.

21.12.2 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, operation (and suboperation if any), and (optional) method-type as an existing method, that method is replaced by the new definition. You can remove a method with undefmethod (see page 419).

These changes always propagate to all flavors that depend upon the changed flavor. Normally the system propagates the changes to all existing instances of the changed flavor and its dependent flavors. 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 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.

21.13 Useful Editor Commands

This section briefly documents some editor commands that are useful in conjunction with flavors.

Meta-.

The Meta-. (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 it a suitable function spec starting with :method, such as (:method tv:sheet :expose). The keyword :method may be omitted if the definition is in the editor already. Completion is available on the flavor name and operation name, as usual only for definitions loaded into the editor.

Meta-X Describe Flavor

Asks for a flavor name in the mini-buffer 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 editor commands to apply to the name and the left-hand mouse button does one of them, typically positioning the editor to the source code for that name.

Meta-X List Methods
Meta-X Edit Methods

Asks you for an operation in the mini-buffer and lists all the flavors that have a method for that operation. You may type in the operation name, point to it with the mouse, or let it default to the operation of the message being sent by the Lisp form the cursor is on. List Methods produces a mouse-sensitive display allowing you to edit selected methods or just to 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 editor command Control-Shift-P advances the editor cursor to the next method in the list, reading in its source file if necessary. Typing Control-Shift-P, while the display is on the screen, edits the first method.

In addition, you can find a copy of the list in the editor buffer *Possibilities*. While in that buffer, the command Control-/ visits the definition of the method described on the line the cursor is pointing at.

These techniques of moving through the objects listed apply to all the following commands as well.

Meta-X List Combined Methods
Meta-X Edit Combined Methods
Asks you for an operation name and a flavor in two mini-buffers and lists all the methods that would be called to handle that operation for an instance of that flavor.

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-Shift-P to get to successive ones.

Meta-X List Flavor Components
Meta-X Edit Flavor Components
Asks you for a flavor and lists or begins visiting all the flavors it depends on.

Meta-X List Flavor Dependents
Meta-X Edit Flavor Dependents
Asks you for a flavor and lists or begins visiting all the flavors that depend on it.

Meta-X List Flavor Direct Dependents
Meta-X Edit Flavor Direct Dependents
Asks you for a flavor and lists or begins visiting all the flavors that depend directly on it.

Meta-X List Flavor Methods
Meta-X Edit Flavor Methods
Asks you for a flavor and lists or begins visiting all the methods defined for that flavor. (This does not include methods inherited from its component flavors.)

21.14 Property List Operations

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. For more details and examples, see the general discussion of property lists (section 5.10, page 113). The usual property list functions (**get**, **putprop**, etc.) all work on instances by sending the instance the corresponding message.

si:property-list-mixin *Flavor*

This mixin flavor provides the basic operations on property lists.

:get *property-name* &optional *default* *Operation on* si:property-list-mixin

Looks up the object's *property-name* property. If it finds such a property, it returns the value; otherwise it returns *default*.

:getl *property-name-list* *Operation on* si:property-list-mixin

Like the :get operation, except that the argument is a list of property names. The :getl operation searches down the property list until it finds a property whose property name is one of the elements of *property-name-list*. It returns the portion of the property list begining with the first such property that it found. If it doesn't find any, it returns **nil**.

:putprop *value property-name* *Operation on* si:property-list-mixin

Gives the object an *property-name* property of *value*.

(send *object* :set :get *property-name value*)

also has this effect.

:remprop *property-name* *Operation on* si:property-list-mixin

Removes the object's *property-name* property, by splicing it out of the property list. It returns one of the cells spliced out, whose car is the former value of the property that was just removed. If there was no such property to begin with, the value is **nil**.

:get-location-or-nil *property-name* *Operation on* si:property-list-mixin
:get-location *property-name* *Operation on* si:property-list-mixin

Both return a locative pointer to the cell in which this object's *property-name* property is stored. If there is no such property, :get-location-or-nil returns **nil**, but :get-location adds a cell to the property list and initialized to **nil**, and a pointer to that cell is returned.

:push-property *value property-name* *Operation on* si:property-list-mixin

The *property-name* property of the object should be a list (note that **nil** is a list and an absent property is **nil**). This operation sets the *property-name* property of the object to a list whose car is *value* and whose cdr is the former *property-name* property of the list. This is analogous to doing

(push *value* (get *object property-name*))

See the **push** special form (page 88).

:property-list *Operation on* si:property-list-mixin
Returns the list of alternating property names and values that implements the property list.

:property-list-location *Operation on* si:property-list-mixin
Returns a locative pointer to the cell in the instance which holds the property list data.

:set-property-list *list* *Operation on* si:property-list-mixin
Sets the list of alternating property names and values that implements the property list to *list*. So does

```
        (send object :set :property-list list)
```

:property-list *list* *Init option for* si:property-list-mixin
This initializes the list of alternating property names and values that implements the property list to *list*.

21.15 Printing Flavor Instances Readably

A flavor instance can print out so that it can be read back in, as long as you give it a :print-self method that produces a suitable printed representation, and provide a way to parse it. The convention for doing this is to print as

#⊂*flavor-name additional-data*⊃

and make sure that the flavor defines or inherits a :read-instance method that can parse the *additional-data* and return an instance (see page 527). A convenient way of doing this is to use si:print-readably-mixin.

si:print-readably-mixin *Flavor*
Provides for flavor instances to print out using the #⊂ syntax, and also for reading things that were printed in that way.

:reconstruction-init-plist *Operation on* si:print-readably-mixin
When you use si:print-readably-mixin, you must define the operation :reconstruction-init-plist. This should return an alternating list of init options and values that could be passed to make-instance to create an instance "like" this one. Sufficient similarity is defined by the practical purposes of the flavor's implementor.

21.16 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 putprop or remprop on one of the instances does not affect the properties of the other

instance.

* 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. Should the copy talk to the same agent, or should a new agent be constructed for it?
* 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, as in (send object :copy). It is up to you to implement the operation in a suitable fashion, such as

```
(defflavor foo (a b c) ()
  (:inittable-instance-variables a b))

(defmethod (foo :copy) ()
  (make-instance 'foo :a a :b b))
```

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.

If a flavor supports the :reconstruction-init-plist operation, a suitable copy can be made by invoking this operation and passing the result to make-instance along with the flavor name. This is because the definition of what the :reconstruction-init-plist operation should do requires it to address all the problems listed above. Implementing this operation is up to you, and so is making sure that the flavor implements sufficient init keywords to transmit any information that is to be copied. See page 446.

22. The I/O System

Zetalisp provides a powerful and flexible system for performing input and output to peripheral devices. Device independent I/O is generalized in the concept of an *I/O stream*. A stream is a source or sink for data in the form of characters or integers; sources are called *input streams* and sinks are called *output streams*. A stream may be capable of use in either direction, in which case it is a *bidirectional* stream. In a few unusual cases, it is useful to have a 'stream' which supports neither input nor output; for example, opening a file with direction :probe returns one (page 583). Streams on which characters are transferred are called *character streams*, and are used more often than *binary streams*, which usually transfer integers of type (unsigned-byte *n*) for some *n*.

Streams automatically provide a modular separation between the program which implements the stream and the program which uses it, because streams obey a standard protocol. The stream protocol is a special case is based on the general message passing protocol: a stream operation is invoked by calling the stream as a function, with a first argument that is a keyword and identifies the I/O operation desired (such as, :tyi to read a character) and additional arguments as that operation calls for them. The stream protocol consists of a particular set of operation names and calling conventions for them. It is documented in section 22.3, page 459.

Many programs do not invoke the stream operations directly; instead, they call standard I/O functions which then invoke stream operations. This is done for two reasons: the functions may provide useful services, and they may be transportable to Common Lisp or Maclisp. Programs that use stream operations directly are not transportable outside Zetalisp. The I/O functions are documented in the first sections of this chapter.

The generality of the Zetalisp I/O stream comes from the fact that I/O operations on it can invoke arbitrary Lisp code. For example, it would be very simple to implement a "morse code" stream that accepted character output and used beep with appropriate pauses to 'display' it. How to implement a stream is documented in section 22.3.12, page 474, and the following sections.

The most commonly used streams are windows, which read input from the keyboard and dispose of output by drawing on the screen, file streams, editor buffer streams which get input from the text in a buffer and insert output into the buffer, and string streams which do likewise with the contents of a string.

Another unusual aspect of Lisp I/O is the ability to input and output general Lisp objects, represented as text. These are done using the read and related functions and using print and related functions. They are documented in chapter 23.

22.1 Input Functions

The input functions read characters, lines, or bytes from an input stream. This argument is called *stream*. If omitted or nil, the current value of *standard-input*. This is the "default input stream", which in simple use reads from the terminal keyboard. If the argument is t, the current value of *terminal-io* is used; this is conventionally supposed to access "the user's terminal" and nearly always reads from the keyboard in processes belonging to windows.

If the stream is an interactive one, such as the terminal, the input is echoed, and functions which read more than a single character allow editing as well. peek-char echoes all of the characters that were skipped over if read-char would have echoed them; the character not removed from the stream is not echoed either.

When an input stream has no more data to return, it reports end of file. Each stream input operation has a convention for how to do this. The input functions accept an argument *eof-option* or two arguments *eof-error* and *eof-value* to tell them what to do if end of file is encountered instead of any input. The functions that take two *eof-* arguments are the Common Lisp ones. For them, end of file is an error if *eof-error* is non-nil or if it is unsupplied. If *eof-error* is nil, then the function returns *eof-value* at end of file.

The functions which have one argument called *eof-option* are from Maclisp. End of file causes an error if the argument is not supplied. Otherwise, end of file causes the function to return the argument's value. Note that an *eof-option* of nil means to return nil if the end of the file is reached; it is *not* equivalent to supplying no *eof-option*.

sys:end-of-file (error) *Condition*

All errors signaled to report end of file possess this condition name.

The :stream operation on the condition instance returns the stream on which end of file was reached.

22.1.1 String Input Functions

read-line &optional *stream* (*eof-errorp* t) *eof-value ignore options*

Reads a line of text, terminated by a Return. It returns the line as a character string, *without* the Return character that ended the line. The argument *ignore* must be accepted for the sake of the Common Lisp specifications but it is not used.

This function is usually used to get a line of input from the user. If rubout processing is happening, then *options* is passed as the list of options to the rubout handler (see section 22.5, page 500).

There is a second value, t if the line was terminated by end of file.

readline &optional *stream eof-option options*

Like read-line but uses the Maclisp convention for specifying what to do about end of file. This function can take its first two arguments in the other order, for Maclisp compatibility only; see the note in section 22.1.3, page 451.

readline-trim &optional *stream eof-option options*

This is like readline except that leading and trailing spaces and tabs are discarded from the value before it is returned.

readline-or-nil &optional *stream eof-option options*

Like readline-trim except that nil is returned if the line is empty or all blank.

read-delimited-string &optional *delimiter stream eof rubout-handler-options buffer-size*

Reads input from *stream* until a delimiter character is reached, then returns as a string all the input up to but not including the delimiter. *delimiter* is either a character or a list of characters which all serve as delimiters. It defaults to the character End. *stream* defaults to the value of *standard-input*.

If *eof* is non-nil, then end of file on attempting to read the first character is an error. Otherwise it just causes an empty string to be returned. End of file once at least one character has been read is never an error but it does cause the function to return all the input so far.

Input is done using rubout handling and echoing if stream supports the :rubout-handler operation. In this case, *rubout-handler-options* are passed as the options argument to that operation.

buffer-size specifies the size of string buffer to allocate initially.

The second value returned is t if input ended due to end of file.

The third value is the delimiter character which terminated input, or nil if input terminated due to end of file. This character is currently represented as a fixnum, but perhaps someday will be a character object instead.

22.1.2 Character-Level Input Functions

read-char &optional *stream* (*eof-errorp* t) *eof-value*

Reads a character from *stream* and returns it as a character object. End of file is an error if *eof-errorp* is non-nil; otherwise, it causes read-char to return *eof-value*. This uses the :tyi stream operation.

read-byte *stream* &optional (*eof-errorp* t) *eof-value*

Like read-char but returns an integer rather than a character object. In strict Common Lisp, only read-char can be used on character streams and only read-byte can be used on binary streams.

read-char-no-hang &optional *stream* (*eof-errorp* t) *eof-value*
Similar but returns nil immediately when no input is available on an interactive stream. Uses the :tyi-no-hang stream operation (page 466).

unread-char *char* &optional *stream*
Puts *char* back into *stream* so that it will be read again as the next input character. *char* must be the same character that was read from stream most recently. It may not work to unread two characters in a row before reading again. Uses the :untyi stream operation (page 461).

peek-char *peek-type* &optional *stream* (*eof-errorp* t) *eof-value*
If *peek-type* is nil, this is like read-char except leaves the character to be read again by the next input operation.

If *peek-type* is t, skips whitespace characters and peeks at the first nonwhitespace character. That character is the value, and is also left to be reread.

If *peek-type* is a character, reads input until that character is seen. That character is unread and also returned.

listen &optional *stream*
t if input is now available on *stream*. Uses the :listen operation (page 466).

clear-input &optional *stream*
Discards any input now available on *stream*, if it is an interactive stream. Uses the :clear-input stream operation (page 469).

22.1.3 Maclisp Compatibility Input Functions

These functions accept an argument *eof-option* to tell them what to do if end of file is encountered instead of any input. End of file signals an error if the argument is not supplied. Otherwise, end of file causes the function to return the argument's value. Note that an *eof-option* of nil means to return nil if the end of the file is reached; it is *not* equivalent to supplying no *eof-option*.

The arguments *stream* and *eof-option* can also be given in the reverse order for compatibility with old Maclisp programs. The functions attempt to figure out which way they were called by seeing whether each argument is a plausible stream. Unfortunately, there is an ambiguity with symbols: a symbol might be a stream and it might be an eof-option. If there are two arguments, one being a symbol and the other being something that is a valid stream, or only one argument, which is a symbol, then these functions interpret the symbol as an eof-option instead of as a stream. To force them to interpret a symbol as a stream, give the symbol an si:io-stream-p property whose value is t.

tyi &optional *stream eof-option*

Reads one character from *stream* and returns it. The character is echoed if *stream* is interactive, except that **Rubout** is not echoed. The **Control**, **Meta**, etc. shifts echo as **C-**, **M-**, etc.

The :tyi stream operation is preferred over the tyi function for some purposes. Note that it does not echo. See page 461.

(This function can take its arguments in the other order, for Maclisp compatibility only; see the note above.)

readch &optional *stream eof-option*

Like tyi except that instead of returning a fixnum character, it returns a symbol whose print name is the character. The symbol is interned in the current package. This is just Maclisp's version of character object. (This function can take its arguments in the other order, for Maclisp compatibility only; see the note above.)

This function is provided only for Maclisp compatibility, since in Zetalisp never uses symbols to represent characters in this way.

tyipeek &optional *peek-type stream eof-option*

This function is provided mainly for Maclisp compatibility; the :tyipeek stream operation is usually clearer (see page 461).

What tyipeek does depends on the *peek-type*, which defaults to nil. With a *peek-type* of nil, tyipeek returns the next character to be read from *stream*, without actually removing it from the input stream. The next time input is done from *stream* the character will still be there; in general, (= (tyipeek) (tyi)) is t.

If *peek-type* is a fixnum less than 1000 octal, then tyipeek reads characters from *stream* until it gets one equal to *peek-type*. That character is not removed from the input stream.

If *peek-type* is t, then tyipeek skips over input characters until the start of the printed representation of a Lisp object is reached. As above, the last character (the one that starts an object) is not removed from the input stream.

The form of tyipeek supported by Maclisp in which *peek-type* is a fixnum not less than 1000 octal is not supported, since the readtable formats of the Maclisp reader and the Zetalisp reader are quite different.

Characters passed over by tyipeek are echoed if *stream* is interactive.

22.1.4 Interactive Input with Prompting

prompt-and-read *type-of-parsing format-string* &rest *format-args*

Reads some sort of object from ***query-io***, parsing it according to *type-of-parsing*, and prompting by calling **format** using *format-string* and *format-args*.

type-of-parsing is either a keyword or a list starting with a keyword and continuing with a list of options and values, whose meanings depend on the keyword used.

Most keywords specify reading a line of input and parsing it in some way. The line can be terminated with **Return** or **End**. Sometimes typing just **End** has a special meaning.

The keywords defined are

:eval-sexp
:eval-form This keyword directs **prompt-and-read** to accept a Lisp expression. It is evaluated, and the value is returned by **prompt-and-read**.

If the Lisp expression is not a constant or quoted, the user is asked to confirm the value it evaluated to.

A default value can be specified with an option, as in

```
(:eval-sexp :default default)
```

Then, if the user types **Space**, **prompt-and-read** returns the *default* as the first value and **:default** as the second value.

:eval-sexp-or-end
:eval-form-or-end
Synonymously direct **prompt-and-read** to accept a Lisp expression or just the character **End**. If **End** is typed, **prompt-and-read** returns **nil** as its first value and **:end** as its second value. Otherwise, things proceed as for **:eval-sexp**.

A default value is allowed, as in **:eval-sexp**.

:read
:expression Synonymously direct **prompt-and-read** to read an object and return it, with no evaluation.

:expression-or-end
Is like **:expression** except that the user is also allowed to type just **End**. If he does so, **prompt-and-read** returns the two values **nil** and **:end**.

:number Directs **prompt-and-read** to read and return a number. It insists on getting a number, forcing the user to rub out anything else. Additional features can be specified with options:

```
(:number :input-radix radix :or-nil nil-ok-flag)
```

parses the number using radix *radix* if the number is a rational. (By default, the ambient radix is used). If *nil-ok-flag* is non-nil, then the user is also permitted to type just **Return** or **End**, and then **nil** is returned.

:decimal-number
:number-or-nil
:decimal-number-or-nil
Abbreviations for
```
(:number :input-radix 10)
(:number :or-nil t)
(:number :input-radix 10 :or-nil t)
```

:date Directs prompt-and-read to read a date and time, terminated with *Return* or End, and return it as a universal time (see page 777). It allows several options:
(:date :never-p *never-ok* :past-p *past-required*)
If *past-required* is non-nil, the date must be before the present time, or the user must rub out and use a different date. If *never-ok* is non-nil, the user may also type "never"; then nil is returned.

:date-or-never
:past-date
:past-date-or-never
Abbreviations for
```
(:date :never-p t)
(:date :past-p t)
(:date :never-p t :past-p t)
```

:character Directs prompt-and-read to read a single character and return a character object representing it.

:string Directs prompt-and-read to read a line and return its contents as a string, using **readline**.

:string-or-nil Directs prompt-and-read to read a line and return its contents as a string, using readline-trim. In addition, if the result would be empty, nil is returned instead of the empty string.

:string-list Like :string-trim but regards the line as a sequence of input strings separated by commas. Each substring between commas is trimmed, and a list of the strings is returned.

:keyword-list Like :string-list but converts each string to a keyword by interning it in the keyword package. The value is therefore a list of keywords.

:font-list Like :string-list but converts each string to a font name by interning it in the fonts package. The symbols must already exist in that package or the user is required to retype the input.

:delimited-string
Directs prompt-and-read to read a string terminated by specified delimiters. With
(:delimited-string :delimiter *delimiter-list*
:buffer-size *size*)
you can specify a list of delimiter characters and an initial size for the buffer. The list defaults to (#\end) and the size to 100.

The work is done by read-delimited-string (page 450). The delimiters and size are passed to that function.

:delimited-string-or-nil
: Like :delimited-string except that nil is returned instead of the empty string if the first character read is a delimiter.

:host
: Directs prompt-and-read to read a line and interpret the contents as a network host name. The value returned is the host, looked up using si:parse-host (page 576). An option is defined:

```
(:host :default default-name :chaos-only chaos-only)
```

If the line read is empty, the host named *default-name* is used. If *chaos-only* is non-nil, only hosts on the Chaosnet are permitted input.

:host-list
: Like :host but regards the line as a sequence of host names separated by commas. Each host name is looked up as in :host and a list of the resulting hosts is returned.

:pathname-host
: Like :host but uses fs:get-pathname-host to look up the host object from its name (page 577). Thus, you find hosts that can appear in pathnames rather than hosts that are on the network.

:pathname
: Directs prompt-and-read to read a line and parse it as a pathname, merging it with the defaults. If the line is empty, the default pathname is used. These options are defined:

```
(:pathname :defaults defaults-alist-or-pathname
           :version default-version)
```

uses *defaults-alist-or-pathname* as the *defaults* argument to fs:merge-pathname-defaults, and *default-version* as the *version* argument to it.

:pathname-or-nil
: Is like :pathname, but if the user types just End it is interpreted as meaning "no pathname" rather than "use the default". Then nil is returned.

:pathname-list
: Like :pathname but regards the line as a sequence of filenames separated by commas. Each filename is parsed and defaulted and a list of the resulting pathnames is returned.

:fquery
: Directs prompt-and-read to query the user for a fixed set of alternatives, using fquery. *type-of-parsing* should always be a list, whose car is :fquery and whose cdr is a list to be passed as the list of options (fquery's first argument).

Example:

```
(prompt-and-read '(:fquery
                   . ,format:y-or-p-options)
                 "Eat it? ")
```

is equivalent to

```
(y-or-n-p "Eat it? ")
```

This keyword is most useful as a way to get to **fquery** when going through an interface defined to call **prompt-and-read**.

22.2 Output Functions

These functions all take an optional argument called *stream*, which is where to send the output. If unsupplied *stream* defaults to the value of ***standard-output***. If *stream* is **nil**, the value of ***standard-output*** (i.e. the default) is used. If it is **t**, the value of ***terminal-io*** is used (i.e. the interactive terminal). This is all more-or-less compatible with Maclisp, except that instead of the variable ***standard-output*** Maclisp has several variables and complicated rules. For detailed documentation of streams, refer to section 22.3, page 459.

For **print** and the other expression output functions, see section 23.4, page 527.

write-char *char* &optional *stream*
tyo *char* &optional *stream*

Outputs *char* to *stream* (using **:tyo**). *char* may be an integer or a character object; in the latter case, it is converted to an integer before the **:tyo**.

write-byte *number* &optional *stream*

Outputs number to stream using **:tyo**. In strict Common Lisp, output to binary streams can be done only with **write-byte** and output to character streams requires **write-char**. In fact, the two functions are identical on the Lisp Machine.

write-string *string* &optional *stream* &key (*start* **0**) *end*

Outputs *string* (or the specified portion of it) to *stream*.

write-line *string* &optional *stream* &key (*start* **0**) *end*

Outputs *string* (or the specified portion) to *stream*, followed by a **Return** character.

fresh-line &optional *stream*

Outputs a **Return** character to stream unless either

(1) nothing has been output to *stream* yet, or

(2) the last thing output was a **Return** character, or

(3) *stream* does not remember what previous output there has been.

This uses the **:fresh-line** stream operation. The value is **t** if a **Return** is output, **nil** if nothing is output.

force-output &optional *stream*

Causes *stream*'s buffered output, if any, to be transmitted immediately. This uses the **:force-output** stream operation.

finish-output &optional *stream*

Causes *stream*'s buffered output, if any, to be transmitted immediately, and waits until that is finished. This uses the :finish stream operation.

clear-output &optional *stream*

Discards any output buffered in *stream*. This uses the :clear-output stream operation.

terpri &optional *stream*

Outputs a Return character to *stream*. It returns t for Maclisp compatibility. It is wise not to depend on the value terpri returns.

cli:terpri &optional *stream*

Outputs a Return character to *stream*. Returns nil to meet Common Lisp specifications. It is wise not to depend on the value cli:terpri returns.

The format function (see page 483) is very useful for producing nicely formatted text. It can do anything any of the above functions can do, and it makes it easy to produce good looking messages and such. format can generate a string or output to a stream.

stream-copy-until-eof *from-stream to-stream* &optional *leader-size*

stream-copy-until-eof inputs characters from *from-stream* and outputs them to *to-stream*, until it reaches the end of file on the *from-stream*. For example, if x is bound to a stream for a file opened for input, then (stream-copy-until-eof x *terminal-io*) prints the file on the console.

If *from-stream* supports the :line-in operation and *to-stream* supports the :line-out operation, then stream-copy-until-eof uses those operations instead of :tyi and :tyo, for greater efficiency. *leader-size* is passed as the argument to the :line-in operation.

beep &optional *beep-type* (*stream* *terminal-io*)

This function is intended to attract the user's attention by causing an audible beep, or flashing the screen, or something similar. If the stream supports the :beep operation, then this function sends it a :beep message, passing *beep-type* along as an argument. Otherwise it just causes an audible beep on the terminal.

beep-type is a keyword which explains the significance of this beep. Users can redefine beep to make different noises depending on the beep type. The defined beep types are:

zwei:converse-problem	Used for the beep that is done when Converse is unable to send a message.
zwei:converse-message-received	Used for the beeps done when a Converse message is received.
zwei:no-completion	Used when you ask for completion in the editor and the string does not complete.
tv:notify	Used for the beep done when you get a notification that cannot be printed on the selected window.

fquery	Used for the beep done by yes-or-no-p or by fquery with the :beep option specified.
supdup:terminal-bell	Used for the beep requested by the remote host being used through a Supdup window.
nil	Used whenever no other beep type applies.

The :beep operation is described on page 467.

cursorpos &rest *args*

This function exists primarily for Maclisp compatibility. Usually it is preferable to send the appropriate messages (see the Window System manual).

cursorpos normally operates on the *standard-output* stream; however, if the last argument is a stream or t (meaning *terminal-io*) then cursorpos uses that stream and ignores it when doing the operations described below. cursorpos only works on streams that are capable of these operations, such as windows. A stream is taken to be any argument that is not a number and not a symbol, or that is a symbol other than nil with a name more than one character long.

(cursorpos) => (*line* . *column*), the current cursor position.

(cursorpos *line column*) moves the cursor to that position. It returns t if it succeeds and nil if it doesn't.

(cursorpos *op*) performs a special operation coded by *op*, and returns t if it succeeds and nil if it doesn't. *op* is tested by string comparison, it is not a keyword symbol and may be in any package.

f	Moves one space to the right.
b	Moves one space to the left.
d	Moves one line down.
u	Moves one line up.
t	Homes up (moves to the top left corner). Note that t as the last argument to cursorpos is interpreted as a stream, so a stream *must* be specified if the t operation is used.
z	Home down (moves to the bottom left corner).
a	Advances to a fresh line. See the :fresh-line stream operation.
c	Clears the window.
e	Clear from the cursor to the end of the window.
l	Clear from the cursor to the end of the line.
k	Clear the character position at the cursor.
x	b then k.

22.3 I/O Streams

An *I/O stream*, or just *stream*, is a source and/or sink of characters or bytes. A set of *operations* is available with every stream; operations include things like "output a character" and "input a character". The way to perform an operation on a stream is the same for all streams, although what happens inside the stream is very different depending on what kind of a stream it is. So all a program has to know is how to deal with streams using the standard, generic operations. A programmer creating a new kind of stream only needs to implement the appropriate standard operations.

A stream is a message-receiving object. This means that it is something that you can apply to arguments. The first argument is a keyword symbol which is the name of the operation you wish to perform. The rest of the arguments depend on what operation you are doing. Message-passing and generic operations are explained in the flavor chapter (chapter 21, page 401).

Some streams can only do input, some can only do output, and some can do both. Some operations are only supported by some streams. Also, there are some operations that the stream may not support by itself, but which work anyway, albeit slowly, because the *stream default handler* can handle them. All streams support the operation :which-operations, which returns a list of the names of all of the operations that are supported "natively" by the stream. (:which-operations itself is not in the list.)

All input streams support all the standard input operations, and all output streams support all the standard output operations. All bidirectional streams support both.

streamp *object*

According to Common Lisp, this returns t if *object* is a stream. In the Lisp machine, a stream is any object which can be called as a function with certain calling conventions. It is theoretically impossible to test for this. However, streamp does return t for any of the usual types of streams, and nil for any Common Lisp datum which is not a stream.

22.3.1 Standard Streams

There are several variables whose values are streams used by many functions in the Lisp system. These variables and their uses are listed here. By convention, variables that are expected to hold a stream capable of input have names ending with -input, and similarly for output. Those expected to hold a bidirectional stream have names ending with -io. The names with asterisks are synonyms introduced for the sake of Common Lisp.

standard-input *Variable*
standard-input *Variable*

In the normal Lisp top-level loop, input is read from *standard-input* (that is, whatever stream is the value of *standard-input*). Many input functions, including tyi and read, take a stream argument that defaults to *standard-input*.

standard-output *Variable*
standard-output *Variable*

In the normal Lisp top-level loop, output is sent to *standard-output* (that is, whatever stream is the value of *standard-output*). Many output functions, including tyo and print, take a stream argument that defaults to *standard-output*.

error-output *Variable*
error-output *Variable*

The value of *error-output* is a stream on which noninteractive error or warning messages should be printed. Normally this is the same as *standard-output*, but *standard-output* might be bound to a file and *error-output* left going to the terminal.

debug-io *Variable*
debug-io *Variable*

The value of *debug-io* is used for all input and output by the error handler. Normally this is a synonym for *terminal-io*. The value may be nil, which is regarded as equivalent to a synonym for *terminal-io*. This feature is provided because users often set *debug-io* by hand, and it is much easier to set it back to nil afterward than to figure out the proper synonym stream pointing to *terminal-io*.

query-io *Variable*
query-io *Variable*

The value of *query-io* is a stream that should be used when asking questions of the user. The question should be output to this stream, and the answer read from it. The reason for this is that when the normal input to a program may be coming from a file, questions such as "Do you really want to delete all of the files in your directory??" should be sent directly to the user, and the answer should come from the user, not from the data file. *query-io* is used by fquery and related functions; see page 769.

terminal-io *Variable*
terminal-io *Variable*

The value of *terminal-io* is the stream that the program should use to talk to the user's console. In an interactive program, it is the window from which the program is being run; I/O on this stream reads from the keyboard and displays on the screen. However, in a background process that has no window, *terminal-io* defaults to a stream that does not ever expect to be used. If it is used, perhaps by an error printout, it turns into a background window and requests the user's attention.

trace-output *Variable*
trace-output *Variable*

The value of *trace-output* is the stream on which the trace function prints its output.

standard-input, *standard-output*, *error-output*, *debug-io*, *trace-output*, and *query-io* are initially bound to synonym streams that pass all operations on to the stream that is the value of *terminal-io*. Thus any operations performed on those streams go to the keyboard and screen.

Most user programs should not change the value of *terminal-io*. A program which wants (for example) to divert output to a file should do so by binding the value of *standard-output*; that way queries on *query-io*, debugging on *debug-io* and error messages sent to *error-output* can still get to the user by going through *terminal-io*, which is usually what is desired.

22.3.2 Standard Input Stream Operations

:tyi &optional *eof* *Operation on streams*

The stream inputs one character and returns it. For example, if the next character to be read in by the stream is a 'C', then the form

```
(send s :tyi)
```

returns the value of #/C (that is, 103 octal). Note that the :tyi operation does not echo the character in any fashion; it just does the input. The tyi function (see page 452) does echoing when reading from the terminal.

The optional *eof* argument to the :tyi operation tells the stream what to do if it gets to the end of the file. If the argument is not provided or is nil, the stream returns nil at the end of file. Otherwise it signals a sys:end-of-file error. Note that this is *not* the same as the eof-option argument to read, tyi, and related functions.

The :tyi operation on a binary input stream returns a non-negative number, not necessarily to be interpreted as a character.

For some streams (such as windows), not all the input data are numbers. Some are lists, called *blips*. The :tyi operation returns only numbers. If the next available input is not a number, it is discarded, and so on until a number is reached (or end of file is reached).

:any-tyi &optional *eof* *Operation on streams*

Like :tyi but returns any kind of datum. Non-numbers are not discarded as they would be by :tyi. This distinction only makes a difference on streams which can provide input which is not composed of numbers; currently, only windows can do that.

:tyipeek &optional *eof* *Operation on streams*

Peeks at the next character or byte from the stream without discarding it. The next :tyi or :tyipeek operation will get the same character.

eof is the same as in the :tyi operation: if nil, end of file returns nil; otherwise, it signals a sys:end-of-file error.

:untyi *char* *Operation on streams*

Unreads the character or byte *char*; that is to say, puts it back into the input stream so that the next :tyi operation will read it again. For example,

```
(send s :untyi 120)
(send s :tyi) ==> 120
```

This operation is used by read, and any stream that supports :tyi must support :untyi as well.

You are only allowed to :untyi one character before doing a :tyi, and the character you :untyi must be the last character read from the stream. That is, :untyi can only be used to back up one character, not to stuff arbitrary data into the stream. You also can't :untyi after you have peeked ahead with :tyipeek since that does one :untyi itself. Some streams implement :untyi by saving the character, while others implement it by backing up the pointer to a buffer.

:string-in *eof-option string* &optional (*start* 0) *end* *Operation on streams*
Reads characters from the stream and stores them into the array *string*. Many streams can implement this far more efficiently that repeated :tyi's. *start* and *end*, if supplied, delimit the portion of *string* to be stored into. If *eof-option* is non-nil then a sys:end-of-file error is signaled if end of file is reached on the stream before the string has been filled. If *eof-option* is nil, any number of characters before end of file is acceptable, even no characters.

If *string* has an array-leader, the fill pointer is adjusted to *start* plus the number of characters stored into *string*.

Two values are returned: the index of the next position in *string* to be filled, and a flag that is non-nil if end of file was reached before *string* was filled. Most callers do not need to look at either of these values.

string may be any kind of array, not necessarily a string; this is useful when reading from a binary input stream.

:line-in &optional *leader* *Operation on streams*
The stream should input one line from the input source, and return it as a string with the carriage return character stripped off. Contrary to what you might assume from its name, this operation is not much like the readline function.

Many streams have a string that is used as a buffer for lines. If this string itself were returned, there would be problems caused if the caller of the stream attempted to save the string away somewhere, because the contents of the string would change when the next line was read in. In order to solve this problem, the string must be copied. On the other hand, some streams don't reuse the string, and it would be wasteful to copy it on every :line-in operation. This problem is solved by using the *leader* argument to :line-in. If *leader* is nil (the default), the stream does not bother to copy the string and the caller should not rely on the contents of that string after the next operation on the stream. If *leader* is t, the stream does make a copy. If *leader* is a fixnum then the stream makes a copy with an array leader *leader* elements long. (This is used by the editor, which represents lines of buffers as strings with additional information in their array-leaders, to eliminate an extra copy operation.)

If the stream reaches end of file while reading in characters, it returns the characters it has read in as a string and returns a second value of t. The caller of the stream should therefore arrange to receive the second value, and check it to see whether the string returned was a whole line or just the trailing characters after the last carriage return in the input source.

This operation should be implemented by all input streams whose data are characters.

:string-line-in *eof-option string* &optional (*start* 0) *end* — *Operation on streams*

Reads characters, storing them in *string*, until *string* is full or a Return character is read. If input stops due to a Return, the Return itself is not put in the buffer.

Thus, this operation is nearly the same as :string-in, except that :string-in always keeps going until the buffer is full or until end of file.

start and *end*, if supplied, delimit the portion of *string* to be stored into. If *eof-option* is non-nil then a sys:end-of-file error is signaled if end of file is reached on the stream before the string has been filled. If *eof-option* is nil, any number of characters before end of file is acceptable, even no characters.

If *string* has an array-leader, the fill pointer is adjusted to *start* plus the number of characters stored into *string*.

string may be any kind of array, not necessarily a string; this is useful when reading from a binary input stream.

Three values are returned:

(1) The index in *string* at which input stopped. This is the first index not stored in.

(2) t if input stopped due to end of file.

(3) t if the line is incomplete; that is, if a Return character did not terminate it.

:read-until-eof — *Operation on streams*

Discards all data from the stream until it is at end of file, or does anything else with the same result.

:close &optional *ignore* — *Operation on streams*

Releases resources associated with the stream, when it is not going to be used any more. On some kinds of streams, this may do nothing. On Chaosnet streams, it closes the Chaosnet connection, and on file streams, it closes the input file on the file server.

The argument is accepted for compatibility with :close on output streams.

22.3.3 Standard Output Stream Operations

:tyo *char* — *Operation on streams*

The stream outputs the character *char*. For example, if s is bound to a stream, then the form

```
        (send s :tyo #/B)
```

outputs a B to the stream. For binary output streams, the argument is a non-negative number rather than specifically a character.

:fresh-line *Operation on streams*

Tells the stream that it should position itself at the beginning of a new line. If the stream is already at the beginning of a fresh line it should do nothing; otherwise it should output a carriage return. If the stream cannot tell whether it is at the beginning of a line, it should always output a carriage return.

:string-out (*string* 0) &optional *start end* *Operation on streams*

Outputs the characters of *string* successively to *stream*. This operation is provided for two reasons: first, it saves the writing of a loop which is used very often, and second, many streams can perform this operation much more efficiently than the equivalent sequence of :tyo operations.

If *start* and *end* are not supplied, the whole string is output. Otherwise a substring is output: *start* is the index of the first character to be output (defaulting to 0), and *end* is one greater than the index of the last character to be output (defaulting to the length of the string). Callers need not pass these arguments, but all streams that handle :string-out must check for them and interpret them appropriately.

:line-out *string* &optional (*start* 0) *end* *Operation on streams*

Outputs the characters of *string* successively to *stream*, then outputs a Return character. *start* and *end* optionally specify a substring, as with :string-out. If the stream doesn't support :line-out itself, the default handler implements it by means of :tyo.

This operation should be implemented by all output streams whose data are characters.

:close &optional *mode* *Operation on streams*

Closes the stream to make the output final if this is necessary. The stream becomes *closed* and no further output operations should be performed on it. However, it is all right to :close a closed stream. On many file server hosts, a file being written is not accessible to be read until the output stream is closed.

This operation does nothing on streams for which it is not meaningful.

The *mode* argument is normally not supplied. If it is :abort, we are abnormally exiting from the use of this stream. If the stream is outputting to a file, and has not been closed already, the stream's newly-created file is deleted; it will be as if it was never opened in the first place. Any previously existing file with the same name remains undisturbed.

:eof *Operation on streams*

Indicates the end of data on an output stream. This is different from :close because some devices allow multiple data files to be transmitted without closing. :close implies :eof when the stream is an output stream and the close mode is not :abort.

This operation does nothing on streams for which it is not meaningful.

22.3.4 Asking Streams What They Can Do

All streams are supposed to support certain operations which enable a program using the stream to ask which operations are available.

:which-operations *Operation on streams*

Returns a list of operations handled natively by the stream. Certain operations not in the list may work anyway, but slowly, so it is just as well if any programs that work with or without them choose not to use them.

:which-operations itself need not be in the list.

:operation-handled-p *operation* *Operation on streams*

Returns t if *operation* is handled natively by the stream: if *operation* is a member of the :which-operations list, or is :which-operations.

:send-if-handles *operation* &rest *arguments* *Operation on streams*

Performs the operation *operation*, with the specified *arguments*, only if the stream can handle it. If *operation* is handled, this is the same as sending an *operation* message directly, but if *operation* is not handled, using :send-if-handles avoids any error.

If *operation* is handled, :send-if-handles returns whatever values the execution of the *operation* returns. If *operation* is not handled, :send-if-handles returns nil.

:direction *Operation on streams*

Returns :input, :output, or :bidirectional for a bidirectional stream.

There are a few kinds of streams, which cannot do either input or output, for which the :direction operation returns nil. For example, open with the :direction keyword specified as nil returns a stream-like object which cannot do input or output but can handle certain file inquiry operations such as :truename and :creation-date.

:characters *Operation on streams*

Returns t if the data input or output on the stream represent characters, or nil if they are just numbers (as for a stream reading a non-text file).

:element-type *Operation on streams*

Returns a type specified describing in principle the data input or output on the stream. Refer to the function stream-element-type, below, which works using this operation.

These functions for inquiring about streams are defined by Common Lisp.

input-stream-p *stream*
t if *stream* handles input operations (at least, if it handles :tyi).

output-stream-p *stream*
t if *stream* handles output operations (at least, if it handles :tyo).

stream-element-type *stream*
Returns a type specifier which describes, conceptually, the kind of data input from or output to *stream*. The value is always a subtype of integer (for a binary stream) or a subtype of character (for a character stream). If it is a subtype of integer, a Common Lisp program should use read-byte (page 450) or write-byte (page 456) for I/O. If it is a subtype of character, read-char (page 450) or write-char (page 456) should be used.

The value returned is not intended to be rigidly accurate. It describes the typical or characteristic sort of data transferred by the stream, but the stream may on occasion deal with data that do not fit the type; also, not all objects of the type may be possible as input or even make sense as output. For example, windows describe their element type as character even though they may offer blips, which are lists, as input on occasion. In addition, streams which say they provide characters really return integers if the :tyi operation is used rather than the standard Common Lisp function read-char.

22.3.5 Operations for Interactive Streams

The operations :listen, :tyi-no-hang, :rubout-handler and :beep are intended for interactive streams, which communicate with the user. :listen and :tyi-no-hang are supported in a trivial fashion by other streams, for compatibility.

:listen *Operation on streams*
On an interactive device, the :listen operation returns non-nil if there are any input characters immediately available, or nil if there is no immediately available input. On a non-interactive device, the operation always returns non-nil except at end of file.

The main purpose of :listen is to test whether the user has hit a key, perhaps trying to stop a program in progress.

:tyi-no-hang &optional *eof* *Operation on streams*
Just like :tyi except that it returns nil rather than waiting if it would be necessary to wait in order to get the character. This lets the caller check efficiently for input being available and get the input if there is any.

:tyi-no-hang is different from :listen because it reads a character.

Streams for which the question of whether input is available is not meaningful treat this operation just like :tyi. So do Chaosnet file streams. Although in fact reading a character from a file stream may involve a delay, these delays are *supposed* to be insignificant, so we pretend they do not exist.

:any-tyi-no-hang &optional *eof* *Operation on streams*

Like :tyi-no-hang but does not filter and discard input which is not numbers. It is therefore possible to see blips in the input stream. The distinction matters only for input from windows.

:rubout-handler *options function* &rest *args* *Operation on streams*

This is supported by interactive bidirectional streams, such as windows on the terminal, and is described in its own section below (see section 22.5, page 500).

:beep &optional *type* *Operation on streams*

This is supported by interactive streams. It attracts the attention of the user by making an audible beep and/or flashing the screen. *beep-type* is a keyword selecting among several different beeping noises; see beep (page 457) for a list of them.

22.3.6 Cursor Positioning Stream Operations

:read-cursorpos &optional (*units* :pixel) *Operation on streams*

This operation is supported by all windows and some other streams.

It returns two values, the current *x* and *y* coordinates of the cursor. It takes one optional argument, which is a symbol indicating in what units *x* and *y* should be; the symbols :pixel and :character are understood. :pixel means that the coordinates are measured in display pixels (bits), while :character means that the coordinates are measured in characters horizontally and lines vertically.

This operation and :increment-cursorpos are used by the format ~T request (see page 487), which is why ~T doesn't work on all streams. Any stream that supports this operation should support :increment-cursorpos as well.

Some streams return a meaningful value for the horizontal position but always return zero for the vertical position. This is sufficient for ~T to work.

:increment-cursorpos *Operation on streams*
x-increment y-increment &optional (*units* :pixel)

Moves the stream's cursor left or down according to the specified increments, as if by outputting an appropriate number of space or return characters. *x* and *y* are like the values of :read-cursorpos and *units* is the same as the *units* argument to :read-cursorpos.

Any stream which supports this operation should support :read-cursorpos as well, but it need not support :set-cursorpos.

Moving the cursor with :increment-cursorpos differs from moving it to the same place with :set-cursorpos in that this operation is thought of as doing output and :set-cursorpos is not. For example, moving a window's cursor down with :increment-cursorpos when it is near the bottom to begin with will wrap around, possibly doing a **MORE**. :set-cursorpos, by comparison, cannot move the cursor "down" if it is at the bottom of the window; it can move the cursor explicitly to the top of the window, but then no **MORE** will happen.

Some streams, such as those created by with-output-to-string, cannot implement arbitrary cursor motion, but do implement this operation.

:set-cursorpos *x y* &optional (*units* :pixel) *Operation on streams*
This operation is supported by the same streams that support :read-cursorpos. It sets the position of the cursor. *x* and *y* are like the values of :read-cursorpos and *units* is the same as the *units* argument to :read-cursorpos.

:clear-screen *Operation on streams*
Erases the screen area on which this stream displays. Non-window streams don't support this operation.

There are many other special-purpose stream operations for graphics. They are not documented here, but in the window-system documentation. No claim that the above operations are the most useful subset should be implied.

22.3.7 Operations for Efficient Pretty-Printing

grindef runs much more efficiently on streams that implement the :untyo-mark and :untyo operations.

:untyo-mark *Operation on streams*
This is used by the grinder (see page 528) if the output stream supports it. It takes no arguments. The stream should return some object that indicates how far output has gotten up to in the stream.

:untyo *mark* *Operation on streams*
This is used by the grinder (see page 528) in conjunction with :untyo-mark. It takes one argument, which is something returned by the :untyo-mark operation of the stream. The stream should back up output to the point at which the object was returned.

22.3.8 Random Access File Operations

The following operations are implemented only by streams to random-access devices, principally files.

:read-pointer *Operation on streams*
Returns the current position within the file, in characters (bytes in fixnum mode). For text files on ASCII file servers, this is the number of Lisp Machine characters, not ASCII characters. The numbers are different because of character-set translation.

:set-pointer *new-pointer* *Operation on streams*
Sets the reading position within the file to *new-pointer* (bytes in fixnum mode). For text files on ASCII file servers, this does not do anything reasonable unless *new-pointer* is 0, because of character-set translation. Some file systems support this operation for input streams only.

:rewind *Operation on streams*
This operation is obsolete. It is the same as **:set-pointer** with argument zero.

22.3.9 Buffered Stream Operations

:clear-input *Operation on streams*
Discards any buffered input the stream may have. It does nothing on streams for which it is not meaningful.

:clear-output *Operation on streams*
Discards any buffered output the stream may have. It does nothing on streams for which it is not meaningful.

:force-output *Operation on streams*
This is for output streams to buffered asynchronous devices, such as the Chaosnet. **:force-output** causes any buffered output to be sent to the device. It does not wait for it to complete; use **:finish** for that. If a stream supports **:force-output**, then **:tyo**, **:string-out**, and **:line-out** may have no visible effect until a **:force-output** is done.

This operation does nothing on streams for which it is not meaningful.

:finish *Operation on streams*
This is for output streams to buffered asynchronous devices, such as the Chaosnet. **:finish** does a **:force-output**, then waits until the currently pending I/O operation has been completed.

This operation does nothing on streams for which it is not meaningful.

The following operations are implemented only by buffered input streams. They allow increased efficiency by making the stream's internal buffer available to the user.

:read-input-buffer &optional *eof* *Operation on streams*
Returns three values: a buffer array, the index in that array of the next input byte, and the index in that array just past the last available input byte. These values are similar to the *string*, *start*, *end* arguments taken by many functions and stream operations. If the end of the file has been reached and no input bytes are available, this operation returns **nil** or signals an error, based on the *eof* argument, just like the **:tyi** operation. After reading as many bytes from the array as you care to, you must use the **:advance-input-buffer** operation.

:get-input-buffer &optional *eof* *Operation on streams*
This is an obsolete operation similar to **:read-input-buffer**. The only difference is that the third value is the number of significant elements in the buffer-array, rather than a final index. If found in programs, it should be replaced with **:read-input-buffer**.

:advance-input-buffer &optional *new-pointer* *Operation on streams*

If *new-pointer* is non-nil, it is the index in the buffer array of the next byte to be read. If *new-pointer* is nil, the entire buffer has been used up.

22.3.10 Obtaining Streams to Use

Windows are one important class of streams. Each window can be used as a stream. Output is displayed on the window and input comes from the keyboard. A window is created using make-instance on a window flavor. Simple programs use windows implicitly through *terminal-io* and the other standard stream variables.

Also important are *file streams*, which are produced by the function open (see page 582). These read or write the contents of a file.

Chaosnet streams are made from Chaosnet connections. Data output to the stream goes out over the network; data coming in over the network is available as input from the stream. File streams that deal with Chaosnet file servers are very similar to Chaosnet streams, but Chaosnet streams can be used for many purposes other than file access.

String streams read or write the contents of a string. They are made by with-output-to-string or with-input-from-string (see page 473), or by make-string-input-stream or make-string-output-stream, below.

Editor buffer streams read or write the contents of an editor buffer.

The *null stream* may be passed to a program that asks for a stream as an argument. It returns immediate end of file if used for input and throws away any output. The null stream is the symbol si:null-stream. This is to say, you do not call that function to get a stream or use the symbol's value as the stream; *the symbol itself* is the object that is the stream.

The *cold-load stream* is able to do I/O to the keyboard and screen without using the window system. It is what is used by the error handler, if you type Terminal Call, to handle a background error that the window system cannot deal with. It is called the cold-load stream because it is what is used during system bootstrapping, before the window system has been loaded.

si:null-stream *operation* &rest *arguments*

This function is the null stream. Like any stream, it supports various operations. Output operations are ignored and input operations report end of file immediately, with no data. Usage example:

```
(let ((*standard-output* 'si:null-stream))
  (function-whose-output-I-dont-want))
```

si:cold-load-stream *Constant*

The one and only cold-load stream. Usage example:

```
(let ((*query-io* si:cold-load-stream))
  (yes-or-no-p "Clear all window system locks? "))
```

with-open-stream (*variable expression*) *body...* *Macro*

body is executed with *variable* bound to the value of *expression*, which ought to be a stream. On exit, whether normal or by throwing, a :close message with argument :abort is sent to the stream.

This is a generalization of with-open-file, which is equivalent to using with-open-stream with a call to open as the *expression*.

with-open-stream-case (*variable expression*) *clauses...* *Macro*

Like with-open-stream as far as opening and closing the stream are concerned, but instead of a simple body, it has clauses like those of a condition-case that say what to do if *expression* does or does not get an error. See with-open-file-case, page 580.

make-synonym-stream *symbol-or-locative*
make-syn-stream *symbol-or-locative*

Creates and returns a *synonym* stream ('syn' for short). Any operations sent to this stream are redirected to the stream that is the value of the argument (if it is a symbol) or the contents of it (if it is a locative).

A synonym stream is actually an uninterned symbol whose function defnition is forwarded to the function cell of the argument or to the contents of the argument as appropriate. If the argument is a symbol, the synonym stream's print-name is *symbol*-syn-stream; otherwise the name is just syn-stream. Once a synonym stream is made for a symbol, it is recorded, and the same one is handed out again if there is another request for it.

The two names for this function are synonyms too.

make-concatenated-stream &rest *streams*

Returns an input stream which will read its input from the first of *streams* until that reaches its eof, then read input from the second of *streams*, and so on until the last of *streams* has reached end of file.

make-two-way-stream *input-stream output-stream*

Returns a bidirectional stream which passes input operations to *input-stream* and passes output operations to *output-stream*. This works by attempting to recognize all standard input operations; anything not recognized is passed to *output-stream*.

make-echo-stream *input-stream output-stream*

Like make-two-way-stream except that each input character read via *input-stream* is output to *output-stream* before it is returned to the caller.

make-broadcast-stream &rest *streams*

Returns a stream that only works in the output direction. Any output sent to this stream is forwarded to all of the streams given. The :which-operations is the intersection of the :which-operations of all of the streams. The value(s) returned by a stream operation are the values returned by the last stream in *streams*.

zwei:interval-stream *interval-or-from-bp* &optional *to-bp in-order-p hack-fonts*

Returns a bidirectional stream that reads or writes all or part of an editor buffer. Note that editor buffer streams can also be obtained from open by using a pathname whose host is ED, ED-BUFFER or ED-FILE (see section 24.7.6, page 575).

The first three arguments specify the buffer or portion to be read or written. Either the first argument is an *interval* (a buffer is one kind of interval), and all the text of that interval is read or written, or the first two arguments are two buffer pointers delimiting the range to be read or written. The third argument is used only in the latter case; if non-nil, it tells the function to assume that the second buffer pointer comes later in the buffer than the first and not to take the time to verify the assumption.

The stream has only one pointer inside it, used for both input and output. As you do input, the pointer advances through the text. When you do output, it is inserted in the buffer at the place where the pointer has reached. The pointer starts at the beginning of the specified range.

hack-fonts tells what to do about fonts. Its possible values are

t	The character ε is recognized as special when you output to the stream; sequences such as ε2 are interpreted as font-changes. They do not get inserted into the buffer; instead, they change the font in which following output will be inserted. On input, font change sequences are included to indicate faithfully what was in the buffer.
:tyo	You are expected to read and write 16-bit characters containing font numbers.
nil	All output is inserted in font zero and font information is discarded in the input you receive. This is the best mode to use if you are reading or otherwise parsing the contents of an editor buffer.

sys:with-help-stream (*stream options...*) *body...* *Macro*

Executes the *body* with the variable *stream* bound to a suitable stream for printing a large help message. If *standard-output* is a window, then *stream* is also a window; a temporary window which fills the screen. Otherwise, *stream* is just the same as *standard-output*.

The purpose of this is to spare the user the need to read a large help printout in a small window, or have his data overwritten by it permanently. This is the mechanism used if you type the Control-Help key while in the rubout handler.

options is a list of alternating keywords and values.

:label	The value (which is evaluated) is used as the label of the temporary window, if one is used.
:width	The value, which is not evaluated, is a symbol. While *body* is executed, this symbol is bound to the width, in characters, available for the message.

:height — The value is a symbol, like the value after :width, and it is bound to the height in lines of the area available for the help message.

:superior — The value, which is evaluated, specifies the original stream to use in deciding where to print the help message. The default is *standard-output*.

22.3.11 String I/O Streams

The functions and special forms in this section allow you to create I/O streams that input from or output to the contents of a string.

make-string-input-stream *string* &optional (*start* 0) *end*
Returns a stream which can be used to read the contents of *string* (or the portion of it from index *start* to index *end*) as input. End of file occurs on reading past position *end* or the end of string.

make-string-output-stream &optional *string*
Returns an output stream which will accumulate all output in a string. If *string* is non-nil, output is added to it with string-nconc (page 216). Otherwise, a new string is created and used to hold the output.

get-output-stream-string *string-output-stream*
Returns the string of output accumulated so far by a stream which was made by make-string-output-stream. The accumulated output is cleared out, so it will not be obtained again if get-output-stream-string is called another time on the same stream.

with-input-from-string (*var string* &key *start end index*) *body...* *Macro*
The form
```
(with-input-from-string (var string)
    body)
```
evaluates the forms in *body* with the variable *var* bound to a stream which reads characters from the string which is the value of the form *string*. The value of the construct is the value of the last form in its body.

If the *start* and *end* arguments are specified, they should be forms. They are evaluated at run time to produce the indices starting and ending the portion of *string* to be read.

If the *index* argument is specified, it should be something setf can store in. When *body* is finished, the index in the string at which reading stopped is stored there. This is the index of the first character not read. If the entire string was read, it is the length of the string. The value of *index* is not updated until with-input-from-string is exited, so you can't use its value within the body to see how far the reading has gotten. Example:
```
(with-input-from-string
    (foo "This is a test." :start (+ 2 2) :end 8 :index bar)
  (readline))
```
returns " is " and sets bar to eight.

An older calling sequence which used positional rather than keyword arguments is still accepted:

```
(with-input-from-string (var string index end)
  body)
```

The functions read-from-string and cli:read-from-string are convenient special cases of what with-input-from-string can do. See page 533.

with-output-to-string (*var* [*string* [*index*]]) *body...* *Macro*

This special form provides a variety of ways to send output to a string through an I/O stream.

```
(with-output-to-string (var)
  body)
```

evaluates the forms in *body* with *var* bound to a stream which saves the characters output to it in a string. The value of the special form is the string.

```
(with-output-to-string (var string)
  body)
```

appends its output to the string which is the value of the form *string*. (This is like the string-nconc function; see page 216.) The value returned is the value of the last form in the body, rather than the string. Multiple values are not returned. *string* must have a fill pointer. If *string* is too small to contain all the output, adjust-array-size is used to make it bigger.

```
(with-output-to-string (var string index)
  body)
```

is similar to the above except that *index* is a variable or setf-able reference which contains the index of the next character to be stored into. It must be initialized before the with-output-to-string and it is updated upon normal exit. The value of *index* is not updated until with-output-to-string returns, so you can't use its value within the body to see how far the writing has gotten. The presence of *index* means that *string* is not required to have a fill-pointer; if there is one, it is updated on exit.

Another way of doing output to a string is to use the format facility (see page 483).

22.3.12 Implementing Streams

There are two ways to implement a stream: using defun or using flavors.

Using flavors is best when you can take advantage of the predefined stream mixins, including those which perform buffering, or when you wish to define several similar kinds of streams that can inherit methods from each other.

defun (or defselect, which is a minor variation of the technique) may have an advantage if you are dividing operations into broad groups and handling them by passing them off to one or more other streams. In this case, the automatic operation decoding provided by flavors may get in the way. A number of streams in the system are implemented using defun or defselect for historical reasons. It isn't yet clear whether there is any reason not to convert most of them to use flavors.

If you use **defun**, you can use the *stream default handler* to implement some of the standard operations for you in a default manner. If you use flavors, there are predefined mixins to do this for you.

A few streams are individual objects, one of a kind. For example, there is only one null stream, and no need for more, since two null streams would behave identically. But most streams are elements of a general class. For example, there can be many file streams for different files, even though all behave the same way. There can also be multiple streams reading from different points in the same file.

If you implement a class of streams with **defun**, then the actual streams must be closures of the function you define, made with **closure**.

If you use flavors to implement the streams, having a class of similar streams comes naturally: each instance of the flavor is a stream, and the instance variables distinguish one stream of the class from another.

22.3.13 Implementing Streams with Flavors

To define a stream using flavors, define a flavor which incorporates the appropriate predefined stream flavor, and then redefine those operations which are peculiar to your own type of stream.

Flavors for defining unbuffered streams:

si:stream *Flavor*

This flavor provides default definitions for a few standard operations such as **:direction** and **:characters**. Usually you do not have to mention this explicitly; instead you use the higher level flavors below, which are built on this one.

si:input-stream *Flavor*

This flavor provides default definitions of all the mandatory input operations except **:tyi** and **:untyi**, in terms of those two. You can make a simple non-character input stream by defining a flavor incorporating this one and giving it methods for **:tyi** and **:untyi**.

si:output-stream *Flavor*

This flavor provides default definitions of all the mandatory output operations except **:tyo**, in terms of **:tyo**. All you need to do to define a simple unbuffered non-character output stream is to define a flavor incorporating this one and give it a method for the **:tyo** operation.

si:bidirectional-stream *Flavor*

This is a combination of **si:input-stream** and **si:output-stream**. It defines **:direction** to return **:bidirectional**. To define a simple unbuffered non-character bidirectional stream, build on this flavor and define **:tyi**, **:untyi** and **:tyo**.

The unbuffered streams implement operations such as :string-out and :string-in by repeated use of :tyo or :tyi.

For greater efficiency, if the stream's data is available in blocks, it is better to define a buffered stream. You start with the predefined buffered stream flavors, which define :tyi or :tyo themselves and manage the buffers for you. You must provide other operations that the system uses to obtain the next input buffer or to write or discard an output buffer.

Flavors for defining buffered streams:

si:buffered-input-stream *Flavor*

This flavor is the basis for a non-character buffered input stream. It defines :tyi as well as all the other standard input operations, but you must define the two operations :next-input-buffer and :discard-input-buffer, which the buffer management routines use.

:next-input-buffer *Operation on* si:buffered-input-stream

In a buffered input stream, this operation is used as a subroutine of the standard input operations, such as :tyi, to get the next bufferful of input data. It should return three values: an array containing the data, a starting index in the array, and an ending index. For example, in a Chaosnet stream, this operation would get the next packet of input data and return pointers delimiting the actual data in the packet.

:discard-input-buffer *buffer-array* *Operation on* si:buffered-input-stream

In a buffered input stream, this operation is used as a subroutine of the standard input operations such as :tyi. It says that the buffer management routines have used or thrown away all the input in a buffer, and the buffer is no longer needed.

In a Chaosnet stream, this operation would return the packet buffer to the pool of free packets.

si:buffered-output-stream *Flavor*

This flavor is the basis for a non-character buffered output stream. It defines :tyo as well as all the other standard output operations, but you must define the operations :new-output-buffer, :send-output-buffer and :discard-output-buffer, which the buffer management routines use.

:new-output-buffer *Operation on* si:buffered-output-stream

In a buffered output stream, this operation is used as a subroutine of the standard output operations, such as :tyo, to get an empty buffer for storing more output data. How the buffer is obtained depends on the kind of stream, but in any case this operation should return an array (the buffer), a starting index, and an ending index. The two indices delimit the part of the array that is to be used as a buffer.

For example, a Chaosnet stream would get a packet from the free pool and return indices delimiting the part of the packet array which can hold data bytes.

:send-output-buffer *buffer-array ending-index* — *Operation on* si:buffered-output-stream

In a buffered output stream, this operation is used as a subroutine of the standard output operations, such as :tyo, to send the data in a buffer that has been completely or partially filled.

ending-index is the first index in the buffer that has not actually been stored. This may not be the same as the ending index that was returned by the :new-output-buffer operation that was used to obtain this buffer; if a :force-output is being handled, *ending-index* indicates how much of the buffer is currently full.

The method for this operation should process the buffer's data and, if necessary, return the buffer to a free pool.

:discard-output-buffer *buffer-array* — *Operation on* si:buffered-output-stream

In a buffered output stream, this operation is used as a subroutine of the standard output operations, such as :clear-output, to free an output buffer and say that the data in it should be ignored.

It should simply return *buffer-array* to a free pool, if appropriate.

Some buffered output streams simply have one buffer array which they use over and over. For such streams, :new-output-buffer can simply return that particular array each time; :send-output-buffer and :discard-output-buffer do not have to do anything about returning the buffer to a free pool. In fact, :discard-output-buffer can probably do nothing.

si:buffered-stream — *Flavor*

This is a combination of si:buffered-input-stream and si:buffered-output-stream, used to make a buffered bidirectional stream. The input and output buffering are completely independent of each other. You must define all five of the low level operations: :new-output-buffer, :send-output-buffer and :discard-output-buffer for output, and :next-input-buffer and :discard-input-buffer for input.

The data in most streams are characters. Character streams should support either :line-in or :line-out in addition to the other standard operations.

si:unbuffered-line-input-stream — *Flavor*

This flavor is the basis for unbuffered character input streams. You need only define :tyi and :untyi.

si:line-output-stream-mixin — *Flavor*

To make an unbuffered character output stream, mix this flavor into the one you define, together with si:output-stream. In addition, you must define :tyo, as for unbuffered non-character streams.

si:buffered-input-character-stream *Flavor*

This is used just like si:buffered-input-stream, but it also provides the :line-in operation and makes :characters return t.

si:buffered-output-character-stream *Flavor*

This is used just like si:buffered-output-stream, but it also provides the :line-out operation and makes :characters return t.

si:buffered-character-stream *Flavor*

This is used just like si:buffered-stream, but it also provides the :line-in and :line-out operations and makes :characters return t.

To make an unbuffered random-access stream, you need only define the :read-pointer and :set-pointer operations as appropriate. Since you provide the :tyi or :tyo handler yourself, the system cannot help you.

In a buffered random-access stream, the random access operations must interact with the buffer management. The system provides for this.

si:input-pointer-remembering-mixin *Flavor*

Incorporate this into a buffered input stream to support random access. This flavor defines the :read-pointer and :set-pointer operations. If you wish :set-pointer to work, you must provide a definition for the :set-buffer-pointer operation. You need not do so if you wish to support only :read-pointer.

:set-buffer-pointer *new-pointer* *Operation on* si:input-pointer-remembering-mixin

You must define this operation if you use si:input-pointer-remembering-mixin and want the :set-pointer operation to work.

This operation should arrange for the next :next-input-buffer operation to provide a bufferful of data that includes the specified character or byte position somewhere inside it.

The value returned should be the file pointer corresponding to the first character or byte of that next bufferful.

si:output-pointer-remembering-mixin *Flavor*

Incorporate this into a buffered output stream to support random access. This mixin defines the :read-pointer and :set-pointer operations. If you wish :set-pointer to work, you must provide definitions for the :set-buffer-pointer and :get-old-data operations. You need not do so if you wish to support only :read-pointer.

:set-buffer-pointer *Operation on* si:output-pointer-remembering-mixin
new-pointer

This is the same as in si:input-pointer-remembering-mixin.

:get-old-data *Operation on* si:output-pointer-remembering-mixin
buffer-array lower-output-limit

The buffer management routines perform this operation when you do a :set-pointer that is outside the range of pointers that fit in the current output buffer. They first send the old buffer, then do :set-buffer-pointer as described above to say where in the file the next output buffer should come, then do :new-output-buffer to get the new buffer. Then the :get-old-data operation is performed.

It should fill current buffer (*buffer-array*) with the *old* contents of the file at the corresponding addresses, so that when the buffer is eventually written, any bytes skipped over by random access will retain their old values.

The instance variable si:stream-output-lower-limit is the starting index in the buffer of the part that is supposed to be used for output. si:stream-output-limit is the ending index. The instance variable si:output-pointer-base is the file pointer corresponding to the starting index in the buffer.

si:file-stream-mixin *Flavor*

Incorporate this mixin together with si:stream to make a *file probe stream*, which cannot do input or output but records the answers to an enquiry about a file. You should specify the init option :pathname when you instantiate the flavor.

You must provide definitions for the :plist and :truename operations; in terms of them, this mixin defines the operations :get, :creation-date, and :info.

si:input-file-stream-mixin *Flavor*

Incorporate this mixin into input streams that are used to read files. You should specify the file's pathname with the :pathname init option when you instantiate the flavor.

In addition to the services and requirements of si:file-stream-mixin, this mixin takes care of mentioning the file in the who-line. It also includes si:input-pointer-remembering-mixin so that the :read-pointer operation, at least, will be available.

si:output-file-stream-mixin *Flavor*

This is the analogue of si:input-file-stream-mixin for output streams.

22.3.14 Implementing Streams Without Flavors

You do not need to use flavors to implement a stream. Any object that can be used as a function, and decodes its first argument appropriately as an operation name, can serve as a stream. Although in practice using flavors is as easy as any other way, it is educational to see how to define streams "from scratch".

We could begin to define a simple output stream, which accepts characters and conses them onto a list, as follows:

```
(defvar the-list nil)

(defun list-output-stream (op &optional arg1 &rest rest)
  (ecase op
    (:tyo
     (setq the-list (cons arg1 the-list)))
    (:which-operations '(:tyo))))
```

This is an output stream, and so it supports the :tyo operation. All streams must support :which-operations.

The lambda-list for a stream defined with a defun must always have one required parameter (*op*), one optional parameter (*arg1*), and a rest parameter (*rest*).

This definition is not satisfactory, however. It handles :tyo properly, but it does not handle :string-out, :direction, :send-if-handles, and other standard operations.

The function stream-default-handler exists to spare us the trouble of defining all those operations from scratch in simple streams like this. By adding one additional clause, we let the default handler take care of all other operations, if it can.

```
(defun list-output-stream (op &optional arg1 &rest rest)
    (selectq op
        (:tyo
         (setq the-list (cons arg1 the-list)))
        (:which-operations '(:tyo))
        (otherwise
          (stream-default-handler #'list-output-stream
                                  op arg1 rest))))
```

If the operation is not one that the stream understands (e.g. :string-out), it calls stream-default-handler. Note how the rest argument is passed to it. This is why the argument list must look the way it does. stream-default-handler can be thought of as a restricted analogue of flavor inheritance.

If we want to have only one stream of this sort, the symbol list-output-stream can be used as the stream. The data output to it will appear in the global value of the-list. One more step is required, though:

```
(defprop list-output-stream t si:io-stream-p)
```

This tells certain functions including read to treat the symbol list-output-stream as a stream rather than as an end of file option.

If we wish to be able to create any number of list output streams, each accumulating its own list, we must use closures:

```
(defvar the-stream nil
  "Inside a list output stream, holds the stream itself.")
(defvar the-list nil
  "Inside a list output stream,
holds the list of characters being accumulated.")

(defun list-output-stream (op &optional arg1 &rest rest)
    (selectq op
        (:tyo
         (push arg1 the-list)))
        (:withdrawal (prog1 the-list (setq the-list nil)))
        (:which-operations '(:tyo :withdrawal))
        (otherwise
          (stream-default-handler the-stream
                                  op arg1 rest))))

(defun make-list-output-stream ()
  (let ((the-stream the-list))
    (setq the-stream
          (closure '(the-stream the-list)
                   'list-output-stream))))
```

We have added a new operation **:withdrawal** that can be used to find out what data has been accumulated by a stream. This is necessary because we can no longer simply look at or set the global value of **the-list**; that is not the same as the value closed into the stream.

In addition, we have a new variable **the-stream** which allows the function **list-output-stream** to know which stream it is serving at any time. This variable is passed to **stream-default-handler** so that when it simulates **:string-out** by means of **:tyo**, it can do the **:tyo**'s to the same stream that the **:string-out** was done to.

The same stream could be defined with **defselect** instead of **defun**. It actually makes only a small difference. The **defun** for **list-output-stream** could be replaced with this code:

```
(defselect (list-output-stream list-output-d-h)
  (:tyo (arg1)
    (push arg1 the-list))
  (:withdrawal ()
    (prog1 the-list (setq the-list nil))))

(defun list-output-d-h (op &optional arg1 &rest rest)
  (stream-default-handler the-stream op arg1 rest))
```

defselect takes care of decoding the operations, provides a definition for **:which-operations**, and allows you to write a separate lambda list for each operation.

By comparison, the same stream defined using flavors looks like this:

```
(defflavor list-output-stream ((the-list nil))
           (si:line-output-stream-mixin si:output-stream))

(defmethod (list-output-stream :tyo) (character)
  (push character the-list))

(defmethod (list-outut-stream :withdrawal) ()
  (prog1 the-list (setq the-list nil)))

(defun make-list-output-stream ()
  (make-instance 'list-output-stream))
```

Here is a simple input stream, which generates successive characters of a list.

```
(defvar the-list)       ;Put your input list here
(defvar the-stream)
(defvar untyied-char nil)

(defun list-input-stream (op &optional arg1 &rest rest)
  (selectq op
    (:tyi
     (cond ((not (null untyied-char))
            (prog1 untyied-char (setq untyied-char nil)))
           ((null the-list)
            (and arg1 (error arg1)))
           (t (pop the-list))))
    (:untyi
     (setq untyied-char arg1))
    (:which-operations '(:tyi :untyi))
    (otherwise
      (stream-default-handler the-stream
                              op arg1 rest))))

(defun make-list-input-stream (the-list)
  (let (the-stream untyied-char)
    (setq the-stream
          (closure '(the-list the-stream untyied-char)
                   'list-input-stream))))
```

The important things to note are that :untyi must be supported, and that the stream must check for having reached the end of the information and do the right thing with the argument to the :tyi operation.

stream-default-handler *stream op arg1 rest*

Tries to handle the *op* operation on *stream*, given arguments of *arg1* and the elements of *rest*. The exact action taken for each of the defined operations is explained with the documentation on that operation, above.

22.4 Formatted Output

There are two ways of doing general formatted output. One is the function **format**. The other is the **output** subsystem. **format** uses a control string written in a special format specifier language to control the output format. **format:output** provides Lisp functions to do output in particular formats.

For simple tasks in which only the most basic format specifiers are needed, **format** is easy to use and has the advantage of brevity. For more complicated tasks, the format specifier language becomes obscure and hard to read. Then **format:output** becomes advantageous because it works with ordinary Lisp control constructs.

22.4.1 The Format Function

format *destination control-string* &rest *args*

Produces formatted output. **format** outputs the characters of *control-string*, except that a tilde ('~') introduces a directive. The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies what kind of formatting is desired. Most directives use one or more elements of *args* to create their output; the typical directive puts the next element of *args* into the output, formatted in some special way.

The output is sent to *destination*. If *destination* is **nil**, a string is created which contains the output; this string is returned as the value of the call to **format**. In all other cases **format** returns no interesting value (generally it returns **nil**). If *destination* is a stream, the output is sent to it. If *destination* is **t**, the output is sent to ***standard-output***. If *destination* is a string with an array-leader, such as would be acceptable to **string-nconc** (see page 216), the output is added to the end of that string.

A directive consists of a tilde, optional prefix parameters separated by commas, optional colon (':') and atsign ('@') modifiers, and a single character indicating what kind of directive this is. The alphabetic case of the character is ignored. The prefix parameters are generally decimal numbers. Examples of control strings:

```
"~S"          ; This is an S directive with no parameters.
"~3,4:@s"     ; This is an S directive with two parameters, 3 and 4,
              ;   and both the colon and atsign flags.
"~,4S"        ; The first prefix parameter is omitted and takes
              ;   on its default value, while the second is 4.
```

format includes some extremely complicated and specialized features. It is not necessary to understand all or even most of its features to use **format** efficiently. The beginner should skip over anything in the following documentation that is not immediately useful or clear. The more sophisticated features are there for the convenience of programs with complicated formatting requirements.

Sometimes a prefix parameter is used to specify a character, for instance the padding character in a right- or left-justifying operation. In this case a single quote (" ' ") followed by the desired character may be used as a prefix parameter, so that you don't have to know the decimal numeric values of characters in the character set. For example, you can use "~5,'0d" instead of "~5,48d" to print a decimal number in five columns with leading zeros.

In place of a prefix parameter to a directive, you can put the letter V, which takes an argument from *args* as a parameter to the directive. Normally this should be a number but it doesn't really have to be. This feature allows variable column-widths and the like. Also, you can use the character # in place of a parameter; it represents the number of arguments remaining to be processed.

Here are some relatively simple examples to give you the general flavor of how **format** is used.

```
(format nil "foo") => "foo"
(setq x 5)
(format nil "The answer is ~D." x) => "The answer is 5."
(format nil "The answer is ~3D." x) => "The answer is   5."
(setq y "elephant")
(format nil "Look at the ~A!" y) => "Look at the elephant!"
(format nil "The character ~:@C is strange." #\meta-beta)
      => "The character Meta-β (Greek-b) is strange."
(setq n 3)
(format nil "~D item~:P found." n) => "3 items found."
(format nil "~R dog~:[s are~; is~] here." n (= n 1))
      => "three dogs are here."
(format nil "~R dog~:*~[~1; is~:;s are~] here." n)
      => "three dogs are here."
(format nil "Here ~[~1;is~:;are~] ~:*~R pupp~:@P." n)
      => "Here are three puppies."
```

The directives will now be described. *arg* will be used to refer to the next argument from *args*.

~A *arg*, any Lisp object, is printed without escaping (as by **princ**). ~:A prints ( ) if *arg* is nil; this is useful when printing something that is always supposed to be a list. ~*n*A inserts spaces on the right, if necessary, to make the column width at least *n*. The @ modifier causes the spaces to be inserted on the left rather than the right. ~*mincol,colinc,minpad,padchar*A is the full form of ~A, which allows elaborate control of the padding. The string is padded on the right with at least *minpad* copies of *padchar*; padding characters are then inserted *colinc* characters at a time until the total width is at least *mincol*. The defaults are 0 for *mincol* and *minpad*, 1 for *colinc*, and space for *padchar*.

~S This is just like ~A, but *arg* is printed *with* escaping (as by **prin1** rather than **princ**).

~D *arg*, a number, is printed in base ten. Unlike **print**, ~D never puts a decimal point after the number. ~*n*D uses a column width of *n*; spaces are inserted on the left if the number requires less than *n* columns for its digits and sign. If the number doesn't fit in *n* columns, additional columns are used as needed. ~*n*,*m*D uses *m* as the pad character instead of space. If *arg* is not a number, it is printed in ~A format and decimal base. The @ modifier causes the number's sign to be printed always; the default is only to print it if the number is negative. The : modifier causes commas to be printed between groups of three digits; the third prefix parameter may be used to change the character used as the comma. Thus the most general form of ~D is ~*mincol*,*padchar*,*commachar*D.

~O This is just like ~D but prints in octal instead of decimal.

~X This is just like ~D but prints in hex instead of decimal. Note that ~X used to have a different meaning: print one or more spaces. Uses of ~X intended to have this meaning should be replaced with ~@T.

~B This is just like ~D but prints in binary instead of decimal.

~*w*,*d*,*k*,*ovfl*,*pad* F
arg is printed in nonexponential floating point format, as in '10.5'. (If the magnitude of *arg* is very large or very small, it is printed in exponential notation.) The parameters control the details of the formatting.

w is the total field width desired. If omitted, this is not constrained.

d is the number of digits to print after the decimal point. If *d* is omitted, it is chosen to do a good job based on *w* (if specified) and the value of *arg*.

k is a scale factor. *arg* is multiplied by (exp 10. *k*) before it is printed.

ovfl is a character to use for overflow. If *arg* is too big to print and fit the constraints of field width, etc., and *ovfl* is specified then the whole field is filled with *ovfl*. If *ovfl* is not specified, *arg* is printed using extra width as needed.

pad is a character to use for padding on the left, when the field width is specified and not that many characters are really needed.

If the @ modifier is used, a sign is printed even if *arg* is positive.

Rational numbers are converted to floats and then printed. Anything else is printed with ~*w*D format.

~*w*,*d*,*e*,*k*,*ovfl*,*pad*,*expt* E
arg is printed in exponential notation, as in '.105e+2'. The parameters control the details of the formatting.

w is the total field width desired. If omitted, this is not constrained.

d and *k*
control the number of mantissa digits and their arrangement around the decimal point. $d+1$ digits are printed. If *k* is positive, all of them are significant digits, and the decimal point is printed after the first *k* of them. If *k* is zero or negative, the first $|k|+1$ of the $d+1$ digits are leading zeros, and the decimal point follows the first zero. (This zero can be omitted if necessary to fit the number in *w* characters.) So the number of significant figures is less than *d* if *k* is negative.

The exponent printed always compensates for any powers of ten introduced according to *k*, so 10.5 might be printed as 0.105e+2 or as 1050.0e-2.

If *d* is omitted, the system chooses enough significant figures to represent the float accurately. If *k* is omitted, the default is one.

e is the number of digits to use for the exponent. If it is not specified, however many digits are needed are used.

ovfl is the overflow character. If the exponent doesn't fit in *e* digits or the entire number does not fit in *w* characters, then if *ovfl* is specified, the field of *w* characters is filled with *ovfl*. Otherwise more characters are used as needed.

pad is a character to use for padding on the left, when the field width is specified and not that many characters are really needed.

expt is a character to use to separate the mantissa from the exponent. The default is e or s or f, whichever would be used in printing the number normally.

If the @ modifier is used, a sign is printed even if *arg* is positive.

~*w,d,e,k,ovfl,pad,expt* G
Prints a floating point number *arg* in either ~F or ~E format. Fixed format is used if the absolute value of *arg* is less than (expt 10. *d*), and exponential format otherwise. (If *d* is not specified, it defaults based on the value of *arg*.) If fixed format is used, *e*+2 blanks are printed at the end (where the exponent and its separator and sign would go, in exponential format). These count against the width *w* if that is specified. Four blanks are used if *e* is omitted. The diminished width available, *d*, *ovfl* and *pad* are used as specified. The scale factor used in fixed format is always zero, not *k*.

If exponential format needs to be used, all the parameters are passed to the ~E directive to print the number.

Rational numbers are converted to floats and then printed. Anything else is printed with ~*w*D format.

~$ ~*rdig,ldig,field,padchar*$ prints *arg*, a float, with exactly *rdig* digits after the decimal point. The default for *rdig* is 2, which is convenient for printing amounts of money. At least *ldig* digits are printed preceding the decimal point; leading zeros are printed if there would be fewer than *ldig*. The default for *ldig* is 1. The number is right justified in a field *field* columns long, padded out with *padchar*. The colon modifier means that the sign character is to be at the beginning of the field, before the padding, rather than just to the left of the number. The atsign modifier says that the sign character should always be output.

If *arg* is not a number, or is unreasonably large, it is printed in ~*field,,,padchar*@A format; i.e. it is princ'ed right-justified in the specified field width.

~C (character *arg*) is put in the output. *arg* is treated as a keyboard character (see page 206), thus it may contain extra control-bits. These are printed first by representing them with abbreviated prefixes: 'C-' for Control, 'M-' for Meta, 'H-' for Hyper, and 'S-' for Super.

With the colon flag (~:C), the names of the control bits are spelled out (e.g. 'Control-Meta-F') and non-printing characters are represented by their names (e.g. 'Return') rather than being output as themselves. The printing characters Space and Altmode are also represented as their names, but all others are printed directly.

With both colon and atsign (~:@C), the colon-only format is printed, and then if the character requires the Top or Greek (Front) shift key(s) to type it, this fact is mentioned (e.g. '∀ (Top-U)'). This is the format used for telling the user about a key he is expected to type, for instance in prompt messages.

For all three of these formats, if the character is a mouse character, it is printed as Mouse-, the name of the button, '-', and the number of clicks.

With just an atsign (~@C), the character is printed in such a way that the Lisp reader can understand it, using '#\' or '#/', depending on the escaping character of *readtable* (see page 516).

~% Outputs a carriage return. ~*n*% outputs *n* carriage returns. No argument is used. Simply putting a carriage return in the control string would work, but ~% is usually used because it makes the control string look nicer in the Lisp source program.

~& The :fresh-line operation is performed on the output stream. Unless the stream knows that it is already at the front of a line, this outputs a carriage return. ~*n*& does a :fresh-line operation and then outputs *n*-1 carriage returns.

~| Outputs a page separator character (#\page). ~*n*| does this *n* times. With a : modifier, if the output stream supports the :clear-screen operation this directive clears the screen, otherwise it outputs page separator character(s) as if no : modifier were present. | is vertical bar, not capital I.

~~ Outputs a tilde. ~*n*~ outputs *n* tildes.

~<CR> Tilde immediately followed by a carriage return ignores the carriage return and any whitespace at the beginning of the next line. With a :, the whitespace is left in place. With an @, the carriage return is left in place. This directive is typically used when a format control string is too long to fit nicely into one line of the program.

~* *arg* is ignored. ~*n** ignores the next *n* arguments. ~:* "ignores backwards"; that is, it backs up in the list of arguments so that the argument last processed will be processed again. ~*n*:* backs up *n* arguments. ~*n*@* is absolute; it moves to argument *n* (*n* = 0 specifies the first argument).

When within a ~{ construct (see below), the ignoring (in either direction) is relative to the list of arguments being processed by the iteration.

~P If *arg* is not 1, a lower-case 's' is printed. ('P' is for 'plural'.) ~:P does the same thing, after doing a ~:*; that is, it prints a lower-case s if the *last* argument was not 1. ~@P prints 'y' if the argument is 1, or 'ies' if it is not. ~:@P does the same thing, but backs up first.

~T Spaces over to a given column. ~*n*,*m*T outputs sufficient spaces to move the cursor to column *n*. If the cursor is already past column *n*, it outputs spaces to move it to column *n*+*mk*, for the smallest integer value *k* possible. *n* and *m* default to 1. Without the colon flag, *n* and *m* are in units of characters; with it, they are in units of pixels.

Note: this operation works properly *only* on streams that support the :read-cursorpos and :increment-cursorpos stream operations (see page 467). On other streams, any ~T operation simply outputs two spaces. When **format** is creating a string, ~T works by assuming that the first character in the string is at the left margin.

~@T simply outputs a space. ~*rel* T simply outputs *rel* spaces. ~*rel*, *period* T outputs *rel* spaces and then additional spaces until it reaches a column which is a multiple of *period*. If the output stream does not support :read-cursorpos then it simply outputs *rel* spaces.

~R ~R prints *arg* as a cardinal English number, e.g. four. ~:R prints *arg* as an ordinal number, e.g. fourth. ~@R prints *arg* as a Roman numeral, e.g. IV. ~:@R prints *arg* as an old Roman numeral, e.g. IIII.

~*n*R prints *arg* in radix *n*. The flags and any remaining parameters are used as for the ~D directive. Indeed, ~D is the same as ~10R. The full form here is therefore ~*radix*,*mincol*,*padchar*,*commachar*R.

~? Uses up two arguments, and processes the first one as a format control string using the second one's elements as arguments. Thus,

```
        (format nil "~? ~D" "~O ~O" '(4 20.) 9)
```

returns "4 24 9".

~@? processes the following argument as a format control string, using all the remaining arguments. Any arguments it does not use are left to be processed by the format directives following the ~@? in the original control string.

```
        (format nil "~@? ~D" "~O ~O" 4 20. 9)
```

likewise returns "4 24 9".

~→*str*~← Performs the formatting specified by *str*, with indentation on any new lines. Each time a Return is printed during the processing of *str*, it is followed by indentation sufficient to line up underneath the place where the cursor was at the beginning of *str*. For example,

```
        (format t "Foo: ~8T~→~A~←" string)
```

prints *string* with each line starting at column 8. If *string* is (string-append "This is" #\return "the string") then the output is

```
        Foo:    This is
                the string
```

~(*str*~) Performs output with case conversion. The formatting specified by *str* is done, with all the letters in the resulting output being converted to upper or lower case according to the modifiers given to the ~(command:

~(without modifiers
Converts all the letters to lower case.

~:(Converts the first letter of each word to upper case and the rest to lower case.

~@(Converts the first letter of the first word to upper case, and all other letters to lower case.

~:@(Converts all the letters to upper case.

~1(Converts the first letter of the first word to upper case and does not change anything else. If you arrange to generate all output in lower case except for letters that should be upper case regardless of context, you can use this directive when the output appears at the beginning of a sentence.

Example:
```
        "~(FoO BaR~) ~:(FoO BaR~) ~@(FoO BaR~) ~:@(FoO BaR~)
        ~1(at the White Hart~)"
```
produces
```
        foo bar Foo Bar Foo bar FOO BAR
        At the White Hart
```

~[*str0*~;*str1*~;...~;*strn*~]

This is a set of alternative control strings. The alternatives (called *clauses*) are separated by ~; and the construct is terminated by ~]. For example,
```
        "~[Siamese ~;Manx ~;Persian ~;Tortoise-Shell ~
           ~;Tiger ~;Yu-Shiang ~]kitty"
```
The *arg*th alternative is selected; 0 selects the first. If a prefix parameter is given (i.e. ~*n*[), then the parameter is used instead of an argument (this is useful only if the parameter is '#'). If *arg* is out of range no alternative is selected. After the selected alternative has been processed, the control string continues after the ~].

~[*str0*~;*str1*~;...~;*strn*~:;*default*~] has a default case. If the *last* ~; used to separate clauses is instead ~:;, then the last clause is an "else" clause, which is performed if no other clause is selected. For example,
```
        "~[Siamese ~;Manx ~;Persian ~;Tiger ~
           ~;Yu-Shiang ~:;Bad ~] kitty"
```

~[~*tag00*,*tag01*,...;*str0*~*tag10*,*tag11*,...;*str1*...~] allows the clauses to have explicit tags. The parameters to each ~; are numeric tags for the clause which follows it. That clause is processed which has a tag matching the argument. If ~*a1*,*a2*,*b1*,*b2*,...:; (note the colon) is used, then the following clause is tagged not by single values but by ranges of values *a1* through *a2* (inclusive), *b1* through *b2*, etc. ~:; with no parameters may be used at the end to denote a default clause. For example,
```
        "~[~'+,'-,'*,'//;operator ~'A,'Z,'a,'z:;letter ~
           ~'0,'9:;digit ~:;other ~]"
```

~:[*false*~;*true*~] selects the *false* control string if *arg* is nil, and selects the *true* control string otherwise.

~@[*true*~] tests the argument. If it is not nil, then the argument is not used up, but is the next one to be processed, and the one clause is processed. If it is nil, then the argument is used up, and the clause is not processed. For example,
```
        (setq *print-level* nil *print-length* 5)
        (format nil
                "~@[ *PRINT-LEVEL*=~D~]~@[ *PRINT-LENGTH*=~D~]"
                prinlevel prinlength)
           =>  " *PRINT-LENGTH*=5"
```

The combination of ~[and # is useful, for example, for dealing with English conventions for printing lists:

```
        (setq foo "Items:~#[ none~; ~S~; ~S and ~
                   ~S~:;~@{~#[~1; and~] ~S~^,~}~].")
        (format nil foo)
                   =>  "Items: none."
        (format nil foo 'foo)
                   =>  "Items: FOO."
        (format nil foo 'foo 'bar)
                   =>  "Items: FOO and BAR."
        (format nil foo 'foo 'bar 'baz)
                   =>  "Items: FOO, BAR, and BAZ."
        (format nil foo 'foo 'bar 'baz 'quux)
                   =>  "Items: FOO, BAR, BAZ, and QUUX."
```

~; Separates clauses in ~[and ~< constructions. It is undefined elsewhere.

~] Terminates a ~[. It is undefined elsewhere.

~{*str*~} This is an iteration construct. The argument should be a list, which is used as a set of arguments as if for a recursive call to **format**. The string *str* is used repeatedly as the control string. Each iteration can absorb as many elements of the list as it likes; if *str* uses up two arguments by itself, then two elements of the list get used up each time around the loop. If before any iteration step the list is empty, then the iteration is terminated. Also, if a prefix parameter *n* is given, then there can be at most *n* repetitions of processing of *str*. Here are some simple examples:

```
        (format nil "Here it is:~{ ~S~}." '(a b c))
             => "Here it is: A B C."
        (format nil "Pairs of things:~{ <~S,~S>~}." '(a 1 b 2 c 3))
             => "Pairs of things: <A,1> <B,2> <C,3>."
```

Using ~^ as well, to terminate *str* if no arguments remain, we can print a list with commas between the elements:

```
        (format nil "Elements: ~{~S~^, ~}." '(a b c))
             => "Elements: A, B, C."
```

~:{*str*~} is similar, but the argument should be a list of sublists. At each repetition step one sublist is used as the set of arguments for processing *str*; on the next repetition a new sublist is used, whether or not all of the last sublist had been processed. Example:

```
        (format nil "Pairs of things:~:{ <~S,~S>~}."
                    '((a 1) (b 2) (c 3)))
             => "Pairs of things: <A,1> <B,2> <C,3>."
```

~@{*str*~} is similar to ~{*str*~}, but instead of using one argument which is a list, all the remaining arguments are used as the list of arguments for the iteration. Example:

```
(format nil "Pairs of things:~@{ <~S,~S>~}."
            'a 1 'b 2 'c 3)
      => "Pairs of things: <A,1> <B,2> <C,3>."
```

~:@{*str*~} combines the features of ~:{*str*~} and ~@{*str*~}. All the remaining arguments are used, and each one must be a list. On each iteration the next argument is used as a list of arguments to *str*. Example:

```
(format nil "Pairs of things:~:@{ <~S,~S>~}."
            '(a 1) '(b 2) '(c 3))
      => "Pairs of things: <A,1> <B,2> <C,3>."
```

Terminating the repetition construct with ~:} instead of ~} forces *str* to be processed at least once even if the initial list of arguments is null (however, it does not override an explicit prefix parameter of zero).

If *str* is empty, then an argument is used as *str*. It must be a string, and precedes any arguments processed by the iteration. As an example, the following are equivalent:

```
(apply #'format stream string args)
(format stream "~1{~:}" string args)
```

This uses string as a formatting string. The ~1{ says it must be processed at most once, and the ~:} says it must be processed at least once. Therefore it is processed exactly once, using args as the arguments.

As another example, the format function itself uses format-error (a routine internal to the format package) to signal error messages, which in turn uses ferror, which uses format recursively. Now format-error takes a string and arguments, just like format, but also prints some additional information: if the control string in ctl-string actually is a string (it might be a list—see below), then it prints the string and a little arrow showing where in the processing of the control string the error occurred. The variable ctl-index points one character after the place of the error.

```
(defun format-error (string &rest args)
  (if (stringp ctl-string)
      (ferror nil "~1{~:}~%~VT↓~%~3@T/"~A/"~%"
                  string args (+ ctl-index 3) ctl-string)
    (ferror nil "~1{~:}" string args)))
```

This first processes the given string and arguments using ~1{~:}, then tabs a variable amount for printing the down-arrow, then prints the control string between double-quotes. The effect is something like this:

```
(format t "The item is a ~[Foo~;Bar~;Loser~]." 'quux)
>>ERROR: The argument to the FORMAT "~[" command
         must be a number
                    ↓
   "The item is a ~[Foo~;Bar~;Loser~]."
...
```

~} Terminates a ~{. It is undefined elsewhere.

~< ~*mincol,colinc,minpad,padchar*<*text*~> justifies *text* within a field at least *mincol* wide. *text* may be divided up into segments with ~;—the spacing is evenly divided between the text segments. With no modifiers, the leftmost text segment is left justified in the field, and the rightmost text segment right justified; if there is only one, as a special case, it is right justified. The : modifier causes spacing to be introduced before the first text segment; the @ modifier causes spacing to be added after the last. *Minpad*, default 0, is the minimum number of *padchar* (default space) padding characters to be output between each segment. If the total width needed to satisfy these constraints is greater than *mincol*, then *mincol* is adjusted upwards in *colinc* increments. *colinc* defaults to 1. *mincol* defaults to 0. For example,

```
(format nil "~10<foo~;bar~>")            =>  "foo    bar"
(format nil "~10:<foo~;bar~>")           =>  "  foo  bar"
(format nil "~10:@<foo~;bar~>")          =>  "  foo bar "
(format nil "~10<foobar~>")              =>  "    foobar"
(format nil "~10:<foobar~>")             =>  "    foobar"
(format nil "~10@<foobar~>")             =>  "foobar    "
(format nil "~10:@<foobar~>")            =>  "  foobar  "
(format nil "$~10,,,'*<~3f~>" 2.5902)   =>  "$******2.59"
```

Note that *text* may include format directives. The last example illustrates how the ~< directive can be combined with the ~f directive to provide more advanced control over the formatting of numbers.

Here are some examples of the use of ~^ within a ~< construct. ~^ is explained in detail below, however the general idea is that it eliminates the segment in which it appears and all following segments if there are no more arguments.

```
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo)
        =>  "            FOO"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar)
        =>  "FOO         BAR"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar 'baz)
        =>  "FOO   BAR   BAZ"
```

The idea is that if a segment contains a ~^, and **format** runs out of arguments, it just stops there instead of getting an error, and it as well as the rest of the segments are ignored.

If the first clause of a ~< is terminated with ~:; instead of ~;, then it is used in a special way. All of the clauses are processed (subject to ~^, of course), but the first one is omitted in performing the spacing and padding. When the padded result has been determined, then if it will fit on the current line of output, it is output, and the text for the first clause is discarded. If, however, the padded text will not fit on the current line, then the text segment for the first clause is output before the padded text. The first clause ought to contain a carriage return (~%). The first clause is always processed, and so any arguments it refers to will be used; the decision is whether to use the resulting segment of text, not whether to process the first clause. If the ~:; has a prefix parameter *n*, then the padded text must fit on the current line with *n* character positions to spare to avoid outputting the first clause's text. For example, the control string

```
"~%;; ~{~<~%;; ~1:; ~S~>~^,~}.~%"
```

can be used to print a list of items separated by commas, without breaking items over line boundaries, and beginning each line with ';; '. The prefix parameter 1 in ~1:; accounts for the width of the comma which will follow the justified item if it is not the last element in the list, or the period if it is. If ~:; has a second prefix parameter, then it is used as the width of the line, thus overriding the natural line width of the output stream. To make the preceding example use a line width of 50, one would write

```
"~%;; ~{~<~%;; ~1,50:; ~S~>~^,~}.~%"
```

If the second argument is not specified, then format sees whether the stream handles the :size-in-characters message. If it does, then format sends that message and uses the first returned value as the line length in characters. If it doesn't, format uses 72. as the line length.

Rather than using this complicated syntax, one can often call the function format:print-list (see page 495).

~> Terminates a ~<. It is undefined elsewhere.

~^ This is an escape construct. If there are no more arguments remaining to be processed, then the immediately enclosing ~{ or ~< construct is terminated. If there is no such enclosing construct, then the entire formatting operation is terminated. In the ~< case, the formatting *is* performed, but no more segments are processed before doing the justification. The ~^ should appear only at the *beginning* of a ~< clause, because it aborts the entire clause. ~^ may appear anywhere in a ~{ construct.

If a prefix parameter is given, then termination occurs if the parameter is zero. (Hence ~^ is the same as ~#^.) If two parameters are given, termination occurs if they are equal. If three are given, termination occurs if the second is between the other two in ascending order. Of course, this is useless if all the prefix parameters are constants; at least one of them should be a # or a V parameter.

If ~^ is used within a ~:{ construct, then it merely terminates the current iteration step (because in the standard case it tests for remaining arguments of the current step only); the next iteration step commences immediately. To terminate the entire iteration process, use ~:^.

~Q An escape to arbitrary user-supplied code. *arg* is called as a function; its arguments are the prefix parameters to ~Q, if any. *args* can be passed to the function by using the V prefix parameter. The function may output to *standard-output* and may look at the variables format:colon-flag and format:atsign-flag, which are t or nil to reflect the : and @ modifiers on the ~Q. For example,

```
(format t "~VQ" foo bar)
```

is a fancy way to say

```
(funcall bar foo)
```

and discard the value. Note the reversal of order; the V is processed before the Q.

~\ This begins a directive whose name is longer than one character. The name is terminated by another \ character. The following directives have names longer than one character and make use of the ~\ mechanism as part of their operation.

~\lozenged-string\
: This is like ~A except when output is to a window, in which case the argument is printed in a small font inside a lozenge.

~\lozenged-character\
: This is like ~C except when output is to a window, in which case the argument is printed in a small font inside a lozenge if it has a character name, even if it is a formatting character or graphic character.

~\date\
: This expects an argument that is a universal time (see page 776), and prints it as a date and time using **time:print-universal-date**.
Example:

```
        (format t "It is now ~\date\" (get-universal-time))
```

prints

```
        It is now Saturday the fourth of December, 1982; 4:00:32 am
```

~\time\
: This expects an argument that is a universal time (see page 776), and prints it in a brief format using **time:print-universal-time**.
Example:

```
        (format t "It is now ~\time\" (get-universal-time))
```

prints

```
        It is now 12/04/82 04:01:38
```

~\datime\
: This prints the current time and date. It does not use an argument. It is equivalent to using the ~\time\ directive with (time:get-universal-time) as argument.

~\time-interval\
: This prints a time interval measured in seconds using the function **time:print-interval-or-never**.
Example:

```
        (format t "It took ~\time-interval\." 3601.)
```

prints

```
        It took 1 hour 1 second.
```

You can define your own directives. How to do this is not documented here; read the code. Names of user-defined directives longer than one character may be used if they are enclosed in backslashes (e.g. ~4,3\GRAPH\).

(Note: **format** also allows *control-string* to be a list. If the list is a list of one element, which is a string, the string is simply printed. This is for the use of the **format:outfmt** function below. The old feature wherein a more complex interpretation of this list was possible is now considered obsolete; use **format:output** if you like using lists.)

A condition instance can also be used as the *control-string*. Then the :report operation is used to print the condition instance; any other arguments are ignored. This way, you can pass a condition instance directly to any function that normally expects a format string and arguments.

format:print-list *destination element-format list* &optional *separator start-line tilde-brace-options*

This function provides a simpler interface for the specific purpose of printing comma-separated lists with no list element split across two lines; see the description of the ~:; directive (page 492) to see the more complex way to do this within **format**. *destination* tells where to send the output; it can be **t**, **nil**, a **string-nconc**'able string, or a stream, as with **format**. *element-format* is a **format** control-string that tells how to print each element of *list*; it is used as the body of a ~{...~} construct. *separator*, which defaults to ", " (comma, space) is a string which goes after each element except the last. **format** control commands are not recommended in *separator*. *start-line*, which defaults to three spaces, is a **format** control-string that is used as a prefix at the beginning of each line of output, except the first. **format** control commands are allowed in *separator*, but they should not swallow arguments from *list*. *tilde-brace-options* is a string inserted before the opening '{'; it defaults to the null string, but allows you to insert colon and/or atsign. The line-width of the stream is computed the same way that the ~:; command computes it; it is not possible to override the natural line-width of the stream.

22.4.2 The Output Subsystem

The formatting functions associated with the **format:output** subsystem allow you to do formatted output using Lisp-style control structure. Instead of a directive in a **format** control string, there is one formatting function for each kind of formatted output.

The calling conventions of most of the formatting functions are similar. The first argument is usually the datum to be output. The second argument is usually the minimum number of columns to use. The remaining arguments are keyword arguments.

Most of the functions accept the keyword arguments *padchar*, *minpad* and *tab-period*. *padchar* is a character to use for padding. *minpad* is a minimum number of padding characters to output after the data. *tab-period* is the distance between allowable places to stop padding. To make the meaning of *tab-period* clearer, if the value of *tab-period* is 5, if the minimum size of the field is 10, and if the value of *minpad* is 2, then a datum that takes 9 characters is padded out to 15 characters. The requirement to use at least two characters of padding means it can't fit into 10 characters, and the *tab-period* of 5 means the next allowable stopping place is at 10+5 characters. The default values for *minpad* and *tab-period*, if they are not specified, are zero and one. The default value for *padchar* is space.

The formatting functions always output to ***standard-output*** and do not require an argument to specify the stream. The macro **format:output** allows you to specify the stream or a string, just as **format** does, and also makes it convenient to concatenate constant and variable output.

format:output *stream string-or-form...* *Macro*

Makes it convenient to intersperse arbitrary output operations with printing of constant strings. ***standard-output*** is bound to *stream*, and each *string-or-form* is processed in succession from left to right. If it is a string, it is printed; otherwise it is a form, which is evaluated for effect. Presumably the forms will send output to ***standard-output***.

If *stream* is written as **nil**, then the output is put into a string which is returned by **format:output**. If *stream* is written as **t**, then the output goes to the prevailing value of ***standard-output***. Otherwise *stream* is a form, which must evaluate to a stream.

Here is an example:

```
        (format:output t "FOO is " (prin1 foo) " now." (terpri))
```

Because **format:output** is a macro, what matters about *stream* is not whether it *evaluates* to **t** or **nil**, but whether it is actually written as **t** or **nil**.

format:outfmt *string-or-form...* *Macro*

Some system functions ask for a **format** control string and arguments, to be printed later. If you wish to generate the output using the formatted output functions, you can use **format:outfmt**, which produces a control argument that will eventually make **format print** the desired output (this is a list whose one element is a string containing the output). A call to **format:outfmt** can be used as the second argument to **ferror**, for example:

```
(ferror nil (format:outfmt "Foo is " (format:onum foo)
                           " which is too large"))
```

format:onum *number* &optional *radix minwidth* &key *padchar minpad tab-period signed commas*

Outputs *number* in base *radix*, padding to at least *minwidth* columns and obeying the other padding options specified as described above.

radix can be a number, or it can be :roman, :english, or :ordinal. The default *radix* is 10. (decimal).

If *signed* is non-nil, a + sign is printed if the number is positive. If *commas* is non-nil, a comma is printed every third digit in the customary way. These arguments are meaningful only with numeric radices.

format:ofloat *number* &optional *n-digits force-exponential-notation minwidth* &key *padchar minpad tab-period*

Outputs *number* as a floating point number using *n-digits* digits. If *force-exponential-notation* is non-nil, then an exponent is always used. *minwidth* and the padding options are interpreted as usual.

format:ostring *string* &optional *minwidth* &key *padchar minpad tab-period right-justify*

Outputs *string*, padding to at least *minwidth* columns if *minwidth* is not nil, and obeying the other padding options specified as described above.

Normally the data are left justified; any padding follows the data. If *right-justify* is non-nil, the padding comes before the data. The amount of padding is not affected.

The argument need not really be a string. Any Lisp object is allowed, and it is output with princ.

format:oprint *object* &optional *minwidth* &key *padchar minpad tab-period right-justify*

Prints *object*, any Lisp object, padding to at least *minwidth* columns if *minwidth* is not nil, and obeying the padding options specified as described above.

Normally the data are left justified; any padding follows the data. If *right-justify* is non-nil, the padding comes before the data. The amount of padding is not affected.

The printing of the object is done with prin1.

format:ochar *character* &optional *style top-explain minwidth* &key *padchar minpad tab-period*

Outputs *character* in one of three styles, selected by the *style* argument. *minwidth* and the padding options control padding as usual.

:read or nil	The character is printed using #\ or #/ so that it could be read back in.
:editor	Output is in the style of 'Meta-Rubout'. Non-printing characters, and the two printing characters Space and Altmode, are represented by their names. Other printing characters are printed directly.

:brief	Brief prefixes such as 'C-' and 'M-' are used, rather than 'Control-' or 'Meta-'. Also, character names are used only if there are meta bits present.
:lozenged	The output is the same as that of the :editor style, but If the character is not a graphic character or if it has meta bits, and the stream supports the :display-lozenged-string operation, that operation is used instead of :string-out to print the text. On windows this operation puts the character name inside a lozenge.
:sail	'*α*', '', etc. are used to represent Control and Meta, and shorter names for characters are also used when possible. See section 10.1.1, page 205.

top-explain is useful with the :editor, :brief and :sail styles. It says that any character that has to be typed using the Top or Greek keys should be followed by an explanation of how to type it. For example: '→ (Top-K)' or '*α* (Greek-a)'.

format:tab *mincol* &key *padchar minpad tab-period terpri unit*

Outputs padding at least until column *mincol*. It is the only formatting function that bases its actions on the actual cursor position rather than the width of what is being output. The padding options *padchar*, *minpad*, and *tab-period* are obeyed. Thus, at least the *minpad* number of padding characters are output even if that goes past *mincol*, and once past *mincol*, padding can only stop at a multiple of *tab-period* characters past *mincol*.

In addition, if the *terpri* option is t, then if column *mincol* is passed, format:tab starts a new line and indents it to *mincol*.

The *unit* option specifies the units of horizontal position. The default is to count in units of characters. If *unit* is specified as :pixel, then the computation (and the argument *mincol* and the *minpad* and *tab-period* options) are in units of pixels.

format:pad (*minwidth* &key *padchar minpad tab-period...*) *body...* *Macro*

format:pad is used for printing several items in a fixed amount of horizontal space, padding between them to use up any excess space. Each of the *body* forms prints one item. The padding goes between items. The entire format:pad always uses at least *minwidth* columns; any columns that the items don't need are distributed as padding between the items. If that isn't enough space, then more space is allocated in units controlled by the *tab-period* option until there is enough space. If it's more than enough, the excess is used as padding.

If the *minpad* option is specified, then at least that many pad characters must go between each pair of items.

Padding goes only between items. If you want to treat several actual pieces of output as one item, put a progn around them. If you want padding before the first item or after the last, as well as between the items, include a dummy item nil at the beginning or the end.

If there is only one item, it is right justified. One item followed by **nil** is left-justified. One item preceded and followed by **nil** is centered. Therefore, **format:pad** can be used to provide the usual padding options for a function that does not provide them itself.

format:plural *number singular* &optional *plural*

Outputs either the singular or the plural form of a word depending on the value of *number*. The singular is used if and only if *number* is 1. *singular* specifies the singular form of the word. **string-pluralize** is used to compute the plural, unless *plural* is explicitly specified.

It is often useful for *number* to be a value returned by **format:onum**, which returns its argument. For example:

```
(format:plural (format:onum n-frobs) " frob")
```

prints "1 frob" or "2 frobs".

format:breakline *linel print-if-terpri print-always...* *Macro*

Goes to the next line if there is not enough room for something to be output on the current line. The *print-always* forms print the text which is supposed to fit on the line. *linel* is the column before which the text must end. If it doesn't end before that column, then **format:breakline** moves to the next line and executes the *print-if-terpri* form before doing the *print-always* forms.

Constant strings are allowed as well as forms for *print-if-terpri* and *print-always*. A constant string is just printed.

To go to a new line unconditionally, simply call **terpri**.

Here is an example that prints the elements of a list, separated by commas, breaking lines between elements when necessary.

```
(defun pcl (list linel)
  (do ((l list (cdr l))) ((null l))
    (format:breakline linel "  "
      (princ (car l))
      (and (cdr l) (princ ", ")))))
```

22.5 Rubout Handling

The rubout handler is a feature of all interactive streams, that is, streams that connect to terminals. Its purpose is to allow the user to edit minor mistakes made during type-in. At the same time, it is not supposed to get in the way; input is to be seen by Lisp as soon as a syntactically complete form has been typed. The definition of 'syntactically complete form' depends on the function that is reading from the stream; for **read**, it is a Lisp expression.

Some interactive streams ('editing Lisp listeners') have a rubout handler that allows input to be edited with the full power of the ZWEI editor. (ZWEI is the general editor implementation on which Zmacs and ZMail are based.) Most windows have a rubout handler that apes ZWEI, implementing about twenty common ZWEI commands. The cold load stream has a simple rubout handler that allows just rubbing out of single characters, and a few simple commands like clearing the screen and erasing the entire input typed so far. All three kinds of rubout handler use the same protocol, which is described in this section. We also say a little about the most common of the three rubout handlers.
[Eventually some version of ZWEI will be used for all streams except the cold load stream]

The tricky thing about the rubout handler is the need for it to figure out when you are all done. The idea of a rubout handler is that you can type in characters, and they are saved up in a buffer so that if you change your mind, you can rub them out and type different characters. However, at some point, the rubout handler has to decide that the time has come to stop putting characters into the buffer and to let the function parsing the input, such as **read**, return. This is called *activation*. The right time to activate depends on the function calling the rubout handler, and may be very complicated (if the function is **read**, figuring out when one Lisp expression has been typed requires knowledge of all the various printed representations, what all currently-defined reader macros do, and so on). Rubout handlers should not have to know how to parse the characters in the buffer to figure out what the caller is reading and when to activate; only the caller should have to know this. The rubout handler interface is organized so that the calling function can do all the parsing, while the rubout handler does all the handling of editing commands, and the two are kept completely separate.

The basic way that the rubout handler works is as follows. When an input function that reads characters from a stream, such as **read** or **readline** (but not **tyi**), is invoked with a stream which has **:rubout-handler** in its **:which-operations** list, that function "enters" the rubout handler. It then goes ahead :tyi'ing characters from the stream. Because control is inside the rubout handler, the stream echoes these characters so the user can see what he is typing. (Normally echoing is considered to be a higher-level function outside of the province of streams, but when the higher-level function tells the stream to enter the rubout handler it is also handing it the responsibility for echoing.) The rubout handler is also saving all these characters in a buffer, for reasons disclosed in the following paragraph. When the parsing function decides it has enough input, it returns and control "leaves" the rubout handler. This is the easy case.

If the user types a rubout, a **throw** is done out of all recursive levels of **read**, reader macros, and so forth, back to the point where the rubout handler was entered. Also the rubout is echoed by erasing from the screen the character which was rubbed out. Now the **read** is tried over again, re-reading all the characters that have not been rubbed out, not echoing them this time. When the saved characters have been exhausted, additional input is read from the user in the usual fashion.

The effect of this is a complete separation of the functions of rubout handling and parsing, while at the same time mingling the execution of these two functions in such a way that input is always activated at just the right time. It does mean that the parsing function (in the usual case, **read** and all macro-character definitions) must be prepared to be thrown through at any time and should not have non-trivial side-effects, since it may be called multiple times.

If an error occurs while inside the rubout handler, the error message is printed and then additional characters are read. When the user types a rubout, it rubs out the error message as well as the character that caused the error. The user can then proceed to type the corrected expression; the input will be reparsed from the beginning in the usual fashion.

The rubout handler based on the ZWEI editor interprets control characters in the usual ZWEI way: as editing commands, allowing you to edit your buffered input.

The common rubout handler also recognizes a subset of the editor commands, including Rubout, Control-F and Meta-F and others. Typing Help while in the rubout handler displays a list of the commands. The kill and yank commands in the rubout handler use the same kill ring as the editor, so you can kill an expression in the editor and yank it back into a rubout handler with Control-Y, or kill an expression in the rubout handler with Control-K or Clear-input and yank it back in the editor. The rubout processor also keeps a ring buffer of most recent input strings (a separate ring for each stream), and the commands Control-C and Meta-C retrieve from this ring just as Control-Y and Meta-Y do from the kill ring.

When not inside the rubout handler, and when typing at a program that uses control characters for its own purposes, control characters are treated the same as ordinary characters.

Some programs such as the debugger allow the user to type either a control character or an expression. In such programs, you are really not inside the rubout handler unless you have typed the beginning of an expression. When the input buffer is empty, a control character is treated as a command for the program (such as, Control-C to continue in the debugger); when there is text in the rubout handler buffer, the same character is treated as a rubout handler command. Another consequence of this is that the message you get by typing Help varies, being either the rubout handler's documentation or the debugger's documentation.

To write a parsing function that reads with rubout handling, use **with-input-editing**.

with-input-editing (*stream options*) *body...* *Macro*

Invokes the rubout handler on *stream*, if *stream* supports it, and then executes *body*. *body* is executed in any case, within the rubout handler if possible. **rubout-handler** is non-nil while in *body* if rubout handling is in use.

options are used as the rubout handler options. If already within an invocation of the rubout handler, *options* are appended to the front of the options already in effect. This happens if a function which reads input using **with-input-editing**, such as **read** or **readline**, is called from the body of another **with-input-editing**. The **:norecursive** option can be used to cause the outer set of options to be completely ignored even when not overridden by new ones.

body's values are returned by with-input-editing. *body* ought to read input from *stream* and return a Lisp object that represents the input. It should have no nontrivial side effects aside from reading input from *stream* structure, as it may be aborted at any time it reads input and may be executed over and over.

If the :full-rubout option is specified, and the user types some input and rubs it all out, the with-input-editing form returns immediately. See :full-rubout, below.

If a preemptive command is input by the user, with-input-editing returns immediately with the values being as specified below under the :command and :preemptable options. *body* is aborted from its call to the :tyi operation, and the input read so far remains in the rubout handler editing buffer to be read later.

rubout-handler *Variable*
If control is inside the rubout handler in this process, the value is the stream on which rubout handling is being done. Otherwise, the value is nil.

:rubout-handler *options function* &rest *args* *Operation on streams*
Invokes the rubout handler on the stream, with *options* as the options, and parses by applying *function* to *args*. with-input-editing uses this operation.

:read-bp *Operation on streams*
This operation may be used only from within the code for parsing input from this stream inside the rubout handler. It returns the index within the rubout handler buffer which parsing has reached.

:force-rescan *Operation on streams*
This operation may be used only from within the code for parsing input from this stream inside the rubout handler. It causes parsing to start again immediately from the beginning of the buffer.

:rescanning-p *Operation on streams*
This operation may be used only from within the code for parsing input from this stream inside the rubout handler. It returns t if parsing is now being done on input already in the buffer, nil if parsing has used up all the buffered input and the next character parsed will come from the keyboard.

Each option in the list of rubout handler options consists of a list whose first element is a keyword and whose remaining elements are the arguments of that keyword. Note that this is not the same format as the arguments to a typical function that takes keyword arguments; rather this is an alist of options. The standard options are:

(:activation *fn args...*)
Activate if certain characters are typed in. When the user types an activation character, the rubout handler moves the editing pointer immediately to the end of the buffer and inserts the activation character. This immediately causes the parsing function to begin rescanning the input.

fn is used to test characters for being activators. It is called with an input character as the first arg (possibly a fixnum, possibly a character object) and *args* as additional args. If *fn* returns non-nil, the character is an activation. *fn* is not called for blips.

After the parsing function has read the entire contents of the buffer, it sees the activation character as a blip (:activation *char numeric-arg*) where *char* is the character that activated and *numeric-arg* is the numeric arg that was pending for the next rubout handler command. Normally the parsing function will return at this point. Then the activation character does not echo. But if the parsing function continues to read input, the activation character echoes and is inserted in the buffer.

(:do-not-echo *chars...*)
Poor man's activation characters. Like :activation except that the characters that should activate are listed explicitly, and the character itself is returned to the parsing function rather than a blip.

(:full-rubout *val*)
If the user rubs out all the characters he typed, then control is returned from the rubout handler immediately. Two values are returned; the first is nil and the second is *val*. (If the user doesn't rub out all the characters, then the rubout handler propagates multiple values back from the function that it calls, as usual.) In the absence of this option, the rubout handler would simply wait for more characters to be typed in and would ignore any additional rubouts.

This is how the debugger knows to remove Eval: from the screen if you type the beginning of a form and rub it all out.

(:pass-through *char1 char2...*)
The characters *char1*, *char2*, etc. are not to be treated as special by the rubout handler. They are read as input by the parsing function. If the parsing function does not return, they can be rubbed out. This works only for characters with no modifier bits.

(:preemptable *value*)
Makes all blips read as input by the rubout handler act as preemptive commands. If this option is specified, the rubout handler returns immediately when it reads a blip. It returns two values: the blip that was read, and *value*. The parsing function is not allowed to finish parsing up to a delimiter; instead, any buffered input remains in the buffer for the next time input is done. In the mean time, the preemptive command character can be processed by the command loop.

While this applies to all blips, the blips which it is probably intended for are mouse blips.

(:command *fn args...*)
Makes certain characters preemptive commands. A preemptive command returns instantly to the caller of the :rubout-handler operation, regardless

of the input in the buffer. It returns two values: a list (:command *char numeric-arg*) and the keyword :command. The parsing function is not allowed to finish parsing up to a delimiter; instead, any buffered input remains in the buffer for the next time input is done. In the mean time, the preemptive command character can be processed by the command loop.

The test for whether a character should be a preemptive command is done using *fn* and *args* just as in :activation.

(:editing-command (*char doc*)...)

Defines editing commands to be executed by the parsing function itself. This is how qsend implements the Control-Meta-Y command. Each *char* is such a command, and *doc* says what it does. (*doc* is printed out by the rubout handler's Help command.) If any of these characters is read by the rubout handler, it is returned immediately to the parsing function regardless of where the editing pointer is in the buffer. (Normal inserted text is not returned immediately when read unless the editing pointer is at the end of the buffer.)

The parsing function should not regard these characters as part of the input. There are two reasonable things that the parsing function can do when it receives one of the editing command characters: print some output, or force some input.

If it prints output, it should invoke the :refresh-rubout-handler operation afterward before the next :tyi. This causes the rubout handler to redisplay so that the input being edited appears after the output that was done.

If the parsing function forces input, the input is read by the rubout handler. This can be used to modify the buffered input. qsend's Control-Meta-Y command works by forcing the yanked text as input. There is no way to act directly on the buffered input because different implementations of the rubout handler store it in different ways.

(:prompt *function*)
(:reprompt *function*)

When it is time for the user to be prompted, *function* is called with two arguments. The first is a stream it may print on; the second is the character which caused the need for prompting, e.g. #\clear-input or #\clear-screen, or nil if the rubout handler was just entered.

The difference between :prompt and :reprompt is that the latter does not call the prompt function when the rubout handler is first entered, but only when the input is redisplayed (e.g. after a screen clear). If both options are specified then :reprompt overrides :prompt except when the rubout handler is first entered.

function may also be a string. Then it is simply printed.

If the rubout handler is exited with an empty buffer due to the **:full-rubout** option, whatever prompt was printed is erased.

(:initial-input *string*)
: Pretends that the user typed *string*. When the rubout handler is entered, *string* is typed out. The user can input more characters or rub out characters from it.

(:initial-input-index *index*)
: Positions the editing pointer initially *index* characters into the initial input string. Used only in company with with **:initial-input**.

(:no-input-save t)
: Don't save this batch of input in the input history when it is done. For example, **yes-or-no-p** specifies this option.

(:norecursive t)
: If this invocation of the rubout handler is within another one, the options specified in the previous call should be completely ignored during this one. Normally, individual options specified this time override the previous settings for the same options, but any of the previous options not individually overridden are still in effect.

Rubout handlers handle the condition **sys:parse-error** if it is signaled by the parsing function. The handling consists of printing the error message, waiting for the user to rub out, erasing the error message, and parsing the input again. All errors signaled by a parsing function that signify that the user's input was syntactically invalid should have this condition name. For example, the errors **read** signals have condition name **sys:parse-error** since it is is a consequence of **sys:read-error**.

sys:parse-error (error) *Condition*

The condition name for syntax errors in input being parsed.

The compiler handles **sys:parse-error** by proceeding with proceed-type **:no-action**. All signalers of **sys:parse-error** should offer this proceed type, and respond to its use by continuing to parse, ignoring the invalid input.

sys:parse-ferror *format-string* &rest *args*

Signals a **sys:parse-error** error, using *format-string* and *args* to print the error message. The proceed-type **:no-action** is provided, and if a handler uses it, this function returns **nil**.

23. Expression Input and Output

People cannot deal directly with Lisp objects, because the objects live inside the machine. In order to let us get at and talk about Lisp objects, Lisp provides a representation of objects in the form of printed text; this is called the *printed representation*. This is what you have been seeing in the examples throughout this manual. Functions such as **print**, **prin1**, and **princ** take a Lisp object and send the characters of its printed representation to a stream. These functions (and the internal functions they call) are known as the *printer*. The **read** function takes characters from a stream, interprets them as a printed representation of a Lisp object, builds a corresponding object, and returns it. It and related functions are known as the *reader*. (Streams are explained in section 22.3, page 459.)

For the rest of the chapter, the phrase 'printed representation' is abbreviated as 'p.r.'

23.1 What the Printer Produces

The printed representation of an object depends on its type. In this section, we consider each type of object and explain how it is printed. There are several variables which you can set before calling the printer to control how certain kinds of objects print. They are mentioned where relevant in this section and summarized in the following section, but one of them is so important it must be described now. This is the *escaping* feature, controlled by the value of ***print-escape***.

Escaping means printing extra syntactical delimiters and *escape characters* when necessary to avoid ambiguity. Without escaping, a symbol is printed by printing the contents of its name; therefore, the symbol whose name consists of the three characters 1, . and 5 prints just like the floating point number 1.5. Escaping causes the symbol to print as |1.5| to differentiate the two. | is a kind of escape character; see page 516 for more information on escape characters and what they mean syntactically.

Escaping also involves printing package prefixes for symbols, printing double-quotes or suitable delimiters around the contents of strings, pathnames, host names, editor buffers, condition objects, and many other things. For example, without escaping, the pathname SYS: SYS; QCP1 LISP prints as exactly those characters. The string with those contents prints indistinguishably. With escaping, the pathname prints as

```
    #⊂FS:LOGICAL-PATHNAME "SYS: SYS; QCP1 LISP"⊃
```

and the string prints as "SYS: SYS; QCP1 LISP".

The non-escaped version is nicer looking in general, but if you give it to **read** it won't do the right thing. The escaped version is carefully set up so that **read** will be able to read it in. Printing with escaping is useful in writing expressions into files. Printing without escaping is useful when constructing messages for the user. However, when the purpose of a message printed for the user is to *mention* an object, the object should be printed with escaping:

```
    Your output is in the file SYS: SYS; QCP1 QFASL.
```
vs
```
    Expected pathname properties missing from
     #⊂FS:LOGICAL-PATHNAME "SYS: SYS; QCP1 LISP"⊃.
```

The printed representation of an object also may depend on whether Common Lisp syntax is in use. Common Lisp syntax and traditional Zetalisp syntax are incompatible in some aspects of their specifications. In order to print objects so that they can be read back in, the printer needs to know which syntax rules the reader will use. This decision is based on the current readtable: the value of *readtable* at the time printing is done.

Now we describe how each type of object is standardly printed.

Integers:

For an integer (a fixnum or a bignum): the printed representation consists of

* a possible radix prefix
* a minus sign, if the number is negative
* the representation of the number's absolute value
* a possible radix suffix.

The radix used for printing the number's absolute value is found as the value of *print-base*. This should be either a positive fixnum or a symbol with an si:princ-function property. In the former case, the number is simply printed in that radix. In the latter case, the property is called as a function with two arguments, minus the absolute value of the number, and the stream to print on. The property is responsible for all printing. If the value of *print-base* is unsuitable, an error is signaled.

A radix prefix or suffix is used if either *nopoint is nil and the radix used is ten, or if *nopoint is non-nil and *print-radix* is non-nil. For radix ten, a period is used as the suffix. For any other radix, a prefix of the form #*radix*r is used. A radix prefix or suffix is useful to make sure that read parses the number using the same radix used to print it, or for reminding the user how to interpret the number.

Ratios:

The printed representation of a ratio consists of

* a possible radix prefix
* a minus sign, if the number is negative
* the numerator
* a ratio delimiter
* the denominator

If Common Lisp syntax is in use, the ratio delimiter is a slash (/). If traditional syntax is in use, backslash (\) is used. The numerator and denominator are printed according to ***print-base***.

The condition for printing a radix prefix is the same as for integers, but a prefix **#10r** is used to indicate radix ten, rather than a period suffix.

Floating Point Numbers:

* a minus sign, if the number is negative
* one or more decimal digits
* a decimal point
* one or more decimal digits
* an exponent, if the number is small enough or large enough to require one. The exponent, if present, consists of
    * a delimiter, the letter **e**, **s** or **f**
    * a minus sign, if the exponent is negative

      one to three decimal digits

The number of digits printed is just enough to represent all the significant mantissa bits the number has. Feeding the p.r. of a float back to the reader is always supposed to produce an equal float. Floats are always printed in decimal; they are not affected by escaping or by ***print-base***, and there are never any radix prefixes or suffixes.

The Lisp Machine supports two floating point number formats. At any time, one of them is the default; this is controlled by the value of ***read-default-float-format***. When a floating point number whose format is *not* currently the default is printed, it must be printed with an exponent so that the exponent delimiter can specify the format. The exponent is introduced in this case by **f** or **s** to specify the format. To the reader, **f** specifies **single-float** format and **s** specifies **short-float** format.

A floating point number of the default format is printed with no exponent if this looks nice; namely, if this does not require too many extra zeros to be printed before or after the decimal point. Otherwise, an exponent is printed and is delimited with **e**. To the reader, **e** means 'use the default format'.

Normally the default float format is **single-float**. Therefore, the printer may print full size floats without exponents or with **e** exponents, but short floats are always printed with exponents introduced by **s** so as to tell the reader to make a short float.

Complex Numbers:

The traditional printed representation of a complex number consists of

* the real part
* a plus sign, if the imaginary part is positive
* the imaginary part
* the letter i, printed in lower case

If the imaginary part is negative, the + is omitted since the initial - of the imaginary part serves to separate it from the real part.

In Common Lisp syntax, a complex number is printed as #C(*realpart imagpart*); for example, #C(5 3). Common Lisp inexplicably does not allow the more natural 5+3i syntax.

The real and imaginary parts are printed individually according to the specifications above.

Symbols:

If escaping is off, the p.r. is simply the successive characters of the print-name of the symbol. If escaping is on, two changes must be made. First, the symbol might require a package prefix in order that **read** work correctly, assuming that the package into which **read** will read the symbol is the one in which it is being printed. See the chapter on packages (chapter 27, page 636) for an explanation of the package name prefix. If the symbol is one which would have another symbol substituted for it if printed normally and read back, such as the symbol **member** printed using Common Lisp syntax which would be replaced with **cli:member** if read in thus, it is printed with a package prefix (e.g., **global:member**) to make it read in properly. See page 519 for more information on this.

If the symbol is uninterned, #: is printed instead of a package prefix, provided ***print-gensym*** is non-**nil**.

Secondly, if the p.r. would not read in as a symbol at all (that is, if the print-name looks like a number, or contains special characters), then escape characters are added so as to suppress the other reading. Two kinds of escape characters may be used: single-character escapes and multiple escapes. A single-character escape can be used in front of a character to overrule its special syntactic meaning. Multiple escapes are used in pairs, and all the characters between the pair have their special syntactic meanings suppressed *except single-character escapes*. If the symbol name contains escape characters, they are escaped with single-character escapes. If the symbol name contains anything else problematical, a pair of multiple escape characters are printed around it.

The single-character and multiple escape characters are determined by the current readtable. Standardly the multiple escape character is vertical bar (|), in both traditional and Common Lisp syntax. The single-character escape character is slash (/) in traditional syntax and backslash (\) in Common Lisp syntax.

```
FOO          ;typical symbol, name composed of upper case letters
A/|B         ;symbol with a vertical bar in its name
|Symbol with lower case and spaces in its name|
|One containing slash (//) and vertical bar (/|) also|
```

Except when multiple escape characters are printed, any upper case letters in the symbol's name may be printed as lower case, according to the value of the variable *print-case*. This is true whether escaping is enabled or not. See the next section for details.

Conses:

The p.r. for conses tends to favor *lists*. It starts with an open-parenthesis. Then the car of the cons is printed and the cdr of the cons is examined. If it is nil, a close-parenthesis is printed. If it is anything else but a cons, space dot space followed by that object is printed. If it is a cons, we print a space and start all over (from the point *after* we printed the open-parenthesis) using this new cons. Thus, a list is printed as an open-parenthesis, the p.r.'s of its elements separated by spaces, and a close-parenthesis. This is how the printer produces representations such as (a b (foo bar) c) in preference to synonymous forms such as (a . (b . ((foo . (bar . nil)) . (c . nil)))).

The following additional feature is provided for the p.r. of conses: as a list is printed, print maintains the length of the list so far and the depth of recursion of printing lists. If the length exceeds the value of the variable *print-length*, print terminates the printed representation of the list with an ellipsis (three periods) and a close-parenthesis. If the depth of recursion exceeds the value of the variable *print-level*, then the character # is printed instead of the list. These two features allow a kind of abbreviated printing that is more concise and suppresses detail. Of course, neither the ellipsis nor the # can be interpreted by read, since the relevant information is lost. In Common Lisp read syntax, either one causes read to signal an error.

If *print-pretty* is non-nil, conses are given to the grinder to print.

If *print-circle* is non-nil, a check is made for cars or cdrs that are circular or shared structure, and any object (except for an interned symbol) already mentioned is replaced by a *#n#* label reference. See page 524 for more information on them.

```
(let ((*print-circle* t))
  (prin1 (circular-list 3 4)))
```

prints

```
#1= (3 4 . #1#)
```

Character Objects:

When escaping is off, a character object is printed by printing the character itself, with no delimiters.

In Common Lisp syntax, a character object is printed with escaping as *#font\character-or-name*. *font* is the character's font number, in decimal, or is omitted if zero. *character-or-name* begins with prefixes for any modifier bits (control, meta, etc.) present in the character, each followed by a hyphen. Then comes a representation of the character sans font and modifier bits. If this reduced character is a graphic character, it represents itself. Otherwise, it certainly has a standard name; the name is used. If a graphic characters has special syntactic properties (such as whitespace, paretheses, and macro characters) and modifier bit prefixes have been printed then a single-character escape character is printed before it.

In traditional syntax, the p.r. is the similar except that the \ is replaced by ≠/.

Strings:

If escaping is off, the p.r. is simply the successive characters of the string. If escaping is on, double-quote characters ("") are printed surrounding the contents, and any single-character escape characters or double-quotes inside the contents are preceded by single-character escapes. If the string contains a Return character followed by an open parenthesis, a single-character escape is printed before the open parenthesis. Examples:

```
"Foo"
"/"Foo/", he said."
```

Named Structures:

If the named structure type symbol has a **named-structure-invoke** property, the property is called as a function with four arguments: the symbol **:print-self**, the named structure itself, the stream to print on, and the current *depth* of list structure (see below). It is this function's responsibility to output a suitable printed representation to the stream. This allows a user to define his own p.r. for his named structures; more information can be found in the named structure section (see page 390). Typically the printed representation used starts with either **#<** if it is not supposed to be readable or **#⊂** (see page 527) if it is supposed to be readable.

If the named structure symbol does not have a **named-structure-invoke** property, the printed-representation depends on whether escaping is in use. If it is, **#s** syntax is used:

#s (*named-structure-symbol*
    component value
    component value
    ...)

Named structure component values are checked for circular or shared structure if ***print-circle*** is **non-nil**.

If escaping is off, the p.r. is like that used for miscellaneous data-types: **#<**, the named structure symbol, the numerical address of the structure, and **>**.

Other Arrays:

If ***print-array*** is non-nil, the array is printed in a way which shows the elements of the array. Bit vectors use #* syntax, other vectors use #(. . .) syntax, and arrays of rank other than one use #*n*a(. . .) syntax. The printed representation does not indicate the array type (that is, what elements it is allowed to contain). If the printed representation is read in, a general array (array type **art-q**) is always created. See page 523 for more information on these syntaxes. Examples:

```
(vector 1 2 5) => #(1 2 5)
(make-array '(2 4) :initial-element t) => #2a((t t t t) (t t t t))
```

Vector and array groupings count like list groupings in maintaining the depth value that is compared with ***print-level*** for cutting off things that get too deep. More than ***print-length*** elements in a given vector or array grouping level are cut off with an ellipsis just like a list that is so long.

Array elements are checked for circular or shared structure if ***print-circle*** is non-**nil**.

If ***print-array*** is **nil**, the p.r. starts with #<. Then the **art-** symbol for the array type is printed. Next the dimensions of the array are printed, separated by hyphens. This is followed by a space, the machine address of the array, and a >, as in #<ART-COMPLEX-FLOAT-3-6 34030451>.

Instances and Entities:

If the object says it can handle the **:print-self** message, that message is sent with three arguments: the stream to print to, the current *depth* of list structure (see below), and whether escaping is enabled. The object should print a suitable p.r. on the stream. See chapter 21, page 401 for documentation on instances. Most such objects print like "any other data type" below, except with additional information such as a name. Some objects print only their name when escaping is not in effect (when **princ**'ed). Some objects, including pathnames, use a printed representation that begins with #⊂, ends with ⊃, and contains sufficient information for the reader to reconstruct an equivalent object. See page 527. If the object cannot handle **:print-self**, it is printed like "any other data type".

Any Other Data Type:

The printed representation starts with #< and ends with >. This sort of printed representation cannot be read back in. The #< is followed by the **dtp-** symbol for this datatype, a space, and the octal machine address of the object. The object's name, if one can be determined, often appears before the address. If this style of printed representation is being used for a named structure or instance, other interesting information may appear as well. Finally a greater-than sign (>) is printed in octal. Examples:

```
#'equal => #<DTP-U-ENTRY EQUAL 410>
(value-cell-location nil) => #<DTP-LOCATIVE 1>
```

Including the machine address in the p.r. makes it possible to tell two objects of this kind apart without explicitly calling **eq** on them. This can be very useful during debugging. It is important to know that if garbage collection is turned on, objects will occasionally be moved, and therefore their octal machine addresses will be changed. It is best to shut off garbage collection temporarily when depending on these numbers.

Printed representations that start with '#<' can never be read back. This can be a problem if, for example, you are printing a structure into a file with the intent of reading it in later. The following feature allows you to make sure that what you are printing may indeed be read with the reader.

si:print-readably *Variable*

When si:print-readably is bound to t, the printer signals an error if there is an attempt to print an object that cannot be interpreted by read. When the printer sends a :print-self or a :print message, it assumes that this error checking is done for it. Thus it is possible for these messages *not* to signal an error, if they see fit.

si:printing-random-object (*object stream . keywords*) &body *body* *Macro*

The vast majority of objects that define :print-self messages have much in common. This macro is provided for convenience so that users do not have to write out that repetitious code. It is also the preferred interface to si:print-readably. With no keywords, si:printing-random-object checks the value of si:print-readably and signals an error if it is not nil. It then prints a number sign and a less-than sign, evaluates the forms in *body*, then prints a space, the octal machine address of the object and a greater-than sign. A typical use of this macro might look like:

```
(si:printing-random-object (ship stream :typep)
  (tyo #\space stream)
  (prin1 (ship-name ship) stream))
```

This might print #<ship "ralph" 23655126>.

The following keywords may be used to modify the behaviour of si:printing-random-object:

:no-pointer	This suppresses printing of the octal address of the object.
:type	This prints the result of (type-of *object*) after the less-than sign. In the example above, this option could have been used instead of the first two forms in the body.

sys:print-not-readable (error) *Condition*

This condition is signaled by si:print-readably when the object cannot be printed readably.

The condition instance supports the operation :object, which returns the object that was being printed.

If you want to control the printed representation of some object, usually the right way to do it is to make the object an array that is a named structure (see page 390), or an instance of a flavor (see chapter 21, page 401). However, occasionally it is desirable to get control over all printing of objects, in order to change, in some way, how they are printed. If you need to do this, the best way to proceed is to customize the behavior of si:print-object (see page 543), which is the main internal function of the printer. All of the printing functions, such as **print** and **princ**, as well as **format**, go through this function. The way to customize it is by using the "advice" facility (see section 30.10, page 742).

23.2 Options that Control Printing

Several special variables are defined by the system for the user to set or bind before calling print or other printing functions. Their values, as set up by the user, control how various kinds of objects are printed.

print-escape *Variable*

Escaping is done if this variable is non-nil. See the previous section for a description of the many effects of escaping. Most of the output functions bind this variable to t or to nil, so you rarely use the variable itself.

print-base *Variable*
base *Variable*

The radix to use for printing integers and ratios. The value must be either an integer from 2 to 36 or a symbol with a valid si:princ-function property, such as :roman or :english.

The default value of *print-base* is ten. In input from files, the Base attribute (see section 25.5, page 594) controls the value of *print-base* (and of *read-base*).

The synonym base is from Maclisp.

print-radix *Variable*

If non-nil, integers and ratios are output with a prefix or suffix indicating the radix used to print them. For integers and radix ten, a period is printed as a suffix. Otherwise, a prefix such as #x or #3r is printed. The default value of *print-radix* is nil.

***nopoint** *Variable*

If the value of *nopoint is nil, a trailing decimal point is printed when a fixnum is printed out in base 10. This allows the numbers to be read back in correctly even if *read-base* is not 10 at the time of reading. The default value of *nopoint is t. *nopoint has no effect if *print-radix* is non-nil.

*nopoint exists for Maclisp compatibility. But to get truly compatible behavior, you must set *nopoint to nil (and, by default, base and ibase to eight).

print-circle *Variable*

If non-nil, the printer recognizes circular and shared structure and prints it using #*n*= labels so that it has a finite printed representation (which can be read back in). The default is nil, since t makes printing slower. See page 524 for information on the #*n*= construct.

print-pretty *Variable*

If non-nil, the printer actually calls grind-top-level so that it prints extra whitespace for the sake of formatting. The default is nil.

print-gensym *Variable*

If non-nil, uninterned symbols are printed with the prefix #: to mark them as such (but only when *print-escape* is non-nil). The prefix causes the reader to construct a similar uninterned symbol when the expression is read. If nil, no prefix is used for uninterned symbols. The default is t.

print-array *Variable*

If non-nil, non-string arrays are printed using the #(...), #* or #*n*a(...) syntax so that you can see their contents (and so that they can be read back in). If nil, such arrays are printed using #<...> syntax and do not show their contents. The default is nil. The printing of strings is not affected by this variable.

print-case *Variable*

Controls the case used for printing upper-case letters in the names of symbols. Its value should be :upcase, :downcase or :capitalize. These mean, respectively, to print those letters as upper case, to print them as lower case, or to capitalize each word (see string-capitalize, page 213). Any lower case letters in the symbol name are printed as lower case and escaped suitably; this flag does not affect them. Note that the case used for printing the upper case letters has no effect on reading the symbols back in, since they are case-converted by read. Any upper case letters that happen to be escaped are always printed in upper case.

```
(dolist (*print-case* '(:upcase :downcase :capitalize))
  (prin1-then-space 'foo)
  (prin1-then-space '|Foo|))
```

prints FOO |Foo| foo |Foo| Foo |Foo| .

print-level *Variable*
prinlevel *Variable*

print-level can be set to the maximum number of nested lists that can be printed before the printer gives up and just prints a # instead of a list element. If it is nil, which it is initially, any number of nested lists can be printed. Otherwise, the value of *print-level* must be a fixnum. Example:

```
(let ((*print-level* 2))
  (prin1 '(a (b (c (d e))))))
```

prints (a (b #)).

The synonym prinlevel is from Maclisp.

print-length *Variable*
prinlength *Variable*

print-length can be set to the maximum number of elements of a list that can be printed before the printer gives up and prints an ellipsis (three periods). If it is nil, which it is initially, any length list may be printed. Example:

```
(let ((*print-length* 3))
  (prin1 '((a b c d) #(e f g h) (i j k l) (m n o p))))
```

prints ((a b c ...) #(e f g ...) (i j k ...) ...).

The synonym **prinlength** is from Maclisp.

23.3 What The Reader Accepts

The purpose of the reader is to accept characters, interpret them as the p.r. of a Lisp object, and create and return such an object. The reader cannot accept everything that the printer produces; for example, the p.r.'s of compiled code objects, closures, stack groups, etc., cannot be read in. However, it has many features that are not seen in the printer at all, such as more flexibility, comments, and convenient abbreviations for frequently-used unwieldy constructs.

This section shows what kind of p.r.'s the reader understands, and explains the readtable, reader macros, and various features provided by **read**.

The syntax specified for Common Lisp is incompatible with the traditional Zetalisp syntax. Therefore, the Lisp Machine supports both traditional and Common Lisp syntax, but **read** must be told in advance which one to use. This is controlled by the choice of readtable (see section 23.6, page 535). When reading input from a file, the Lisp system chooses the syntax according to the file's attribute list: Common Lisp syntax is used if the **Common Lisp** attribute is present (see section 25.5, page 594).

The main difference between traditional and Common Lisp syntax is that traditionally the single-character escape is slash (**/**), whereas in Common Lisp syntax it is backslash (****). Thus, the division function which in traditional syntax is written **//** is written just **/** in Common Lisp syntax. The other differences are obscure and are mentioned below where they occur.

In general, the reader operates by recognizing tokens in the input stream. Tokens can be self-delimiting or can be separated by delimiters such as whitespace. A token is the p.r. of an atomic object such as a symbol or number, or a special character such as a parenthesis. The reader reads one or more tokens until the complete p.r. of an object has been seen, then constructs and returns that object.

Escape characters can be used to suppress the special syntactic significance of any character, including **:**, **Space**, **(** or **"**. There are two kinds of escape character: the *single-character escape* (**/** in traditional syntax, **** in Common Lisp syntax) suppresses the significance of the immediately following character; *multiple escapes* (vertical bar, **|**) are used in pairs, and suppress the special significance of all the characters except escapes between the pair. Escaping a character causes it to be treated as a token constituent and causes the token containing it to be read as a symbol. For example, **(12 5 x)** represents a list of three elements, two of which are integers, but **(/12 5/ x)** or **(|15| |5 X|)** represents a list of two elements, both symbols. Escaping also prevents conversion of letters to upper case, so that **|x|** is the symbol whose print name contains a lower-case **x**.

The circle-cross (**⊗**) character an *octal escape character* which may be useful for including weird characters in the input. The next three characters are read and interpreted as an octal number, and the character whose code is that number replaces the circle-cross and the digits in the input stream. This character is always treated as a token constituent and forces the token to be read as a symbol. **⊗** is allowed in both traditional and Common Lisp syntax, but it is not valid Common Lisp.

Integers:

The reader understands the p.r.'s of integers in a way more general than is employed by the printer. Here is a complete description of the format for integers.

Let a *simple integer* be a string of digits, optionally preceded by a plus sign or a minus sign, and optionally followed by a trailing decimal point. A simple integer is interpreted by **read** as an integer. If the trailing decimal point is present, the digits are interpreted in decimal radix; otherwise, they are considered as a number whose radix is the value of the variable ***read-base***.

read-base *Variable*
ibase *Variable*

The value of **ibase** or ***read-base*** is an integer between 2 and 36 that is the radix in which integers and ratios are read. The initial value of is ten. For input from files or editor buffers, the **Base** attribute specifies the value to be used (see section 25.5, page 594); if it is not given, the ambient value is used.

The synonym **ibase** is from Maclisp.

If the input radix is greater than ten, letters starting with **a** are used as additional "digits" with values ten and above. For example, in radix 16, the letters **a** through **f** are digits with values ten through 15. Alphabetic case is not significant. These additional digits can be used wherever a simple integer is expected and are parsed using the current input radix. For example, if ***read-base*** is 16 then `ff` is recognized as an integer (255 decimal). So is **10e5**, which is a float when ***read-base*** is ten.

Traditional syntax also permits a simple integer, followed by an underscore (_) or a circumflex (^), followed by another simple integer. The two simple integers are interpreted in the usual way; the character in between indicates an operation that is then performed on the two integers. The underscore indicates a binary "left shift"; that is, the integer to its left is doubled the number of times indicated by the integer to its right. The circumflex multiplies the integer to its left by ***read-base*** the number of times indicated by the integer to its right. (The second simple integer is not allowed to have a leading minus sign.) Examples: **3_2** means **12** and **645^3** means **645000**.

Here are some examples of valid representations of integers to be given to **read**:

```
4
23456.
-546
+45^+6       ;means 45000000
2_11         ;4096
72361356126536125376512375126535123712635
-123456789.
105_1000     ;(ash 105 1000) has this value.
105_1000.
```

Floating Point Numbers:

Floats can be written with or without exponent. The syntax for a float without exponent is an optional plus or minus sign, optionally some digits, a decimal point, and one or more digits. A float with exponent consists of a simple integer or a float without exponent, followed by an exponent delimiter (a letter) and a simple integer (the exponent itself) which is the power of ten by which the number is to be scaled. The exponent may not have a trailing decimal point. Both the mantissa and the exponent are always interpreted in base ten, regardless of the value of *read-base*.

Only certain letters are allowed for delimiting the exponent: e, s, f, d and l. The case of the letter is not significant. s specifies that the number should be a short float; f, that it should be a full-size float. d or l are equivalent to f; Common Lisp defines them to mean 'double float' or 'long float', but the Lisp Machine does not support anything longer than a full-size float, so it regards d and l as synonymous with f. e tells the reader to use the current default format, whatever it may be, as specified by the value of *read-default-float-format*.

read-default-float-format *Variable*

The value is the type for read to produce by default for floats whose precise type is not specified by the syntax. The value should be either global:small-float or global:single-float, these being the only distinct floating formats that the Lisp Machine has. The default is single-float, to make full-size floats.

Here are some examples of printed-representations that always read as full-size floats:

```
6.03f23   1F-9      1.f3      3d6
```

Here are some examples of printed-representations that always read as short floats:

```
0s0       1.5s9     -42S3     1.s5
```

These read as floats or as a short floats according to *read-default-float-format*:

```
0.0       1.5       14.0      0.01
.707      -.3       +3.14159  6.03e23
1E-9      1.e3
```

Rationals:

The syntax for a rational is an integer, a ratio delimiter, and another integer. The integers may not include the ^ and _ scaling characters or decimal points, and only the first one may have a sign. The ratio delimiter is backslash (\) in traditional syntax, slash (/) in Common Lisp syntax. Here are examples:

```
1\2       -100000000000000\3    80\10    traditional
1/2       -100000000000000/3    80/10    Common Lisp
```

Recall that rationals include the integers; 80\10 as input to the reader is equivalent to 8.

Complex Numbers:

The traditional syntax for a complex number is a number (for the real part), a sign (+ or -), an unsigned number (for the imaginary part), and the letter i. The real and imaginary parts can be any type of number, but they are converted to be of the same type (both floating of the same format, or both rational). For example:

```
        1-3\4i
        1.2s0+3.45s8i
```

The Common Lisp syntax for a complex number is `#c(`*real imag*`)`, where *real* is the real part and *imag* is the imaginary part. This construction is allowed in traditional syntax too.

```
        #c(1 -3/4)
        #c(1.2s0 3.45s8)
```

Symbols:

A string of letters, numbers, and characters without special syntactic meaning is recognized by the reader as a symbol, provided it cannot be interpreted as a number. Alphabetic case is ignored in symbols; lower-case letters are translated to upper-case unless escaped. When the reader sees the p.r. of a symbol, it *interns* it on a *package* (see chapter 27, page 636, for an explanation of interning and the package system). Symbols may start with digits; you could even have one named **-345t**; **read** accepts this as a symbol without complaint. If you want to put strange characters (such as lower-case letters, parentheses, or reader macro characters) inside the name of a symbol, they must be escaped. If the symbol's name would look like a number, at least one character in the name must be escaped, but it matters not which one.
Examples of symbols:

```
        foo
        bar/(baz/)      ; traditional
        bar\(baz\)      ; Common Lisp
        34w23
        |Frob Sale| and F|rob |S|ale| are equivalent

        |a/|b|      ; traditional
        |a\|b|      ; Common Lisp
```

In Common Lisp syntax, a symbol composed only of two or more periods is not allowed unless escaping is used.

The reader can be directed to perform substitutions on the symbols it reads. Symbol substitutions are used to implement the incompatible Common Lisp definitions of various system functions. Reading of Common Lisp code is done with substitutions that replace **subst** with **cli:subst**, **member** with **cli:member**, and so on. This is why, when a Common Lisp program uses the function **member**, it gets the standard Common Lisp **member** function rather than the traditional one. This is why we say that **cli:member** is "the Common Lisp version of **member**". While **cli:member** can be referred to from any program in just that way, it exists primarily to be referred to from a Common Lisp program which says simply **member**.

Symbol substitutions do not apply to symbols written with package prefixes, so one can use a package prefix to force a reference to a symbol that is normally substituted for, such as using **global:member** in a Common Lisp program.

Strings:

Strings are written with double-quote characters (") before and after the string contents. To include a double-quote character or single-character escape character in the contents, write an extra single-character escape character in front of it.
Examples of strings:

```
"This is a typical string."
"That is a /"cons cell/"."        ;; traditional
"That is a \"cons cell\"."        ;; Common Lisp
"Strings are often used for I//O."     ;; traditional
"Strings are often used for I/O."      ;; Common Lisp
"Here comes one backslash: \\"         ;; Common Lisp
```

Conses:

When **read** sees an open-parenthesis, it knows that the p.r. of a cons is coming, and calls itself recursively to get the elements of the cons or the list that follows. The following are valid p.r.'s of conses:

```
(foo . bar)
(foo "bar" 33)
(foo . ("bar" . (33 . nil)))
(foo bar . quux)
```

The first is a cons, whose car and cdr are both symbols. The second is a list, and the third is equivalent to the second (although **print** would never produce it). The fourth is a dotted list; the cdr of the last cons cell (the second one) is not **nil**, but **quux**.

The reader always allocates new cons cells to represent parentheses. They are never shared with other structure, not even part of the same read. For example,

```
(let ((x (read)))
  (eq (car x) (cdr x)))
((a b) . (a b))              ;; data for read
  => nil
```

because each time **(a b)** is read, a new list is constructed. This contrasts with the case for symbols, as very often **read** returns symbols that it found interned in the package rather than creating new symbols itself. Symbols are the only thing that work this way.

The dot that separates the two elements of a dotted-pair p.r. for a cons is only recognized if it is surrounded by delimiters (typically spaces). Thus dot may be freely used within print-names of symbols and within numbers. This is not compatible with Maclisp; in Maclisp **(a.b)** reads as a cons of symbols **a** and **b**, whereas in Zetalisp it reads as a list of a symbol **a.b**.

Comments:

A comment begins with a semicolon (;) and continues to the end of the line. Comments are ignored completely by the reader. If the semicolon is escaped or inside a string, it is not recognized as starting a comment; it is part of a symbol or part of the string.

```
;; This is a comment.
"This is a string; but no comment."
```

Another way to write a comment is to start it with `#|` and end it with `|#`. This is useful for commenting out multiple-line segments of code. The two delimiters nest, so that `#| #| |# |#` is a single comment. This prevents surprising results if you use this construct to comment out code which already contains such a comment.

```
(cond ((atom x) y)
 #|
      ((foo x)
       (do-it y))
    |#
      (t (hack y)))
```

Abbreviations:

The single-quote character (') is an abbreviation for a list starting with the symbol **quote**. The following pairs of p.r.'s produce equal lists:

```
'a     and    (quote a)
'(x (y))    and    (quote (x (y)))
```

The backquote character (`) and comma are used in a syntax that abbreviates calls to the list and vector construction functions. For example,

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

reads as a list whose meaning as a Lisp form is equivalent to

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

See section 18.2.2, page 325 for full details about backquote.

23.3.1 Sharp-sign Constructs

Sharp-sign (#) is used to introduce syntax extensions. It is the beginning of a two-character sequence whose meaning depends on the second character. Sharp-sign is only recognized with a special meaning if it occurs at the beginning of a token. If encountered while a token is in progress, it is a symbol constituent. For example, `#xff` is a sharp-sign construct that interprets `ff` as a hexidecimal number, but `1#xff` is just a symbol.

If the sharp-sign is followed by decimal digits, the digits form a parameter. The first non-digit determines which sharp-sign construct is actually in use, and the decimal integer parsed from the digits is passed to it. For example, `#r` means "read in specified radix"; it must actually be used with a radix notated in decimal between the `#` and the `r`, as in `#8r`.

It is possible for a sharp-sign construct to have different meanings in Common Lisp and traditional syntax. The only constructs which differ are **#** and **#/**.

The function **set-dispatch-macro-character** (see page 541) can be used to define additional sharp sign abbreviations.

Here are the currently-defined sharp sign constructs:

#/ **#/** is used in traditional syntax only to represent the number that is the character code for a character. You can follow the **#/** with the character itself, or with the character's name. The name is preferable for nonprinting characters, and it is the only way to represent characters which have control bits since they cannot go in files. Here are examples of **#/**:

`#/a`	`#o141`
`#/A`	`#o101`
`#/(`	`#o50`
`#/c-a`	the character code for **Control-A**
`#/c-/a`	the character code for **Control-a**
`#/c-sh-a`	the character code for **Control-a**
`#/c-/A`	the character code for **Control-A**
`#/c-/(`	the character code for **Control-(**
`#/return`	the character code for **Return**
`#/h-m-system`	the character code for **Hyper-Meta-System**

To represent a printing character, write **#/***x* where *x* is the character. For example, **#/a** is equivalent to **#o141** but clearer in its intent. To avoid ambiguity, the character following *x* should not be a letter; good style would require this anyway.

As in strings, upper and lower-case letters are distinguished after **#/**. Any character works after **#/**, even those that are normally special to **read**, such as parentheses. Thus, **#/A** is equivalent to **#o101**, and **#/(** is equivalent to **#o50**. Note that the slash causes this construct to be parsed correctly by the editors Emacs and Zmacs. Even non-printing characters may be used, but for them it is preferable to use the character's name.

To refer to a character by name, write **#/** followed by the name. For example, **#/return** reads as the numeric code for the character **Return**. The defined character names are documented below (see section 10.1.6, page 211). In general, the names that are written on the keyboard keys are accepted. In addition, all the nonalphanumeric characters have names. The abbreviations **cr** for **return** and **sp** for **space** are accepted, since these characters are used so frequently. The page separator character is called **page**, although **form** and **clear-screen** are also accepted since the keyboard has one of those legends on the page key. The rules for reading *name* are the same as those for symbols; thus letters are converted to upper case unless escaped, and the name must be terminated by a delimiter such as a space, a carriage return, or a parenthesis.

When the system types out the name of a special character, it uses the same table that **#/** uses; therefore, any character name typed out is acceptable as input.

#/ can also be used to read in the names of characters that have modifier bits (Control, Meta, Super and Hyper). The syntax looks like #/control-meta-b to get a 'B' character with the control and meta bits set. You can use any of the prefix bit names **control**, **meta**, **hyper**, and **super**. They may be in any order, and case is not significant. Prefix bit names can be abbreviated as the single letters c, m, h and s, and **control** may be spelled **ctrl** as it is on the keyboard. The last hyphen may be followed by a single character or by any of the special character names normally recognized by #/. A single character is treated the same way the reader normally treats characters in symbols; if you want to use a lower-case character or a special character such as a parenthesis, you must precede it by a slash character. Examples: #/Hyper-Super-A, #/meta-hyper-roman-i, #/CTRL-META-/(.

An obsolete method of specifying control bits in a character is to insert the characters α, β, ε, π and λ between the # and the /. Those stand for control, meta, control-meta, super and hyper, respectively. This syntax should be converted to the new #\control-meta-x syntax described below.

greek (or **front**), **top**, and **shift** (or **sh**) are also allowed as prefixes of names. Thus, #/top-g is equivalent to #/↑ or #/uparrow. #/top-g should be used if you are specifying the keyboard commands of a program and the mnemonic significance belongs to the 'G' rather than to the actual character code.

#\ In traditional syntax, #\ is a synonym for #/. In the past, #/ had to be used before a single character and #\ had to be used in all other cases. Now either one is allowed in either case.

In Common Lisp syntax, #\ produces a character object rather than a fixnum representing a character.

#⋆/ #⋆/*x* is the traditional syntax way to produce a character object. It is used just like #/. Thus, Common Lisp #\ is equivalent to traditional syntax #⋆/.

#^ #^*x* is exactly like #/control-*x* if the input is being read by Zetalisp; it generates Control-*x*. In Maclisp *x* is converted to upper case and then exclusive-or'ed with 100 (octal). Thus #^*x* always generates the character returned by **tyi** if the user holds down the control key and types *x*. (In Maclisp #/control-*x* sets the bit set by the Control key when the TTY is open in **fixnum** mode.)

#' #'*foo* is an abbreviation for (**function** *foo*). *foo* is the p.r. of any object. This abbreviation can be remembered by analogy with the ' macro-character, since the **function** and **quote** special forms are somewhat analogous.

#(#(*elements*...) constructs a vector (rank-one array) of type **art-q** with elements *elements*. The length of the vector is the number of elements written. Thus, #(a 5 "Foo") reads as a vector containing a symbol, an integer and a string. If a decimal integer appears after the #, it specifies the length of the vector. The last element written is replicated to fill the remaining elements.

#a #*n*a *contents* signifies an array of rank *n*, containing *contents*. *contents* is passed to **make-array** as the *initial-contents* argument. It is a list of lists of lists... or vector of vectors... as deep as *n*. The dimensions of the array are specified by the lengths of the lists or vectors. The rank is specified explicitly so that the reader can distinguish whether

a list or vector in the contents is a sequence of array elements or a single array element. The array type is always **art-q**.
Examples:

```
        #2a ((x y) (a b) ((uu 3) "VV"))
```

produces a 3 by 2 array. **(uu 3)** is one of the elements.

```
        #2a ("foo" "bar")
```

produces a 2 by 3 array whose elements are character objects. Recall that a string is a kind of vector.

```
        #0a 5
```

produces a rank-0 array whose sole element is 5.

#* #**bbb*... signifies a bit vector; *bbb*... are the bits (characters **1** or **0**). A vector of type **art-1b** is created and filled with the specified bits, the first bit specified going in array element 0. The length is however many bits you specify. Alternatively, specify the length with a decimal number between the # and the *. The last **1** or **0** specified is duplicated to fill the additional bits. Thus, **#8*0101** is the same as **#*01011111**.

#s #s(*type slot value slot value* ...) constructs a structure of type *type*. Any structure type defined with **defstruct** can be used as *type* provided it has a standard constructor taking slot values as keyword arguments. (Standard constructors can be functions or macros; either kind works for **#s**.) The slot names and values appearing in the read syntax are passed to the constructor so that they initialize the structure. Example:

```
        (defstruct (foo :named)
          bar
          lose)
        #s (foo :bar 5 :lose haha)
```

produces a **foo** whose **bar** component is 5 and whose **lose** component is **haha**.

#=
Are used to represent circular structure or shared structure. **#*n*=** preceding an object "labels" that object with the label *n*, a decimal integer. This has no effect on the way the object labeled is read, but it makes the label available for use in a **#*n*#** construct within that object (to create circular structure) or later on (to create shared structure). **#*n*#** counts as an object in itself, and reads as the object labeled by *n*.

For example, **#1=(a . #1#)** is a way of notating a circular list such as would be produced by **(circular-list 'a)**. The list is labeled with label 1, and then its cdr is given as a reference to label 1. **(#1=#:foo #1#)** is an example of shared structure. An uninterned symbol named **foo** is used as the first element of the list, and labeled. The second element of the list is the very same uninterned symbol, by virtue of a reference to the label.

Printing outputs **#*n*=** and **#*n*#** to represent circular or shared structure when ***print-circle** is non-**nil**.

#, Evaluate a form at load time. **#,** *foo* evaluates *foo* (the p.r. of a Lisp form) at read time, except that during file-to-file compilation it is arranged that *foo* will be evaluated when the QFASL file is loaded. This is a way, for example, to include in your code complex list-structure constants that cannot be written with **quote**. Note that the reader does not put **quote** around the result of the evaluation. You must do this yourself if you want it, typically by using the ' macro-character. An example of a case where you do not want quote around it is when this object is an element of a constant list.

#. #.*foo* evaluates *foo* (the p.r. of a lisp form) at read time, regardless of who is doing the reading.

#' #' is a construct for repeating an expression with some subexpressions varying. It is an abbreviation for writing several similar expressions or for the use of mapc. Each subexpression that is to be varied is written as a comma followed by a list of the things to substitute. The expression is expanded at read time into a progn containing the individual versions.

```
        #'(send stream ',(:clear-input :clear-output))
```
expands into
```
        (progn (send stream :clear-input)
               (send stream :clear-output))
```

Multiple repetitions can be done in parallel by using commas in several subexpressions:
```
        #'(rename-file ,("foo" "bar") ,("ofoo" "obar"))
```
expands into
```
        (progn (rename-file "foo" "ofoo")
               (rename-file "bar" "obar"))
```

If you want to do multiple independent repetitions, you must use nested #' constructs. Individual commas inside the inner #' apply to that #'; they vary at maximum speed. To specify a subexpression that varies in the outer #', use two commas.
```
        #'#'(print (* ,(5 7) ,,(11. 13.)))
```
expands into
```
        (progn (progn (print (* 5 11.)) (print (* 7 11.)))
               (progn (print (* 5 13.)) (print (* 7 13.))))
```

#o #o *number* reads *number* in octal regardless of the setting of *read-base*. Actually, any expression can be prefixed by #o; it is read with *read-base* bound to 8.

#b Like #o but reads in binary.

#x Like #x but reads in radix 16 (hexadecimal). The letters a through f are used as the digits beyond 9.

#r #*radix*r *number* reads *number* in radix *radix* regardless of the setting of *read-base*. As with #o, any expression can be prefixed by #*radix*r; it is read with *read-base* bound to *radix*. *radix* must be a valid decimal integer between 2 and 36.

For example, #3r102 is another way of writing 11. and #11r32 is another way of writing 35. Bases larger than ten use the letters starting with a as the additional digits.

#c #c(*real imag*) constructs a complex number with real part *real* and imaginary *part*. It is equivalent to *real* + *imag*i, except that #c is allowed in Common Lisp syntax and the other is not.

#+ This abbreviation provides a read-time conditionalization facility. It is used as #+*feature form*. If *feature* is a symbol, then this is read as *form* if *feature* is present in the list *features* (see page 803). Otherwise, the construct is regarded as whitespace.

Alternately, *feature* may be a boolean expression composed of **and**, **or**, and **not** operators and symbols representing items that may appear on ***features***. Thus, **#+(or lispm amber)** causes the following object to be seen if either of the features **lispm** or **amber** is present.

For example, **#+lispm** *form* makes *form* count if being read by Zetalisp, and is thus equivalent to **#q** *form*. Similarly, **#+maclisp** *form* is equivalent to **#m** *form*. **#+(or lispm nil)** *form* makes *form* count on either Zetalisp or in NIL.

Here is a list of features with standard meanings:

lispm	This feature is present on any Lisp machine (no matter what version of hardware or software).
maclisp	This feature is present in Maclisp.
nil	This feature is present in NIL (New Implementation of Lisp).
mit	This feature is present in the MIT Lisp machine system, which is what this manual is about.
symbolics	This feature is present in the Symbolics version of the Lisp machine system. May you be spared the dishonor of using it.

#+, and the other read-time conditionalization constructs that follow, discard the following expression by reading it with ***read-suppress*** bound to **t** if the specified condition is false.

#- **#-***feature form* is equivalent to **#+(not** *feature***)** *form*.

#q **#q** *foo* reads as *foo* if the input is being read by Zetalisp, otherwise it reads as nothing (whitespace). This is considered obsolete; use **#+lispm** instead.

#m **#m** *foo* reads as *foo* if the input is being read into Maclisp, otherwise it reads as nothing (whitespace). This is considered obsolete; use **#+maclisp** instead.

#n **#n** *foo* reads as *foo* if the input is being read into NIL or compiled to run in NIL, otherwise it reads as nothing (white space). This is considered obsolete; use **#+nil** instead.

#◊ **#◊** introduces an expression in infix notation. ◊ should be used to terminate it. The text in between describes a Lisp object such as a symbol, number or list but using a nonstandard, infix-oriented syntax. For example,

```
        #◊x:y+car(a1[i,j])◊
```
is equivalent to
```
        (setq x (+ y (car (aref a1 i j))))
```

It is not strictly true that the Lisp object produced in this way has to be an expression. Since the conversion is done at read time, you can use a list expressed this way for any purpose. But the infix syntax is designed to be used for expressions.

For full details, refer to the file **SYS: IO1; INFIX LISP**.

#< This is not legal reader syntax. It is used in the p.r. of objects that cannot be read back in. Attempting to read a **#<** signals an error.

#⊂

This is used in the p.r. of miscellaneous objects (usually named structures or instances) that can be read back in. #⊂ should be followed by a typename and any other data needed to construct an object, terminated with a ⊃. For example, a pathname might print as

```
        #⊂FS:ITS-PATHNAME "AI: RMS; TEST 5"⊃
```

The typename is a keyword that **read** uses to figure out how to read in the rest of the printed representation and construct the object. It is read in in package **user** (but it can contain a package prefix). The resulting symbol should either have a **si:read-instance** property or be the name of a flavor that handles the **:read-instance** operation.

In the first case, the property is applied as a function to the typename symbol itself and the input stream. In the second, the handler for that operation is applied to the operation name (as always), the typename symbol, and the input stream (three arguments, but the first is implicit and not mentioned in the **defmethod**). **self** will be **nil** and instance variables should not be referred to. **si:print-readably-mixin** is a useful implementation the **:read-instance** operation for general purposes; see page 446.

In either case, the handler function should read the remaining data from the stream, and construct and return the datum it describes. It should return with the ⊃ character waiting to be read from the input stream (**:untyi** it if necessary). **read** signals an error after it is returned to if a ⊃ character is not next.

The typename can be any symbol with an appropriate property or flavor, not necessarily related to the type of object that is created; but for clarity, it is good if it is the same as the **type-of** of the object printed. Since the type symbol is passed to the handler, one flavor's handler can be inherited by many other flavors and can examine the type symbol read in to decide what flavor to construct.

#| #| is used to comment out entire pieces of code. Such a comment begins with #| and ends with |#. The text in between should be one or more properly balanced p.r.'s of Lisp objects, possibly including nested #|...|# comments. This text is skipped over by the reader, and does not contribute to the value returned by **read**.

23.4 Expression Output Functions

These functions all take an optional argument called *stream*, which is where to send the output. If unsupplied *stream* defaults to the value of ***standard-output***. If *stream* is **nil**, the value of ***standard-output*** (i.e. the default) is used. If it is **t**, the value of ***terminal-io*** is used (i.e. the interactive terminal). This is all more-or-less compatible with Maclisp, except that instead of the variable ***standard-output*** Maclisp has several variables and complicated rules. For detailed documentation of streams, refer to section 22.3, page 459.

prin1 *object* &optional *stream*
Outputs the printed representation of *object* to *stream*, with escaping (see page 506). *object* is returned.

prin1-then-space *object* &optional *stream*
Like **prin1** except that output is followed by a space.

print *object* &optional *stream*
Like **prin1** except that output is preceded by a carriage return and followed by a space. *object* is returned.

princ *object* &optional *stream*
Like **prin1** except that the output is not escaped. *object* is returned.

write *object* &key *stream escape radix base circle pretty level length case gensym array*
Prints *object* on *stream*, having bound all the printing flags according to the keyword arguments if specified. For example, the keyword argument *array* specifies how to bind ***print-array***; if *array* is omitted, the ambient value of ***print-array*** is used. This function is sometimes cleaner than binding a printing control variable explicitly. The value is *object*.

23.4.1 Pretty-Printing Output Functions

pprint *object* &optional *stream*
pprint is like **prin1** except that ***print-pretty*** is bound to **t** so that the grinder is used. **pprint** returns zero values, just as the form **(values)** does.

grindef *function-spec...* *Macro*
Prints the definitions of one or more functions, with indentation to make the code readable. Certain other "pretty-printing" transformations are performed: The **quote** special form is represented with the ' character. Displacing macros are printed as the original code rather than the result of macro expansion. The code resulting from the backquote (`) reader macro is represented in terms of `.

The subforms to **grindef** are the function specs whose definitions are to be printed; the usual way **grindef** is used is with a form like **(grindef foo)** to print the definition of **foo**. When one of these subforms is a symbol, if the symbol has a value its value is prettily printed also. Definitions are printed as **defun** special forms, and values are printed as **setq** special forms.

If a function is compiled, **grindef** says so and tries to find its previous interpreted definition by looking on an associated property list (see **uncompile** (page 301). This works only if the function's interpreted definition was once in force; if the definition of the function was simply loaded from a QFASL file, **grindef** cannot not find the interpreted definition.

With no subforms, **grindef** assumes the same arguments as when it was last called.

grind-top-level *obj* &optional *width* (*stream* *standard-output*) (*untyo-p* nil) (*displaced* 'si:displaced) (*terpri-p* t) *notify-fun* *loc*

Pretty-prints *obj* on *stream*, putting up to *width* characters per line. This is the primitive interface to the pretty-printer. Note that it does not support variable-width fonts. If the *width* argument is supplied, it is how many characters wide the output is to be. If *width* is unsupplied or nil, grind-top-level tries to figure out the natural width of the stream, by sending a :size-in-characters message to the stream and using the first returned value. If the stream doesn't handle that message, a width of 95. characters is used instead.

The remaining optional arguments activate various strange features and usually should not be supplied. These options are for internal use by the system and are documented here for only completeness. If *untyo-p* is t, the :untyo and :untyo-mark operations are be used on *stream*, speeding up the algorithm somewhat. *displaced* controls the checking for displacing macros; it is the symbol which flags a place that has been displaced, or nil to disable the feature. If *terpri-p* is nil, grind-top-level does not advance to a fresh line before printing.

If *notify-fun* is non-nil, it should be a function that to be called with three arguments for each "token" in the pretty-printed output. Tokens are atoms, open and close parentheses, and reader macro characters such as '. The arguments given to *notify-fun* are the token, its "location" (see next paragraph), and t if it is an atom or nil if it is a character.

loc is the "location" (typically a cons) whose car is *obj*. As the grinder recursively descends through the structure being printed, it keeps track of the location where each thing came from, for the benefit of the *notify-fun*. This makes it possible for a program to correlate the printed output with the list structure. The "location" of a close parenthesis is t, because close parentheses have no associated location.

23.4.2 Non-Stream Printing Functions

write-to-string *object* &key *escape radix base circle pretty level length case gensym array*
prin1-to-string *object*
princ-to-string *object*

Like write, prin1 and princ, respectively, but put the output in a string and return the string (see page 528).

See also the with-output-to-string special form (page 474).

The following obsolete functions are for Maclisp compatibility only. The examples use traditional syntax.

exploden *object*

Returns a list of characters (represented as fixnums) that are the characters that would be typed out by (princ *object*) (i.e. the unescaped printed representation of *object*).

Example:
```
        (exploden '(+ /12 3)) => #o(50 53 40 61 62 40 63 51)
```

explodec *object*

Returns a list of characters represented by symbols, interned in the current package, whose names are the characters that would be typed out by (princ *object*) (i.e. the unescaped printed representation of *object*).
Example:
```
        (explodec '(+ /12 3)) => (|(| + | | |1| |2| | | |3| |)|)
```
(Note that there are escaped spaces in the above list.)

explode *object*

Like explodec but uses the escaped printed representation.
Example:
```
        (explode '(+ /12 3)) => (|(| + | | // |1| |2| | | |3| |)|)
```
(Note that there are escaped spaces in the above list.)

flatsize *object*

Returns the number of characters in the escaped printed representation of *object*.

flatc *object*

Returns the number of characters in the unescaped printed representation of *object*.

23.5 Expression Input Functions

Most expression input functions read characters from an input stream. This argument is called *stream*. If unsupplied it defaults to the value of *standard-input*.

All of these functions echo their input and permit editing if used on an interactive stream (one which supports the :rubout-handler operation; see below.)

The functions accept an argument *eof-option* or two arguments *eof-error* and *eof-value* to tell them what to do if end of file is encountered instead of an object's p.r. The functions that take two *eof-* arguments are the Common Lisp ones.

In functions that accept the *eof-option* argument, if no argument is supplied, an error is signaled at eof. If the argument is supplied, end of file causes the function to return that argument. Note that an *eof-option* of nil means to return nil if the end of the file is reached; it is *not* equivalent to supplying no *eof-option*.

In functions that accept two arguments *eof-error* and *eof-value*, end of file is an error if *eof-error* is non-nil or if it is unsupplied. If *eof-error* is nil, then the function returns *eof-value* at end of file.

An error is always signaled if end of file is encountered in the middle of an object; for example, if a file does not contain enough right parentheses to balance the left parentheses in it. Mere whitespace does not count as starting an object. If a file contains a symbol or a number immediately followed by end-of-file, it can be read normally without error; if an attempt is made to read further, end of file is encountered immediately and the *eof-* argument(s) obeyed.

These end-of-file conventions are not completely compatible with Maclisp. Maclisp's deviations from this are generally considered to be bugs rather than features.

For Maclisp compatibility, **nil** as the *stream* argument also means to use the value of ***standard-input***, and **t** as the *stream* argument means to use the value of ***terminal-io***. This is only advertised to work in functions that Maclisp has, and should not be written in new programs. Instead of the variable ***standard-input*** Maclisp has several variables and complicated rules. For detailed documentation of streams, refer to section 22.3, page 459.

The functions below that take *stream* and *eof-option* arguments can also be called with the stream and eof-option in the other order. This functionality is only for compatibility with old Maclisp programs, and should never be used in new programs. The functions attempt to figure out which way they were called by seeing whether each argument is a plausible stream. Unfortunately, there is an ambiguity with symbols: a symbol might be a stream and it might be an eof-option. If there are two arguments, one being a symbol and the other being something that is a valid stream, or only one argument, which is a symbol, then these functions interpret the symbol as an eof-option instead of as a stream. To force them to interpret a symbol as a stream, give the symbol an **si:io-stream-p** property whose value is **t**.

read &optional *stream eof-option rubout-handler-options*

Reads the printed representation of a Lisp object from *stream*, builds a corresponding Lisp object, and returns the object. *rubout-handler-options* are used as options for the rubout handler, if *stream* supports one; see section 22.5, page 500 for more information on this.

cli:read &optional *stream* (*eof-errorp* **t**) *eof-value recursive-p*

The Common Lisp version of **read** differs only in how its arguments are passed.

recursive-p should be non-**nil** when calling from the reader or from the defining function of a read-macro character; that is, when reading a subexpression as part of the task of reading a larger expression. This has two effects: the subexpression is allowed to share *#n#* labels with the containing expression, and whitespace which terminates the subexpression (if it is a symbol or number) is not discarded.

read-or-end &optional *stream eof-option rubout-handler-options*

Like **read**, but on an interactive stream if the input is just the character **End** it returns the two values **nil** and **:end**.

read-preserve-delimiters *Variable*

Certain printed representations given to **read**, notably those of symbols and numbers, require a delimiting character after them. (Lists do not, because the matching close-parenthesis serves to mark the end of the list.) Normally **read** throws away the delimiting character if it is whitespace, but preserves it (using the **:untyi** stream operation) if the character is syntactically meaningful, since it may be the start of the next expression.

If **read-preserve-delimiters** is bound to **t** around a call to **read**, the delimiting character is never thrown away, even if it is whitespace. This may be useful for certain reader macros or special syntaxes.

read-preserving-whitespace &optional *stream* (*eof-errorp* t) *eof-value recursive-p*

Like cli:read but binds read-preserve-delimiters to t. This is the Common Lisp way of requesting the read-preserve-delimiters feature.

read-delimited-list *char* &optional *stream recursive-p*

Reads expressions from *stream* until the character *char* is seen at top level when an expression is expected; then returns a list of the objects read. *char* may be a fixnum or a character object. For example, if *char* is #/], and the text to be read from *stream* is a (b c)] ... then the objects a and (b c) are read, the] is seen as a terminator and discarded, and the value returned is (a (b c)). *recursive-p* is as for cli:read. End of file within this function is always an error since it is always "within an object"—the object whose textual representation is terminated by *char*.

Note that use of this function does not cause *char* to terminate tokens. Usually you want that to happen, but it is purely under the control of the readtable. So you must modify the readtable to make this so. The usual way is to define *char* as a macro character whose defining function just signals an error. The defining function is not called when *char* is encountered in the expected context by read-delimited-list; if *char* is encountered anywhere else, it is an unbalanced bracket and an error is appropriate.

read-for-top-level &optional *stream eof-option*

This is a slightly different version of read. It differs from read only in that it ignores close-parentheses seen at top level, and it returns the symbol si:eof if the stream reaches end-of-file if you have not supplied an *eof-option* (instead of signalling an error as read would). This version of read is used in the system's "read-eval-print" loops.

read-check-indentation &optional *stream eof-option*

This is like read, but validates the input based on indentation. It assumes that the input data is formatted to follow the usual convention for source files, that an open-parenthesis in column zero indicates a top-level list (with certain specific exceptions). An open-parenthesis in column zero encountered in the middle of a list is more likely to result from close-parentheses missing before it than from a mistake in indentation.

If read-check-indentation finds an open-parenthesis following a return character in the middle of a list, it invents enough close-parentheses to close off all pending lists, and returns. The offending open-parenthesis is :untyi'd so it can begin the next list, as it probably should. End of file in the middle of a list is handled likewise.

read-check-indentation notifies the caller of the incorrect formatting by signaling the condition sys:missing-closeparen. This is how the compiler is able to record a warning about the missing parentheses. If a condition handler proceeds, read goes ahead and invents close-parentheses.

There are a few special forms that are customarily used around function definitions—for example, eval-when, local-declare, and comment. Since it is desirable to begin the function definitions in column zero anyway, read-check-indentation allows a list to begin in column zero within one of these special forms. A non-nil si:may-surround-defun property identifies the symbols for which this is allowed.

read-check-indentation *Variable*

This variable is non-nil during a read in which indentation is being checked.

23.5.1 Non-Stream Parsing Functions

The following functions do expression input but get the characters from a string or a list instead of a stream.

read-from-string *string* &optional *eof-option* (*start* 0) *end*

The characters of *string* are given successively to the reader, and the Lisp object built by the reader is returned. Macro characters and so on all take effect. If *string* has a fill-pointer it controls how much can be read.

eof-option is what to return if the end of the string is reached, as in read. *start* is the index in the string of the first character to be read. *end* is the index at which to stop reading; that point is treated as end of file.

read-from-string returns two values; the first is the object read and the second is the index of the first character in the string not read. If the entire string was read, this is the length of the string.

Example:

```
        (read-from-string "(a b c)") => (a b c) and 7
```

cli:read-from-string *string* &optional (*eof-errorp* t) *eof-value* &key (*start* 0) *end* *preserve-whitespace*

The Common Lisp version of read-from-string uses a different calling convention. The arguments mean the same thing but are arranged differently. There are three arguments with no counterparts: *eof-errorp* and *eof-value*, which are simply passed on to cli:read, and *preserve-whitespace*, which if non-nil means that the reading is done with read-preserve-delimiters bound to t.

See also the with-input-from-string special form (page 473).

parse-integer *string* &key (*start* 0) *end* (*radix* 10.) *junk-allowed*

Parses the contents of *string* (or the portion from *start* to *end*) as a numeral for an integer using the specified radix, and returns the integer. Radices larger than ten are allowed, and they use letters as digits beyond 9. Leading whitespace is always allowed and ignored. A leading sign is also allowed and considered part of the number.

When *junk-allowed* is nil, the entire specified portion of string must consist of an integer and leading and trailing whitespace. Otherwise, an error happens.

If *junk-allowed* is non-nil, parsing just stops when a non-digit is encountered. The number parsed so far is returned as the first value, and the index in *string* at which parsing stopped is returned as the second value. This number equals *end* (or the length of *string*) if there is nothing but a number. If non-digits are found without finding a number first, the first value is nil. Examples:

```
(parse-integer " 1A " :radix 16.) => 26.
(parse-integer " 15X " :end 3) => 15.
(parse-integer " -15X " :junk-allowed t) => -15. 3
(parse-integer " 15X ") => error!
```

readlist *char-list*

This function is provided mainly for Maclisp compatibility. *char-list* is a list of characters. The characters may be represented by anything that the function character accepts: character objects, fixnums, strings, or symbols. The characters are given successively to the reader, and the Lisp object built by the reader is returned. Macro characters and so on all take effect.

If there are more characters in *char-list* beyond those needed to define an object, the extra characters are ignored. If there are not enough characters, some kind of sys:read-end-of-file error is signaled.

23.5.2 Input Error Conditions

sys:read-error (sys:parse-error error) *Condition*

This condition name classifies all errors detected by the reader per se. Since sys:parse-error is implied, all sys:read-error errors must provide the proceed type :no-action so that automatic proceed is possible if the error happens during compilation. See page 505.

Since this condition name implies sys:parse-error and error, those two are not mentioned as implications below when sys:read-error is.

sys:read-end-of-file (sys:read-error sys:end-of-file) *Condition*

Whenever the reader signals an error for end of file, the condition object possesses this condition name.

Since sys:end-of-file is implied, the :stream operation on the condition instance returns the stream on which end of file was reached.

sys:read-list-end-of-file *Condition*
(sys:read-end-of-file sys:read-error sys:end-of-file)

This condition is signaled when read detects end of file in the middle of a list.

In addition to the :stream operation provided because sys:end-of-file is one of the proceed types, the condition instance supports the :list operation, which returns the list read so far.

Proceed type :no-action is provided. If it is used, the reader invents a close-parenthesis to close off the list. Within read-check-indentation, the reader signals the error only once, no matter how many levels of list are unterminated.

sys:read-string-end-of-file *Condition*
(sys:read-end-of-file sys:read-error sys:end-of-file)
This is signaled when read detects end of file in the middle of a string delimited by double-quotes.

The :string operation on the condition instance returns the string read so far.

Proceed type :no-action terminates the string and returns. If the string is within other constructs that are unterminated, another end of file error is will be signaled later.

sys:read-symbol-end-of-file *Condition*
(sys:read-end-of-file sys:read-error sys:end-of-file)
This is signaled when read detects end of file within a multiple escape construct.

The :string operation on the condition instance returns the print name read so far.

Proceed type :no-action terminates the symbol and returns. If the symbol is within other constructs that are unterminated, another end of file error is will be signaled later.

sys:missing-closeparen (condition) *Condition*
This condition, which is not an error, is signaled when read-check-indentation finds an open-parenthesis in column zero within a list.

Proceed type :no-action is provided. On proceeding, the reader invents enough close-parentheses to close off all the lists that are pending.

23.6 The Readtable

The syntax used by the reader is controlled by a data structure called the *readtable*. (Some aspects of printing are also controlled by the readtable.) There can be many readtables, but the one that is used is the one which is the value of *readtable*. A particular syntax can be selected for use by setting or binding *readtable* to a readtable which specifies that syntax before reading or printing. In particular, this is how Common Lisp or traditional syntax is selected. The readtable also controls the symbol substitutions which implement the distinction between the traditional and Common Lisp versions of functions such as subst, memberand defstruct.

The functions in this section allow you to modify the syntax of individual characters in a readtable in limited ways. You can also copy a readtable; then you can modify one copy and leave the other unchanged.

A readtables may have one or more names. Named readtables are recorded in a central data base so that you can find a readtable by name. When you copy a readtable, the new one is anonymous and is not recorded in the data base.

readtable *Variable*
readtable *Variable*

The value of readtable or *readtable* is the current readtable. This starts out as the initial standard readtable. You can bind this variable to change temporarily the readtable being used.

The two names are synonymous.

si:standard-readtable *Constant*

This is copied into *readtable* every time the machine is booted. Therefore, it is normally the same as *readtable* unless you make *readtable* be some other readtable. If you alter the contents of *readtable* without setting or binding it to some other readtable, this readtable is changed.

si:initial-readtable *Constant*

The value of si:initial-readtable is a read-only copy of the default current readtable. Its purpose is to preserve a copy of the standard read syntax in case you modify the contents of *readtable* and regret it later. You could use si:initial-readtable as the *from-readtable* argument to copy-readtable or set-syntax-from-char to restore all or part of the standard syntax.

si:common-lisp-readtable *Constant*

A readtable which initially is set up to define Common Lisp read syntax. Reading of Common Lisp programs is done using this readtable.

si:initial-common-lisp-readtable *Constant*

A read-only copy of si:common-lisp-readtable, whose purpose is to preserve a copy of the standard Common Lisp syntax in case you modify si:common-lisp-readtable (such as, by reading a Common Lisp program which modifies the current readtable).

si:rdtbl-names *readtable*

Returns the list of names of readtable. You may setf this to add or remove names.

si:find-readtable-named *name*

Returns the readtable named name, or nil if none is recorded.

readtablep *object*

t if *object* is a readtable.

The user can program the reader by changing the readtable in any of three ways. The syntax of a character can be set to one of several predefined possibilities. A character can be made into a *macro character*, whose interpretation is controlled by a user-supplied function which is called when the character is read. The user can create a completely new readtable, using the readtable compiler (SYS: IO; RTC LISP) to define new kinds of syntax and to assign syntax classes to characters. Use of the readtable compiler is not documented here.

copy-readtable &optional *from-readtable to-readtable*

from-readtable, which defaults to the current readtable, is copied. If *from-readtable* is nil, the standard Common Lisp syntax is copied. If *to-readtable* is unsupplied or nil, a fresh copy is made. Otherwise *to-readtable* is clobbered with the copied syntax.

Use copy-readtable to get a private readtable before using the following functions to change the syntax of characters in it. The value of *readtable* at the start of a Lisp Machine session is the initial standard readtable, which usually should not be modified.

set-syntax-from-char *to-char from-char* &optional *to-readtable from-readtable*

Copies the syntax of *from-char* in *from-readtable* to character *to-char* in *to-readtable*. *to-readtable* defaults to the current readtable and *from-readtable* defaults to si:initial-standard-readtable (standard traditional syntax).

cli:set-syntax-from-char *to-char from-char* &optional *to-readtable from-readtable*

Is a Common Lisp function which copies the syntax of *from-char* in *from-readtable* to character *to-char* in *to-readtable*. *to-readtable* defaults to the current readtable and *from-readtable* defaults to si:initial-common-lisp-readtable (standard Common Lisp syntax).

Common Lisp has a peculiar idea of what it means to copy the syntax of a character. The only aspect of syntax that the readtable supposedly specifies is the choice among

* token constituent: digits, letters, random things like @, !, $, and also colon!
* whitespace: spaces, Tab, Return.
* single escape character: / traditionally, \ in Common Lisp.
* multiple escape character: vertical-bar.
* macro character: standardly ()",.'';
* nonterminating macro character: # is the only such character standardly defined.

The differences among macro characters are determined entirely by the functions that they invoke. The differences among token constituents (including the difference between A and colon) are fixed! You can make A be a macro character, or whitespace, or a quote character, but if you make it a token constituent then it always behaves the way it normally does. You can make colon be a macro character, or whitespace, etc., but if it is a token constituent it always delimits package names. If you make open-parenthesis into a token constituent, there is only one kind of token constituent it can be (it forces the token to be a symbol, like $ or @ or %).

This is not how Lisp Machine readtables really work, but since cli:set-syntax-from-char is provided just for Common Lisp, the behavior specified by Common Lisp is laboriously provided. So, if *from-char* is some kind of token constituent, this function makes *to-char* into a token constituent of the kind that *to-char* is supposed to be—not the kind of token constituent that *from-char* is.

By contrast, the non-Common-Lisp set-syntax-from-char would make *to-char* have exactly the same syntactic properties that *from-char* has.

set-character-translation *from-char to-char* &optional *readtable*

Changes *readtable* so that *from-char* will be translated to *to-char* upon read-in, when *readtable* is the current readtable. This is normally used only for translating lower case letters to upper case. Character translation is inhibited by escape characters and within strings. *readtable* defaults to the current readtable.

The following syntax-setting functions are more or less obsolete.

set-syntax-from-description *char description* &optional *readtable*

Sets the syntax of *char* in *readtable* to be that described by the symbol *description*. *readtable* defaults to the current readtable.

Each readtable has its own set of descriptions which it defines. The following descriptions are defined in the standard readtable:

si:alphabetic	An ordinary character such as 'A'.
si:break	A token separator such as '('. (Obviously left parenthesis has other properties besides being a break.
si:whitespace	A token separator that can be ignored, such as ' '.
si:single	A self-delimiting single-character symbol. The initial readtable does not contain any of these.
si:escape	The character quoter. In the initial readtable this is '/'.
si:multiple-escape	The symbol print-name quoter. In the initial readtable this is '|'.
si:macro	A macro character. Don't use this; use **set-macro-character** (page 540).
si:non-terminating-macro	A macro character recognized only at the start of a token. In the initial readtable, '#' is such a character. (It is also a dispatching macro, but that is another matter.) The correct way to make a character be a macro is with **set-macro-character**.
si:character-code-escape	The octal escape for special characters. In the initial readtable this is '⊗'.
si:digitscale	a character for shifting an integer by digits. In the initial readtable this is '^'.
si:bitscale	A character for shifting an integer by bits. In the initial readtable this is '_' (underscore).
si:slash si:circlecross	Obsolete synonyms for **si:escape** and **si:character-code-escape**.

Unfortunately it is no longer possible to provide **si:doublequote** as double-quote is now an ordinary macro character.

These symbols may be moved to the keyword package at some point.

setsyntax *character arg2 arg3*

This exists only for Maclisp compatibility. The above functions are preferred in new programs. The syntax of *character* is altered in the current readtable, according to *arg2* and *arg3*. *character* can be a fixnum, a symbol, or a string, i.e. anything acceptable to the character function. *arg2* is usually a keyword; it can be in any package since this is a Maclisp compatibility function. The following values are allowed for *arg2*:

:macro	The character becomes a macro character. *arg3* is the name of a function to be invoked when this character is read. The function takes no arguments, may tyi or read from *standard-input* (i.e. may call tyi or read without specifying a stream), and returns an object which is taken as the result of the read.
:splicing	Like :macro but the object returned by the macro function is a list that is nconced into the list being read. If the character is read anywhere except inside a list (at top level or after a dotted-pair dot), then it may return (), which means it is ignored, or (*obj*), which means that *obj* is read.
:single	The character becomes a self-delimiting single-character symbol. If *arg3* is a fixnum, the character is translated to that character.
nil	The syntax of the character is not changed, but if *arg3* is a fixnum, the character is translated to that character.
a symbol	The syntax of the character is changed to be the same as that of the character *arg2* in the standard initial readtable. *arg2* is converted to a character by taking the first character of its print name. Also if *arg3* is a fixnum, the character is translated to that character.

23.7 Read-Macro Characters

A *read-macro character* (or just *macro character*) is a character whose syntax is defined by a function which the reader calls whenever that character is seen (unless it is escaped). This function can optionally read additional characters and is then responsible for returning the object which they represent.

The standard meanings of the characters open-parenthesis, semicolon, single-quote, double-quote, #, backquote (`) and comma are implemented by making them macro characters.

For example, open-parenthesis is implemented as a macro character whose defining function reads expressions until a close-parenthesis is found, throws away the close-parenthesis, and returns a list of the expressions read. (It actually must be more complicated than this in order to deal properly with dotted lists and with indentation checking.) Semicolon is implemented as a macro character whose defining function swallows characters until a Return and then returns no values.

Close-parenthesis and close-horseshoe (⊃) are also macro characters so that they will terminate symbols. Their defining functions signal errors if actually called; but when these delimiters are encountered in their legitimate contexts they are recognized and handled specially before the defining function is called.

The user can also define macro characters.

When a macro's defining function is called, it receives two arguments: the input stream, and the macro character being handled. The function may read characters from the stream, and should return zero or more values, which are the objects that the macro construct "reads as". Zero values causes the macro construct to be ignored (the semicolon macro character does this), and one value causes the macro construct to read as a single object (most macro characters do this). More than one value is allowed only within a list.

Macro characters may be *terminating* or *non-terminating*. A non-terminating macro character is only recognized as a macro character when it appears at the beginning of a token. If it appears when a token is already in progress, it is treated as a symbol constituent. Of the standard macro characters, all but # are terminating.

One kind of macro character is the *dispatch* macro character. This kind of character is handled by reading one more character, converting it to upper case, and looking it up in a table. Thus, the dispatch macro character is the start of a two-character sequence, with which is associated a defining function. # is the only standardly defined dispatch macro character.

When a dispatch macro character is used, it may be followed by a decimal integer which serves as a parameter. The character for the dispatch is actually the first non-digit seen.

The defining function for a dispatch macro two-character sequence is almost like that of an ordinary macro character. However, it receives one more argument. This is the parameter, the decimal integer that followed the dispatch macro character, or **nil** if no parameter was written. Also, the second argument is the subdispatch character, the second character of the sequence. The dispatch macro character itself is not available.

set-macro-character *char function* &optional *non-terminating-p in-readtable*

Sets the syntax of character *char* in readtable *in-readtable* to be that of a macro character which is handled by *function*. When that character is read by **read**, *function* is called.

char is made a non-terminating macro character if *non-terminating-p* is non-nil, a terminating one otherwise.

get-macro-character *char in-readtable*

Returns two values that describe the macro character status of *char* in *in-readtable*. If *char* is not a macro character, both values are **nil**. Otherwise, the first value is the *function* and the second value is the *non-terminating-p* for this character.

Those two values, passed to **set-macro-character**, are usually sufficient to recreate exactly the syntax *char* has now; however, since one of the arguments that the function receives is the macro character that invoked it, it may not behave the same if installed on a different character or in a different readtable. In particular, the definition of a dispatch macro character is a standard function that looks the macro character up in the readtable. Thus, the definition only records that the macro character *is* a dispatch macro character; it does not say what subcharacters are allowed or what they mean.

make-dispatch-macro-character *char* &optional *non-terminating-p in-readtable*

Makes *char* be a dispatch macro character in *in-readtable*. This means that when *char* is seen read will read one more character to decide what to do. # is an example of a dispatch macro character. *non-terminating-p* means the same thing as in set-macro-character.

set-dispatch-macro-character *char subchar function* &optional *in-readtable*

Sets the syntax of the two-character sequence *char subchar*, assuming that *char* is already a dispatch macro character. *function* becomes the defining function for this sequence.

If *subchar* is lower case, it is converted to upper case. Case is never significant for the character that follows a dispatch macro character. The decimal digits may not be defined as subchars since they are always used for infix numeric arguments as in #5r.

get-dispatch-macro-character *char subchar* &optional *in-readtable*

Returns the *function* for *subchar* following dispatch macro character *char* in readtable *in-readtable*. The value is nil if *subchar* is not defined for following *char*.

These subroutines are for use by the defining functions of macro characters. Ordinary read should not be used for reading subexpressions, and the ordinary :tyi operation or functions read-char or tyi should not be used for single-character input. The functions below should be used instead.

si:read-recursive &optional *stream*

Equivalent to (cli:read *stream* t nil t). See page 531. This is the recommended way for a macro character's defining function to read a subexpression.

si:xr-xrtyi *stream ignore-whitespace no-chars-special no-multiple-escapes*

Reads the next input character from *stream*, for a macro character's defining function. If *ignore-whitespace* is non-nil, any whitespace characters seen are discarded and the first non-whitespace character is returned.

The first value is the character as translated; the third value is the original character, before translation. The second value is a syntax code which is of no interest to users except to be passed to si:xr-xruntyi if this character must be unread.

Normally, this function processes all the escape characters, and performs translations (such as from lower case letters to upper case letters) on characters not escaped. Font specifiers (epsilons followed by digits or *) are ignored if the file is formatted using them.

If *no-multiple-escapes* is non-nil, multiple escapes (vertical bar characters) are not processed; they are returned to the caller. This mode is used for reading the contents of strings. If *no-chars-special* is non-nil, no escape characters are processed. All characters are simply returned to the caller (except that font specifiers are still discarded if appropriate).

si:xr-xruntyi *stream char num*

Unreads *char*, for a macro character's defining function. *char* should be the third value returned by the last call to **si:xr-xrtyi**, and *num* should be the second value.

read-suppress *Variable*

If this variable is non-**nil**, all the standard read functions and macro characters do their best to avoid any errors, and any side effects except for removing characters from the input stream. For example, symbols are not interned to avoid either errors (for nonexistent packages) or side effects (adding new symbols to packages). In fact, **nil** is used in place of any symbol that is written.

User macro characters should also notice this variable when appropriate.

The purpose of the variable is to allow expressions to be skipped and discarded. The read-time conditional constructs **#+** and **#-** bind it to **t** to skip the following expression if it is not wanted.

The following functions for defining macro characters are more or less obsolete.

set-syntax-macro-char *char function* &optional *readtable*

Changes *readtable* so that *char* is a macro character. When *char* is read, *function* is called. *readtable* defaults to the current readtable.

function is called with two arguments, *list-so-far* and the input stream. When a list is being read, *list-so-far* is that list (**nil** if this is the first element). At the top level of **read**, *list-so-far* is the symbol **:toplevel**. After a dotted-pair dot, *list-so-far* is the symbol **:after-dot**. *function* may read any number of characters from the input stream and process them however it likes.

function should return three values, called *thing*, *type*, and *splice-p*. *thing* is the object read. If *splice-p* is **nil**, *thing* is the result. If *splice-p* is non-**nil**, then when reading a list *thing* replaces the list being read—often it will be *list-so-far* with something else **nconc**'ed onto the end. At top-level and after a dot, if *splice-p* is non-**nil** the *thing* is ignored and the macro-character does not contribute anything to the result of **read**. *type* is a historical artifact and is not really used; **nil** is a safe value. Most macro character functions return just one value and let the other two default to **nil**.

Note that the convention for values returned by *function* is different from that used for functions specified in **set-macro-character**, above. **set-syntax-macro-char** works by encapsulating *function* in a closure to convert the values to the sort that **set-macro-character** wants and then passing the closure to **set-macro-character**.

function should not have any side-effects other than on the stream and *list-so-far*. Because of the way the rubout-handler works, *function* can be called several times during the reading of a single expression in which the macro character only appears once.

char is given the same syntax that single-quote, backquote, and comma have in the initial readtable (it is called **:macro** syntax).

set-syntax-#-macro-char *char function* &optional *readtable*

Causes *function* to be called when *#char* is read. *readtable* defaults to the current readtable. The function's arguments and return values are the same as for normal macro characters, documented above. When *function* is called, the special variable si:xr-sharp-argument contains nil or a number that is the number or special bits between the # and *char*.

setsyntax-sharp-macro *character type function* &optional *readtable*

This exists only for Maclisp compatibility. set-dispatch-macro-character should be used instead. If *function* is nil, *#character* is turned off, otherwise it becomes a macro that calls *function*. *type* can be :macro, :peek-macro, :splicing, or :peek-splicing. The splicing part controls whether *function* returns a single object or a list of objects. Specifying peek causes *character* to remain in the input stream when *function* is called; this is useful if *character* is something like a left parenthesis. *function* gets one argument, which is nil or the number between the # and the *character*.

23.8 The :read and :print Stream Operations

A stream can specially handle the reading and printing of objects by handling the :read and :print stream operations. Note that these operations are optional and most streams do not support them.

If the read function is given a stream that has :read in its which-operations, then instead of reading in the normal way it sends the :read message to the stream with one argument, read's *eof-option* if it had one or a magic internal marker if it didn't. Whatever the stream returns is what read returns. If the stream wants to implement the :read operation by internally calling read, it must use a different stream that does not have :read in its which-operations.

If a stream has :print in its which-operations, it may intercept all object printing operations, including those due to the print, prin1, and princ functions, those due to format, and those used internally, for instance in printing the elements of a list. The stream receives the :print message with three arguments: the object being printed, the *depth* (for comparison against the *print-level* variable), and *escape-p* (which is the value of *print-escape*). If the stream returns nil, then normal printing takes place as usual. If the stream returns non-nil, then print does nothing; the stream is assumed to have output an appropriate printed representation for the object. The two following functions are useful in this connection; however, they are in the system-internals package and may be changed without much notice.

si:print-object *object depth stream* &optional *which-operations*

Outputs the printed-representation of *object* to *stream*, as modified by *depth* and the values of the *print-... variables.

This is the internal guts of the Lisp printer. When a stream's :print handler calls this function, it should supply the list (:string-out) for *which-operations*, to prevent itself from being called recursively. Or it can supply nil if it does not want to receive :string-out messages.

If you want to customize the behavior of all printing of Lisp objects, advising (see section 30.10, page 742) this function is the way to do it. See section 23.1, page 513.

si:print-list *list depth stream which-operations*
This is the part of the Lisp printer that prints lists. A stream's :print handler can call this function, passing along its own arguments and its own which-operations, to arrange for a list to be printed the normal way and the stream's :print hook to get a chance at each of the list's elements.

24. Naming of Files

A Lisp Machine generally has access to many file systems. While it may have its own file system on its own disks, usually a community of Lisp Machine users want to have a shared file system accessible by any of the Lisp Machines over a network. These shared file systems can be implemented by any computer that is capable of providing file system service. A file server computer may be a special-purpose computer that does nothing but service file system requests from computers on a network, or it may be a time-sharing system.

Programs need to use names to designate files within these file systems. The main difficulty in dealing with names of files is that different file systems have different naming formats for files. For example, in the ITS file system, a typical name looks like:

```
DSK: GEORGE; FOO QFASL
```

with DSK being a device name, GEORGE being a directory name, FOO being the first file name and QFASL being the second file name. However, in TOPS-20, a similar file name is expressed as:

```
PS:<GEORGE>FOO.QFASL
```

It would be unreasonable for each program that deals with file names to be expected to know about each different file name format that exists, or new formats that could get added in the future. However, existing programs should retain their abilities to manipulate the names.

The functions and flavors described in this chapter exist to solve this problem. They provide an interface through which a program can deal with names of files and manipulate them without depending on anything about their syntax. This lets a program deal with multiple remote file servers simultaneously, using a uniform set of conventions.

24.1 Pathnames

All file systems dealt with by the Lisp Machine are mapped into a common model, in which files are named by something called a *pathname*. A pathname always has six components, each with a standard meaning. These components are the common interface that allows programs to work the same way with different file systems; the mapping of the pathname components into the concepts peculiar to each file system is taken care of by the pathname software. Pathname components are described in the following section, and the mappings between components and user syntax is described for each file system later in this chapter.

pathnamep *object*
: t if *object* is a pathname.

A pathname is an instance of a flavor (see chapter 21, page 401); exactly which flavor depends on what the host of the pathname is, but pathname is always one of its component flavors. If *p* is a pathname, then (typep *p* 'pathname) returns t. One of the messages handled by host objects is the :pathname-flavor operation, which returns the name of the flavor to use for pathnames on that host. And one of the differences between host flavors is how they handle this operation.

There are functions for manipulating pathnames, and there are also messages that can be sent to them. These are described later in this chapter.

Two important operations of the pathname system are *parsing* and *merging*. Parsing is the conversion of a string—which might be something typed in by the user when asked to supply the name of a file—into a pathname object. This involves finding out what host the pathname is for, then using the file name syntax conventions of that host to parse the string into the standard pathname components. Merging is the operation that takes a pathname with missing components and supplies values for those components from a set of defaults.

The function **string**, applied to a pathname, converts it into a string that is in the file name syntax of its host's file system, except that the name of the host followed by a colon is inserted at the front. This is the inverse of parsing. **princ** of a pathname also does this, then prints the contents of the string. Flavor operations such as **:string-for-dired** exist which convert all or part of a pathname to a string in other fashions that are designed for specific applications. **prin1** of a pathname prints the pathname using the `#c` syntax so it can be read back in to produce an equivalent pathname (or the same pathname, if read in the same session).

Since each kind of file server can have its own character string representation of names of its files, there has to be a different parser for each of these representations, capable of examining such a character string and figuring out what each component is. The parsers all work differently. How can the parsing operation know which parser to use? The first thing that the parser does is to figure out which host this filename belongs to. A filename character string may specify a host explicitly by having the name of the host, followed by a colon, at either the beginning or the end of the string. For example, the following strings all specify hosts explicitly:

```
AI: COMMON; GEE WHIZ              ; This specifies host AI.
COMMON; GEE WHIZ AI:              ; So does this.
AI: ARC: USERS1; FOO BAR          ; So does this.
ARC: USERS1; FOO BAR AI:          ; So does this.
EE:PS:<COMMON>GEE.WHIZ.5          ; This specifies host EE.
PS:<COMMON>GEE.WHIZ.5 EE:         ; So does this.
```

If the string does not specify a host explicitly, the parser chooses a host by default and uses the syntax for that host. The optional arguments passed to the parsing function (**fs:parse-pathname**) tell it which host to assume. Note: the parser is not confused by strings starting with **DSK:** or **PS:** because it knows that neither of those is a valid host name. But if the default host has a device whose name happens to match the name of some host, you can prevent the device name from being misinterpreted as a host name by writing an extra colon at the beginning of the string: For example, **:EE:<RMS>FOO.BAR** refers to the device **EE** on the default host (assumed to use TOPS-20 syntax) rather than to the host named **EE**.

Pathnames are kept unique, like symbols, so that there is only one object with a given set of components. This is useful because a pathname object has a property list (see section 5.10, page 113) on which you can store properties describing the file or family of files that the pathname represents. The uniqueness implies that each time the same components are typed in, the program gets the same pathname object and finds there the properties it ought to find.

Note that a pathname is not necessarily the name of a specific file. Rather, it is a way to get to a file; a pathname need not correspond to any file that actually exists, and more than one pathname can refer to the same file. For example, the pathname with **:newest** as its version refers to the same file as a pathname which has the appropriate number as the version. In systems with links, multiple file names, logical devices, etc., two pathnames that look quite different may really turn out to address the same file. To get from a pathname to a file requires doing a file system operation such as **open**.

When you want to store properties describing an individual file, use the pathname you get by sending **:truename** to a stream rather than the pathname you open. This avoids problems with different pathnames that refer to the same file.

To get a unique pathname object representing a family of files, send the message **:generic-pathname** to a pathname for any file in the family (see section 24.5, page 561).

24.2 Pathname Components

These are the components of a pathname. They are clarified by an example below.

host	An object that represents the file system machine on which the file resides. A host object is an instance of a flavor one of whose components is **si:basic-host**. The precise flavor varies depending on the type of file system and how the files are to be accessed.
device	Corresponds to the "device" or "file structure" concept in many host file systems.
directory	The name of a group of related files belonging to a single user or project. Corresponds to the "directory" concept in many host file systems.
name	The name of a group of files that can be thought of as conceptually the "same" file. Many host file systems have a concept of "name" which maps directly into this component.
type	Corresponds to the "filetype" or "extension" concept in many host file systems. This says what kind of file this is; such as, a Lisp source file, a QFASL file, etc.
version	Corresponds to the "version number" concept in many host file systems. This is a number that increments every time the file is modified. Some host systems do not support version numbers.

As an example, consider a Lisp program named **CONCH**. If it belongs to **GEORGE**, who uses the **FISH** machine, the host would be the host-object for the machine **FISH**, the device would probably be the default and the directory would be **GEORGE**. On this directory would be a number of files related to the **CONCH** program. The source code for this program would live in a set of files with name **CONCH**, type **LISP**, and versions **1**, **2**, **3**, etc. The compiled form of the program would live in files named **CONCH** with type **QFASL**; each would have the same version number as the source file that it came from. If the program had a documentation file, it would have type **INFO**.

Not all of the components of a pathname need to be specified. If a component of a pathname is missing, its value is **nil**. Before a file server can do anything interesting with a file, such as opening the file, all the missing components of a pathname must be filled in from defaults. But pathnames with missing components are often handed around inside the machine, since almost all pathnames typed by users do not specify all the components explicitly. The host is not allowed to be missing from any pathname; since the behavior of a pathname is host-dependent to some extent, it has to know what its host is. All pathnames have host attributes, even if the string being parsed does not specify one explicitly.

A component of a pathname can also be the special symbol :unspecific. :unspecific means, explicitly, "this component has been specified as missing", whereas nil means that the component was not specified and should default. In merging, :unspecific counts as a specified component and is not replaced by a default. :unspecific does *not* mean "unspecified"; it is unfortunate that those two words are similar.

:unspecific is used in *generic* pathnames, which refer not to a file but to a whole family of files. The version, and usually the type, of a generic pathname are :unspecific. Another way :unspecific is used has to do with mapping of pathnames into file systems such as ITS that do not have all six components. A component that is really "not there" is :unspecific in the pathname. When a pathname is converted to a string, nil and :unspecific both cause the component not to appear in the string.

A component of a pathname can also be the special symbol :wild. This is useful only when the pathname is being used with a directory primitive such as fs:directory-list (see page 598), where it means that this pathname component matches anything. The printed representation of a pathname usually designates :wild with an asterisk; however, this is host-dependent.

What values are allowed for components of a pathname depends, in general, on the pathname's host. However, in order for pathnames to be usable in a system-independent way certain global conventions are adhered to. These conventions are stronger for the type and version than for the other components, since the type and version are actually understood by many programs, while the other components are usually just treated as something supplied by the user that only needs to be remembered.

In general, programs can interpret the components of a pathname independent of the file system; and a certain minimum set of possible values of each component are supported on all file systems. The same pathname component value may have very different representations when the pathname is made into a string, depending on the file system. This does not affect programs that operate on the components. The user, when asked to type a pathname, always uses the system-dependent string representation. This is convenient for the user who moves between using the Lisp Machine on files stored on another host and making direct use of that host. However, when the mapping between string form and components is complicated, the components may not be obvious from what you type.

The type is always a string, or one of the special symbols nil, :unspecific, and :wild. Certain hosts impose a limit on the size of string allowed, often very small. Many programs that deal with files have an idea of what type they want to use. For example, Lisp source programs are usually "LISP", compiled Lisp programs are "QFASL", etc. However, these file type conventions are host-specific, for the important reason that some hosts do not allow a string five characters long to be used as the type. Therefore, programs should use a *canonical type* rather than an actual string to specify their conventional default file types. Canonical types are described below.

For the version, it is always legitimate to use a positive fixnum, or certain special symbols. nil, :unspecific, and :wild have been explained above. The other standardly allowed symbols are :newest and :oldest. :newest refers to the largest version number that exists when reading a file, or that number plus one when writing a new file. :oldest refers to the smallest version number that exists. Some file systems may define other special version symbols, such as :installed for example, or may allow negative numbers. Some do not support versions at all. Then a pathname may still contain any of the standard version components, but it does not matter what the value is.

The device, directory, and name are more system-dependent. These can be strings (with host-dependent rules on allowed characters and length) or they can be *structured*. A structured component is a list of strings. This is used for file system features such as hierarchical directories. The system is arranged so that programs do not need to know about structured components unless they do host-dependent operations. Giving a string as a pathname component to a host that wants a structured value converts the string to the appropriate form. Giving a structured component to a host that does not understand them converts it to a string by taking the first element and ignoring the rest.

Some host file systems have features that do not fit into this pathname model. For instance, directories might be accessible as files, there might be complicated structure in the directories or names, or there might be relative directories, such as '<' in Multics. These features appear in the parsing of strings into pathnames, which is one reason why the strings are written in host-dependent syntax. Pathnames for hosts with these features are also likely to handle additional messages besides the common ones documented in this chapter, for the benefit of host-dependent programs that want to access those features. However, once your program depends on any such features, it will work only for certain file servers and not others; in general, it is a good idea to make your program work just as well no matter what file server is being used.

24.2.1 Raw Components and Interchange Components

On some host file systems it is conventional to use lower-case letters in file names, while in others upper case is customary, or possibly required. When pathname components are moved from pathnames of one file system to pathnames of another file system, it is useful to convert the case if necessary so that you get the right case convention for the latter file system as a default. This is especially useful when copying files from one file system to another.

The Lisp Machine system defines two representations for each of several pathname components (the device, directory, name and type). There is the *raw* form, which is what actually appears in the filename on the host file system, and there is the *interchange* form, which may differ in alphabetic case from the raw form. The raw form is what is stored inside the pathname object itself, but programs nearly always operate on the interchange form. The :name, :type, etc., operations return the interchange form, and the :new-name, etc., operations expect the interchange form. Additional operations :raw-name, etc., are provided for working with the raw components, but these are rarely needed.

The interchange form is defined so that it is always customarily in upper case. If upper case is customary on the host file system, then the interchange form of a component is the same as the raw form. If lower case is customary on the host file system, as on Unix, then the interchange form has case inverted. More precisely, an all-upper-case component is changed to all-lower-case, an all-lower-case component is changed to all-upper-case, and a mixed-case component is not changed. (This is a one-to-one mapping). Thus, a Unix pathname with a name component of "foo" has an interchange-format name of "FOO", and vice versa.

For host file systems which record case when files are created but ignore case when comparing filenames, the interchange form is always upper case.

The host component is not really a name, and case is always ignored in host names, so there is no need for two forms of host component. The version component does not need them either, because it is never a string.

24.2.2 Pathname Component Operations

:host *Operation on* pathname
:device *Operation on* pathname
:directory *Operation on* pathname
:name *Operation on* pathname
:type *Operation on* pathname
:version *Operation on* pathname

These return the components of the pathname, in interchange form. The returned values can be strings, special symbols, or lists of strings in the case of structured components. The type is always a string or a symbol. The version is always a number or a symbol.

:raw-device *Operation on* pathname
:raw-directory *Operation on* pathname
:raw-name *Operation on* pathname
:raw-type *Operation on* pathname

These return the components of the pathname, in raw form.

:new-device *dev* *Operation on* pathname
:new-directory *dir* *Operation on* pathname
:new-name *name* *Operation on* pathname
:new-type *type* *Operation on* pathname
:new-version *version* *Operation on* pathname

These return a new pathname that is the same as the pathname they are sent to except that the value of one of the components has been changed. The specified component value is interpreted as being in interchange form, which means its case may be converted. The :new-device, :new-directory and :new-name operations accept a string (or a special symbol) or a list that is a structured name. If the host does not define structured components, and you specify a list, its first element is used.

:new-raw-device *dev* *Operation on* pathname
:new-raw-directory *dir* *Operation on* pathname
:new-raw-name *name* *Operation on* pathname
:new-raw-type *type* *Operation on* pathname

These return a new pathname that is the same as the pathname they are sent to except that the value of one of the components has been changed. The specified component value is interpreted as raw.

:new-suggested-name *name* — *Operation on* pathname
:new-suggested-directory *dir* — *Operation on* pathname

These differ from the :new-name and :new-directory operations in that the new pathname constructed has a name or directory based on the suggestion, but not necessarily identical to it. It tries, in a system-dependent manner, to adapt the suggested name or directory to the usual customs of the file system in use.

For example, on a TOPS-20 system, these operations would convert *name* or *dir* to upper case, because while lower-case letters *may* appear in TOPS-20 pathnames, it is not customary to generate such pathnames by default.

:new-pathname &rest *options* — *Operation on* pathname

This returns a new pathname that is the same as the pathname it is sent to except that the values of some of the components have been changed. *options* is a list of alternating keywords and values. The keywords all specify values of pathname components; they are :host, :device, :directory, :name, :type, and :version. Alternatively, the keywords :raw-device, :raw-directory, :raw-name and :raw-type may be used to specify a component in raw form.

Two additional keywords, :canonical-type and :original-type, allow the type field to be specified as a canonical type. See the following section for a description of canonical types. Also, the value specified for the keyword :type may be a canonical type symbol.

If an invalid component is specified, it is replaced by some valid component so that a valid pathname can be returned. You can tell whether a component is valid by specifying it in :new-pathname and seeing whether that component of the resulting pathname matches what you specified.

The operations :new-name, etc., are equivalent to :new-pathname specifying only one component to be changed; in fact, that is how those operations are implemented.

24.2.3 Canonical Types

Canonical types are a way of specifying a pathname type component using host-dependent conventions without making the program itself explicitly host dependent. For example, the function compile-file normally provides a default type of "LISP", but on VMS systems the default must be "LSP" instead, and on Unix systems it is "l". What compile-file actually does is to use a canonical type, the keyword :lisp, as the default. This keyword is given a definition as a canonical type, which specifies what it maps into on various file systems.

A single canonical type may have more than one mapping on a particular file system. For example, on TOPS-20 systems the canonical type :LISP maps into either "LISP" or "LSP". One of the possibilities is marked as "preferred"; in this case, it is "LISP". The effect of this is that either FOO.LISP or FOO.LSP would be acceptable as having canonical type :lisp, but merging yields "LISP" as the type when defaulting from :lisp.

Note that the canonical type of a pathname is not a distinct component. It is another way of describing or specifying the type component.

A canonical type must be defined before it is used.

fs:define-canonical-type *Macro*
symbol standard-mapping system-dependent-mappings...

Defines *symbol* as a canonical type. *standard-mapping* is the actual type component that it maps into (a string), with exceptions as specified by *system-dependent-mappings*. Each element of *system-dependent-mappings* (that is, each additional argument) is a list of the form

(*system-type preferred-mapping other-mappings*...)

system-type is one of the system-type keywords the :system-type operation on a host object can return, such as :unix, :tops20, and :lispm (see page 577). The argument describes how to map this canonical type on that type of file system. *preferred-map* (a string) is the preferred mapping of the canonical type, and *other-mappings* are additional strings that are accepted as matching the canonical type.

system-type may also be a list of system types. Then the argument applies to all of those types of file systems.

All of the mapping strings are in interchange form.

For example, the canonical type :lisp is defined as follows:

```
(fs:define-canonical-type :lisp "LISP"
  (:unix "L" "LISP")
  (:vms "LSP")
  ((:tops20 :tenex) "LISP" "LSP"))
```

Other canonical types defined by the system include :qfasl, :text, :press, :qwabl, :babyl, :mail, :xmail, :init, :patch-directory, :midas, :palx, :unfasl, :widths, :output, mac, tasm, doc, mss, tex, pl1 and clu. The standard mapping for each is the symbol's pname.

To match a pathname against a canonical type, use the :canonical-type operation.

:canonical-type *Operation on* pathname

Returns two values which describe whether and how this pathname's type component matches any canonical type.

If the type component is one of the possible mappings of some canonical type, the first value is that canonical type (the symbol). The second value is nil if the type component is the preferred mapping of the canonical type; otherwise it is the actual type component, in interchange form. The second value is called the *original type* of the pathname.

If the type component does not match a canonical type, the first value is the type component in interchange form (a string), and the second value is nil.

This operation is useful in matching a pathname against a canonical type; the first value is eq to the canonical type if the pathname matches it. The operation is also useful for transferring a type field from one file system to another while preserving canonical type; this is described below.

A new pathname may also be constructed by specifying a canonical type.

:new-canonical-type *canonical-type* &optional *original-type* — *Operation on* pathname

Returns a pathname different from this one in having a type component that matches *canonical-type*.

If *original-type* is a possible mapping for *canonical-type* on this pathname's host, then it is used as the type component. Otherwise, the preferred mapping for *canonical-type* is used. If *original-type* is not specified, it defaults to this pathname's type component. If it is specified as nil, the preferred mapping of the canonical type is always used. If *canonical-type* is a string rather than an actual canonical type, it is used directly as the type component, and the *original-type* does not matter.

The :new-pathname operation accepts the keywords :canonical-type and :original-type. The :new-canonical-type operation is equivalent to :new-pathname with those keywords.

Suppose you wish to copy the file named *old-pathname* to a directory named *target-directory-pathname*, possibly on another host, while preserving the name, version and canonical type. That is, if the original file has a name acceptable for a QFASL file, the new file should also. Here is how to compute the new pathname:

```
(multiple-value-bind (canonical original)
    (send old-pathname :canonical-type)
  (send target-directory-pathname :new-pathname
        :name (send old-pathname :name)
        :version (send old-pathname :version)
        :canonical-type canonical
        :original-type original))
```

Suppose that *old-pathname* is OZ:<FOO>A.LISP.5, where OZ is a TOPS-20, and the target directory is on a VMS host. Then canonical is :lisp and original is "LISP". Since "LISP" is not an acceptable mapping for :lisp on a VMS system, the resulting pathname has as its type component the preferred mapping for :lisp on VMS, namely, "LSP".

But if the target host is a Unix host, the new file's type is "LISP", since that is an acceptable (though not preferred) mapping for :lisp on Unix hosts. If you would rather that the preferred mapping always be used for the new file's type, omit the :original-type argument to the :new-pathname operation. This would result in a type component of "L" in interchange form, or "l" in raw form, in the new file's pathname.

The function compile-file actually does something cleverer than using the canonical type as a default. Doing that, and opening the resulting pathname, would look only for the preferred mapping of the canonical type. compile-file actually tries to open *each* possible mapping, trying the preferred mapping first. Here is how it does so:

:open-canonical-default-type *Operation on* pathname
canonical-type &rest *options*

If this pathname's type component is non-**nil**, the pathname is simply opened, passing the *options* to the :open operation. If the type component is nil, each mapping of *canonical-type* is tried as a type component, in the order the mappings appear in the canonical type definition. If an open succeeds, a stream is returned. The possibilities continue to be tried as long as fs:file-not-found errors happen; other errors are not handled. If all the possibilities fail, a fs:file-not-found error is signaled for the caller, with a pathname that contains the preferred mapping as its type component.

24.3 Defaults and Merging

When the user is asked to type in a pathname, it is of course unreasonable to require the user to type a complete pathname, containing all components. Instead there are defaults, so that components not specified by the user can be supplied automatically by the system. Each program that deals with pathnames typically has its own set of defaults.

The system defines an object called a *defaults alist*. Functions are provided to create one, get the default pathname out of one, merge a pathname with one, and store a pathname back into one. A defaults alist can remember more than one default pathname if defaults are being kept separately for each host; this is controlled by the variable fs:*defaults-are-per-host*. The main primitive for using defaults is the function fs:merge-pathname-defaults (see page 558).

In place of a defaults alist, you may use just a pathname. Defaulting one pathname from another is useful for cases such as a program that has an input file and an output file, and asks the user for the name of both, letting the unsupplied components of one name default from the other. Unspecified components of the output pathname come from the input pathname, except that the type should default not to the type of the input but to the appropriate default type for output from this program.

The implementation of a defaults alist is an association list of host names and default pathnames. The host name nil is special and holds the defaults for all hosts, when defaults are not per-host.

The *merging* operation takes as input a pathname, a defaults alist (or another pathname), a default type, and a default version, and returns a pathname. Basically, the missing components in the pathname are filled in from the defaults alist. However, if a name is specified but the type or version is not, then the type or version is treated specially.

Here are the merging rules in full detail.

If no host is specified, the host is taken from the defaults. If the pathname explicitly specifies a host and does not supply a device, then the the default file device for that host is used.

If the pathname specifies a device named DSK, that is replaced with the *working device* for the pathname's host, and the directory defaults to the *working directory* for the host if it is not specified. See fs:set-host-working-directory, below.

Next, if the pathname does not specify a host, device, directory, or name, that component comes from the defaults.

If the value of fs:*always-merge-type-and-version* is non-nil, the type and version are merged just like the other components.

If fs:*always-merge-type-and-version* is nil, as it normally is, the merging rules for the type and version are more complicated and depend on whether the pathname specifies a name. If the pathname doesn't specify a name, then the type and version, if not provided, come from the defaults, just like the other components. However, if the pathname does specify a name, then the type and version come from the *default-type* and *default-version* arguments to merge-pathname-defaults. If those arguments were omitted, the value of fs:*name-specified-default-type* (initially, :lisp) is used as the default type, and :newest is used as the default version.

The reason for this is that the type and version "belong to" some other filename, and are thought to be unlikely to have anything to do with the new filename you are typing in.

fs:set-host-working-directory *host pathname*

Sets the *working device* and *working directory* for *host* to those specified in *pathname*. *host* should be a host object or the name of a host. *pathname* may be a string or a pathname. The working device and working directory are used for defaulting pathnames in which the device is specified as DSK.

The editor command Meta-X Set Working Directory provides a convenient interface to this function.

The following special variables are parts of the pathname interface that are relevant to defaults.

fs:*defaults-are-per-host* *Variable*

This is a user customization option intended to be set by a user's LISPM INIT file (see section 35.8, page 800). The default value is nil, which means that each program's set of defaults contains only one default pathname. If you type in just a host name and a colon, the other components of the name default from the previous host, with appropriate translation to the new host's pathname syntax. If fs:*defaults-are-per-host* is set to t, each program's set of defaults maintains a separate default pathname for each host. If you type in just a host name and a colon, the last file that was referenced on that host is used.

fs:*always-merge-type-and-version* *Variable*

If this variable is non-nil, then the type and version are defaulted only from the pathname defaults just like the other components.

fs:*name-specified-default-type* *Variable*

If fs:*always-merge-type-and-version* is nil, then when a name is specified but not a type, the type defaults from an argument to the merging function. If that argument is not specified, this variable's value is used. It may be a string or a canonical type keyword. The value is initially :lisp.

default-pathname-defaults *Variable*

This is the default defaults alist: if the pathname primitives that need a set of defaults are not given one, they use this one. Most programs, however, should have their own defaults rather than using these.

cli:*default-pathname-defaults* *Variable*

The Common Lisp version of the default pathname defaults. The value of this variable is a pathname rather than an alist. This variable is magically (with a forwarding pointer) identified with a cell in the defaults-alist which the system really uses, so that setting this variable modifies the contents of the alist.

fs:last-file-opened *Variable*

This is the pathname of the last file that was opened. Occasionally this is useful as a default. Since some programs deal with files without notifying the user, you must not expect the user to know what the value of this symbol is. Using this symbol as a default may cause unfortunate surprises if you don't announce it first, and so such use is discouraged.

These functions are used to manipulate defaults alists directly.

fs:make-pathname-defaults

Creates a defaults alist initially containing no defaults. If you ask this empty set of defaults for its default pathname before anything has been stored into it you get the file FOO on the user's home directory on the host he logged in to.

fs:copy-pathname-defaults *defaults*

Creates a defaults alist, initially a copy of *defaults*.

fs:default-pathname &optional *defaults host default-type default-version*

This is the primitive function for getting a default pathname out of a defaults alist. Specifying the optional arguments *host*, *default-type*, and *default-version* to be non-nil forces those fields of the returned pathname to contain those values.

If fs:*defaults-are-per-host* is nil (its default value), this gets the one relevant default from the alist. If it is t, this gets the default for *host* if one is specified, otherwise for the host most recently used.

If *defaults* is not specified, the default defaults are used.

This function also has an additional optional argument *internal-p*, which is obsolete.

fs:default-host *defaults*

Returns the default host object specified by the defaults-alist *defaults*. This is the host used by pathname defaulting with the given defaults if no host is specified.

fs:set-default-pathname *pathname* &optional *defaults*

This is the primitive function for updating a set of defaults. It stores *pathname* into *defaults*. If *defaults* is not specified, the default defaults are used.

24.4 Pathname Functions

This function obtains a pathname from an object if that is possible.

pathname *object*

Converts *object* to a pathname and returns that, if possible. If *object* is a string or symbol, it is parsed. If *object* is a plausible stream, it is asked for its pathname with the :pathname operation. If *object* is a pathname, it is simply returned. Any other kind of *object* causes an error.

These functions are what programs use to parse and default file names that have been typed in or otherwise supplied by the user.

parse-namestring *thing* &optional *host defaults* &key (*start* 0) *end junk-allowed*

Is the Common Lisp function for parsing file names. It is equivalent to fs:parse-pathname except in that it takes some keyword arguments where the other function takes all positional arguments.

fs:parse-pathname *thing* &optional *host defaults* (*start* 0) *end junk-allowed*

This turns *thing*, which can be a pathname, a string, a symbol, or a Maclisp-style name list, into a pathname. Most functions that are advertised to take a pathname argument call fs:parse-pathname on it so that they can accept anything that can be turned into a pathname. If thing is itself a pathname, it is returned unchanged.

If *thing* is a string, *start* and *end* are interpreted as indices specifying a substring to parse. They are just like the second and third arguments to **substring**. The rest of *thing* is ignored. *start* and *end* are ignored if *thing* is not a string.

If *junk-allowed* is non-nil, parsing stops without error if the syntax is invalid, and this function returns nil. The second value is then the index of the invalid character. If parsing is successful, the second value is the index of the place at which parsing was supposed to stop (*end*, or the end of *thing*). If *junk-allowed* is nil, invalid syntax signals an error.

This function does *not* do defaulting, even though it has an argument named *defaults*; it only does parsing. The *host* and *defaults* arguments are there because in order to parse a string into a pathname, it is necessary to know what host it is for so that it can be parsed with the file name syntax peculiar to that host. If *thing* does not contain a manifest host name, then if *host* is non-nil, it is the host name to use, as a string. If *thing* is a string, a manifest host name may be at the beginning or the end, and consists of the name of a host followed by a colon. If *host* is nil then the host name is obtained from the default pathname in *defaults*. If *defaults* is not supplied, the default defaults (***default-pathname-defaults***) are used.

Note that if *host* is specified, and *thing* contains a host name, an error is signaled if they are not the same host.

fs:pathname-parse-error (fs:pathname-error error) *Condition*

This condition is signaled when fs:parse-pathname finds a syntax error in the string it is given.

fs:parse-pathname sets up a nonlocal proceed type :new-pathname for this condition. The proceed type expects one argument, a pathname, which is returned from fs:parse-pathname.

fs:merge-pathname-defaults *pathname* &optional *defaults default-type default-version*

Fills in unspecified components of *pathname* from the defaults and returns a new pathname. This is the function that most programs should call to process a file name supplied by the user. *pathname* can be a pathname, a string, a symbol, or a Maclisp namelist. The returned value is always a pathname. The merging rules are documented on page 554.

If *defaults* is a pathname, rather than a defaults alist, then the defaults are taken from its components. This is how you merge two pathnames. (In Maclisp that operation is called mergef.)

defaults defaults to the value of *default-pathname-defaults* if unsupplied. *default-type* defaults to the value of fs:*name-specified-default-type*. *default-version* defaults to :newest.

merge-pathnames *pathname* &optional *defaults* (*default-version* :newest)

Is the Common Lisp function for pathname defaulting. It does only some of the things that fs:merge-pathname-defaults does. It merges defaults from *defaults* (which defaults to the value of *default-pathname-defaults*) into *pathname* to get a new pathname, which is returned. *pathname* can be a string (or symbol); then it is parsed and the result is defaulted. *default-version* is used as the version when pathname has a name but no version.

fs:merge-and-set-pathname-defaults *pathname* &optional *defaults default-type default-version*

This is the same as fs:merge-pathname-defaults except that after it is done the defaults-list *defaults* is modified so that the merged pathname is the new default. This is handy for programs that have sticky defaults, which means that the default for each command is the last filename used. (If *defaults* is a pathname rather than a defaults alist, then no storing back is done.) The optional arguments default the same way as in fs:merge-pathname-defaults.

These functions convert a pathname into a namestring for all or some of the pathname's components.

namestring *pathname*

Returns a string containing the printed form of *pathname*, as you would type it in. This uses the :string-for-printing operation.

file-namestring *pathname*

Returns a string showing just the name, type and version of *pathname*. This uses the :string-for-dired operation.

directory-namestring *pathname*

Returns a string showing just the device and directory of *pathname*. This uses the :string-for-directory operation.

enough-namestring *pathname* &optional *defaults*

Returns a string showing just the components of *pathname* which would not be obtained by defaulting from *defaults*. This is the shortest string that would suffice to specify pathname, given those defaults. It is made by using the :string-for-printing operation on a modified pathname.

This function yields a pathname given its components.

make-pathname &key (*defaults* t) *host device raw-device directory raw-directory name raw-name type raw-type version canonical-type original-type*

Returns a pathname whose components are as specified.

If *defaults* is a pathname or a defaults-alist, any components not explicitly specified default from it. If *defaults* is t (which is the default), then unspecified components default to **nil**, except for the host (since every pathname must have a specific host), which defaults based on *default-pathname-defaults*.

These functions give the components of a pathname.

pathname-host *pathname*

Returns the host component of *pathname*.

pathname-device *pathname*
pathname-directory *pathname*
pathname-name *pathname*
pathname-type *pathname*
pathname-version *pathname*

Likewise, for the other components

These functions return useful information.

fs:user-homedir &optional *host reset-p* (*user* user-id) *force-p*
user-homedir-pathname &optional *host reset-p* (*user* user-id) *force-p*

Returns the pathname of the *user* 's home directory on *host*. These default to the logged in user and the host logged in to. Home directory is a somewhat system-dependent concept, but from the point of view of the Lisp Machine it is the directory where the user keeps personal files such as init files and mail.

This function returns a pathname without any name, type, or version component (those components are all **nil**).

If *reset-p* is specified non-nil, the machine the user is logged in to is changed to be *host*.

The synonym **user-homedir-pathname** is from Common Lisp.

init-file-pathname *program-name* &optional *host*

Returns the pathname of the logged-in user's init file for the program *program-name*, on the *host*, which defaults to the host the user logged in to. Programs that load init files containing user customizations call this function to find where to look for the file, so that they need not know the separate init file name conventions of each host operating system. The *program-name* "LISPM" is used by the **login** function.

These functions are useful for poking around.

fs:describe-pathname *pathname*

If *pathname* is a pathname object, this describes it, showing you its properties (if any) and information about files with that name that have been loaded into the machine. If *pathname* is a string, this describes all interned pathnames that match that string, ignoring components not specified in the string. One thing this is useful for is finding the directory of a file whose name you remember. Giving **describe** (see page 791) a pathname object invokes this function.

fs:pathname-plist *pathname*

Parses and defaults *pathname*, then returns the list of properties of that pathname.

fs:*pathname-hash-table* *Variable*

This is the hash table in which pathname objects are interned. You can find all pathnames ever constructed by applying the function **maphash** to this hash table.

24.5 Generic Pathnames

A generic pathname stands for a whole family of files. The property list of a generic pathname is used to remember information about the family, some of which (such as the package) comes from the -*- line (see section 25.5, page 594) of a source file in the family. Several types of files with that name, in that directory, belong together. They are different members of the same family; for example, they may be source code and compiled code. However, there may be several other types of files that form a logically distinct group even though they have this same name; TEXT and PRESS for example. The exact mapping is done on a per host basis since it can sometimes be affected by host naming conventions.

The generic pathname of pathname *p* usually has the same host, device, directory, and name as *p* does. However, it has a version of :unspecific. The type of the generic pathname is obtained by sending a :generic-base-type *type-of-p* message to the host of *p*. The default response to this message is to return the associated type from fs:*generic-base-type-alist* if there is one, else *type-of-p*. Both the argument and the value are either strings, in interchange form, or canonical type symbols.

However, the ITS file system presents special problems. One cannot distinguish multiple generic base types in this same way since the type component does not exist as such; it is derived from the second filename, which unfortunately is also sometimes used as a version number. Thus, on ITS, the type of a generic pathname is always :unspecific if there is any association for the type of the pathname on fs:*generic-base-type-alist*.

Since generic pathnames are primarily useful for storing properties, it is important that they be as standardized and conceptualized as possible. For this reason, generic pathnames are defined to be backtranslated, i.e. the generic pathname of a pathname that is (or could be) the result of a logical host translation has the host and directory of the logical pathname. For example, the generic pathname of OZ:<L.WINDOW>;STREAM LISP would be SYS:WINDOW;STREAM U U if OZ is the system host.

All version numbers of a particular pathname share the same identical generic pathname. If the values of particular properties have changed between versions, it is possible for confusion to result. One way to deal with this problem is to have the property be a list associating version number with the actual desired property. Then it is relatively easy to determine which versions have which values for the property in question and select one appropriately. But in the applications for which generic pathnames are typically used, this is not necessary.

The :generic-pathname operation on a pathname returns its corresponding generic pathname. See page 563. The :source-pathname operation on a pathname returns the actual or probable pathname of the corresponding source file (with :newest as the version). See page 563.

fs:*generic-base-type-alist* *Variable*

This is an association list of the file types and the type of the generic pathname used for the group of which that file type is a part. Constructing a generic pathname replaces the file type with the association from this list, if there is one (except that ITS hosts always replace with :unspecific). File types not in this list are really part of the name in some sense. The initial list is

```
((:text . :text) ("DOC" . :text)
 (:press . :text) ("XGP" . :text)
 (:lisp . :unspecific) (:qfasl . :unspecific)
 (nil . :unspecific))
```

The association of :lisp and :unspecific is unfortunately made necessary by the problems of ITS mentioned previously. This way makes the generic pathnames of logically mapped LISP files identical no matter whether the logical host is mapped to an ITS host or not.

The first entry in the list with a particular cdr is the entry for the type that source files have. Note how the first element whose cdr is :unspecific is the one for :lisp. This is how the :source-pathname operation knows what to do, by default.

Some users may need to add to this list.

The system records certain properties on generic pathnames automatically.

:warnings This property is used to record compilation and other warnings for the file.

:definitions This property records all the functions and other things defined in the file. The value has one element for each package into which the file has been loaded; the element's car is the package itself and the cdr is a list of definitions made.

Each definition is a cons whose car is the symbol or function spec defined and whose cdr is the type of definition (usually one of the symbols defun, defvar, defflavor and defstruct).

:systems This property's value is a list of the names of all the systems (defined with defsystem, see page 660) of which this is a source file.

:file-id-package-alist
This property records what version of the file was most recently loaded. In case the file has been loaded into more than one package, as is sometimes necessary, the loaded version is remembered for each package separately. This is how make-system tells whether a file needs to be reloaded. The value is a list with an element for each package that the file has been loaded into; the elements look like

```
(package file-information)
```

package is the package object itself; *file-information* is the value returned by the :info operation on a file stream, and is usually a cons whose car is the truename (a pathname) and whose cdr is the file creation date (a universal time number).

Some additional properties are put on the generic pathname by reading the attribute list of the file (see page 597). It is not completely clear that this is the right place to store these properties, so it may change in the future. Any property name can appear in the attributes list and get onto the generic pathname; the standard ones are described in section 25.5, page 594.

24.6 Pathname Operations

This section documents the operations a user may send to a pathname object. Pathnames handle some additional operations that are only intended to be sent by the file system itself, and therefore are not documented here. Someone who wants to add a new host to the system would need to understand those internal operations.

The operations on pathnames that actually operate on files are documented in section 25.4, page 592. Certain pathname flavors, for specific kinds of hosts, allow additional special purpose operations. These are documented in section 24.7, page 568 in the section on the specific host type.

:generic-pathname *Operation on* pathname
Returns the generic pathname for the family of files of which this pathname is a member. See section 24.5, page 561 for documentation on generic pathnames.

:source-pathname *Operation on* pathname
Returns the pathname for the source file in the family of files to which this pathname belongs. The returned pathname has :newest as its version. If the file has been loaded in some fashion into the Lisp environment, then the pathname type is that which the user actually used. Otherwise, the conventional file type for source files is determined from the generic pathname.

:primary-device *Operation on* pathname
Returns the default device name for the pathname's host. This is used in generating the initial default pathname for a host.

Operations dealing with wildcards.

The character * in a namestring is a *wildcard*. It means that the pathname is a really a pattern which specifies a set of possible filenames rather than a single filename. The matches any sequence of characters within a single component of the name. Thus, the component FOO* would match FOO, FOOBAR, FOOT, or any other component starting with FOO.

Any component of a pathname can contain wildcards except the host; wild hosts are not allowed because a known host is required in order to know what flavor the pathname should be. If a pathname component is written in the namestring as just *, the actual component of the pathname instance is the keyword :wild. Components which contain wildcards but are not simply a single wildcard are represented in ways subject to change.

Pathnames whose components contain wildcards are called *wild* pathnames. Wild pathnames useful in functions such as delete-file for requesting the deletion of many files at once. Less obviously but more fundamentally, wild pathnames are required for most use of the function fs:directory-list; an entire directory's contents are obtained by specifying a pathname whose name, type and version components are :wild.

:wild-p *Operation on* pathname

Returns non-nil if this pathname contains any sort of wildcards. If the value is not nil, it is a keyword, one of device, :directory, :name, :type and :version, and it identifies the 'first' component which is wild.

:device-wild-p *Operation on* pathname

t if this pathname's device contains any sort of wildcards.

:directory-wild-p *Operation on* pathname
:name-wild-p *Operation on* pathname
:type-wild-p *Operation on* pathname
:version-wild-p *Operation on* pathname

Similar, for the other components that can be wild. (The host cannot ever be wild.)

:pathname-match *Operation on* pathname
candidate-pathname &optional (*match-host-p* t)

Returns t if candidate-pathname matches the pathname on which the operation is invoked (called, in this context, the *pattern pathname*). If the pattern pathname contains no wildcards, the pathnames match only if they are identical. This operation is intended in cases where wildcards are expected.

Wildcard matching is done individually by component; the operation returns t only if each component matches. Within each component, an occurrences of * in pattern pathname's component can match any sequence of characters in *candidate-pathname*'s component. Other characters, except for host-specific wildcards, must match exactly. :wild as a component of the pattern pathname matches any component that *candidate-pathname* may have.

Note that if a component of the pattern pathname is nil, *candidate-pathname*'s component must be nil also to match it. Most user programs that read pathnames and use them as patterns default unspecified components to :wild first.

Examples:

```
(defvar pattern)
(defun test (str)
  (send pattern
        :pathname-match
        (parse-namestring str)))

(setq pattern
      (parse-namestring "OZ:*:<F*O>*.TEXT.*"))

(test "OZ:<FOO>A.TEXT") => t
(test "OZ:<FO>HAHA.TEXT.3") => t
(test "OZ:<FPPO>HAHA.TEXT.*") => t
(test "OZ:<FOX>LOSE.TEXT") => nil

(setq pattern
      (parse-namestring "OZ:*:<*>A.TEXT*.5"))

(test "OZ:<FOO>A.TEXT.5") => t
(test "OZ:<FOO>A.TEXTTTT.5") => t
(test "OZ:<FOO>A.TEXT") => nil
```

If *match-host-p* is nil, then the host components of the two pathnames are not tested. The result then depends only on the other components.

:translate-wild-pathname *Operation on* pathname
target-pattern starting-data &optional *reversible*

Returns a pathname corresponding to *starting-data* under the mapping defined by the wild pathnames *source-pattern*, which is the pathname this operation is invoked on, and *target-pattern*, the argument. It is expected that *starting-data* would match the source pattern under the :pathname-match operation.

:translate-wild-pathname is used by functions such as copy-file which use one wild pathname to specify a set of files and a second wild pathname to specify a corresponding filename for each file in the set. The first wild pathname would be used as the source-pattern and the second, specifying the name to copy each file to, would be passed as the *target-pattern* pathname.

Each component of the result is computed individually from the corresponding components of *starting-data* and the pattern pathnames, using the following rules:

1) If *target-pattern*'s component is ;wild, then the result component is taken from *starting-data*.

2) Otherwise, each non-wild character in *target-pattern*'s component is taken literally into the result. Each wild character in *target-pattern*'s component is paired with a wild character in *source-pattern*'s component, and thereby with the portion of *starting-data*'s component which that matched. This portion of *starting-data* appears in the result in place of the wild target character.

Example:
```
(setq source (fs:parse-pathname "OZ:PS:<FOO>A*B*.*.*"))
(setq target (fs:parse-pathname "OZ:SS:<*>*LOSE*.*B.*"))

(send source :translate-wild-pathname target
     (fs:parse-pathname "OZ:PS:<FOO>ALIBI.LISP.3"))
 => the pathname OZ:SS:<FOO>LILOSEI.LISPB.3
```

It is easiest to understand the mapping as being done in interchange case: the interchange components of the arguments are used and the results specify the interchange components of the value.

The type component is slightly special; if the *target-pattern* type is :wild, the canonical type of *starting-data* is taken and then interpreted according to the mappings of the target host. Example:
```
(setq source (fs:parse-pathname "OZ:PS:<FOO>A*.*.*"))
(setq target (fs:parse-pathname "U://usr//foo//b*.*"))

(send source :translate-wild-pathname target
     (fs:parse-pathname "OZ:PS:<FOO>ALL.LISP"))
 => the pathname U:/usr/foo/bll.l
```

If *reversible* is non-nil, rule 1 is not used; rule 2 controls all mapping. This mode is used by logical pathname translation. It makes a difference when the target pattern component is :wild and the source pattern component contains wildcards but is not simply :wild. For example, with source and target pattern components BIG* and *, and starting data BIGGER, the result is ordinarily BIGGER by rule 1, but with reversible translation the result is GER.

Operations to get a path name string out of a pathname object:

:string-for-printing *Operation on* pathname
Returns a string that is the printed representation of the path name. This is the same as what you get if you princ the pathname or take string of it.

:string-for-wholine *length* *Operation on* pathname
Returns a string like the :string-for-printing, but designed to fit in *length* characters. *length* is a suggestion; the actual returned string may be shorter or longer than that. However, the who-line updater truncates the value to that length if it is longer.

:string-for-editor *Operation on* pathname
Returns a string that is the pathname with its components rearranged so that the name is first. The editor uses this form to name its buffers.

:string-for-dired *Operation on* pathname
Returns a string to be used by the directory editor. The string contains only the name, type, and version.

:string-for-directory *Operation on* pathname
Returns a string that contains only the device and directory of the pathname. It identifies one directory among all directories on the host.

:string-for-host *Operation on* pathname
Returns a string that is the pathname the way the host file system likes to see it.

Operations to move around through a hierarchy of directories:

:pathname-as-directory *Operation on* pathname
Assuming that the file described by the pathname is a directory, return another pathname specifying that *as* a directory. Thus, if sent to a pathname OZ:<RMS>FOO.DIRECTORY, it would return the pathname OZ:<RMS.FOO>. The name, type and version of the returned pathname are :unspecific.

:directory-pathname-as-file *Operation on* pathname
This is the inverse of the preceding operation. It returns a pathname specifying as a file the directory of the original pathname. The name, type and version of the original pathname are ignored.

The special symbol :root can be used as the directory component of a pathname on file systems that have a root directory.

Operations to manipulate the property list of a pathname:

:get *property-name* &optional *default-value* *Operation on* pathname
:getl *list-of-property-names* *Operation on* pathname
:putprop *value property-name* *Operation on* pathname
:remprop *property-name* *Operation on* pathname
:plist *Operation on* pathname
These manipulate the pathname's property list, and are used if you call the property list functions of the same names (see page 114) giving the pathname as the first argument. Please read the paragraph on page 546 explaining the care you must take in using property lists of pathnames.

24.7 Host File Systems Supported

This section lists the host file systems supported, gives an example of the pathname syntax for each system, and discusses any special idiosyncracies. More host types may be added in the future.

24.7.1 ITS

An ITS pathname looks like "*host*: *device*: *dir*; *name type-or-version*". The primary device is DSK: but other devices such as ML:, ARC:, DVR:, or PTR: may be used.

ITS does not exactly fit the virtual file system model, in that a file name has two components (FN1 and FN2) rather than three (name, type, and version). Consequently to map any virtual pathname into an ITS filename, it is necessary to decide whether the FN2 is the type or the version. The rule is that usually the type goes in the FN2 and the version is ignored; however, certain types (LISP and TEXT) are ignored and instead the version goes in the FN2. Also if the type is :unspecific the FN2 is the version.

Given an ITS filename, it is converted into a pathname by making the FN2 the version if it is '<', '>', or a number. Otherwise the FN2 becomes the type. ITS pathnames allow the special version symbols :oldest and :newest, which correspond to '<' and '>' respectively.

In every ITS pathname either the version or the type is :unspecific or nil; sometimes both are. When you create a new ITS pathname, if you specify only the version or only the type, the one not specified becomes :unspecific. If both are specified, the version is :unspecific unless the type is a normally-ignored type (such as LISP) in which case the version is :newest and the type is :unspecific so that numeric FN2's are found.

Each component of an ITS pathname is mapped to upper case and truncated to six characters.

Special characters (space, colon, and semicolon) in a component of an ITS pathname can be quoted by prefixing them with right horseshoe (⊃) or equivalence sign (≡). Right horseshoe is the same character code in the Lisp Machine character set as control-Q in the ITS character set.

An ITS pathname can have a structured name, which is a list of two strings, the FN1 and the FN2. In this case there is neither a type nor a version.

An ITS pathname with an FN2 but no FN1 (i.e. a type and/or version but no name) is represented with the placeholder FN1 '⊖', because ITS pathname syntax provides no way to write an FN2 without an FN1 before it.

The ITS init file naming convention is "*homedir*; *user program*".

fs:*its-uninteresting-types* *Variable*

The ITS file system does not have separate file types and version numbers; both components are stored in the "FN2". This variable is a list of the file types that are "not important"; files with these types use the FN2 for a version number. Files with other types use the FN2 for the type and do not have a version number. The initial list is

```
        ("LISP" "TEXT" nil :unspecific)
```
Some users may need to add to this list.

:fn1 *Operation on* its-pathname
:fn2 *Operation on* its-pathname
These two operations return a string that is the FN1 or FN2 host-dependent component of the pathname.

:type-and-version *Operation on* pathname
:new-type-and-version *new-type new-version* *Operation on* pathname
These two operations provide a way of pretending that ITS pathnames can have both a type and a version. They use the first three characters of the FN2 to store a type and the last three to store a version number.

On an ITS-pathname, :type-and-version returns the type and version thus extracted (not the same as the type and version of the pathname). :new-type-and-version returns a new pathname constructed from the specified new type and new version.

On any other type of pathname, these operations simply return or set both the type component and the version component.

24.7.2 TOPS-20 (Twenex), Tenex, and VMS.

A pathname on TOPS-20 (better known as Twenex) looks like
host: *device*: <*directory*> *name*. *type*. *version*
The primary device is PS:.

TOPS-20 pathnames are mapped to upper case. Special characters (including lower-case letters) are quoted with the circle-cross (⊗) character, which has the same character code in the Lisp Machine character set as Control-V, the standard Twenex quoting character, in the ASCII character set.

If you specify a period after the name, but nothing after that, then the type is :unspecific, which translates into an empty extension on the TOPS-20 system. If you omit the period, you have allowed the type to be defaulted.

TOPS-20 pathnames allow the special version symbols :oldest and :newest. In the string form of a pathname, these are expressed as '.-2', and as an omitted version.

The directory component of a TOPS-20 pathname may be structured. The directory <FOO.BAR> is represented as the list ("FOO" "BAR").

The characters * and % are wildcards that match any sequence of characters and any single character (within one pathname component), respectively. To specify a filename that actually contains a * or % character, quote the character with ⊗. When a component is specified with just a single *, the symbol :wild appears in the pathname object.

The TOPS-20 init file naming convention is "<*user*>*program*.INIT".

When there is an attempt to display a TOPS-20 file name in the who-line and there isn't enough room to show the entire name, the name is truncated and followed by a center-dot character to indicate that there is more to the name than can be displayed.

Tenex pathnames are almost the same as TOPS-20 pathnames, except that the version is preceeded by a semi-colon instead of a period, the default device is DSK instead of PS, and the quoting requirements are slightly different.

VMS pathnames are basically like TOPS-20 pathnames, with a few complexities. The primary device is USRD$.

First of all, only alphanumeric characters are allowed in filenames (though $ and underscore can appear in device names).

Secondly, a version number is preceded by ';' rather than by '.'.

Thirdly, file types (called "extensions" in VMS terminology) are limited to three characters. Each of the system's canonical types has a special mapping for VMS pathnames, which is three characters long:

```
:lisp → LSP      :text → TXT      :qfasl → QFS     :midas → MID
:press → PRS     :widths → WID    :patch-directory → PDR
:qwabl → QWB     :babyl → BAB     :mail → MAI      :xmail → XML
:init → INI      :unfasl → UNF    :output → OUT
```

24.7.3 Unix and Multics Pathnames

A Unix pathname is a sequence of directory or file names separated by slashes. The last name is the filename; preceding ones are directory names (but directories are files anyway). There are no devices or versions. Alphabetic case is significant in Unix pathnames, no case conversion is normally done, and lower case is the default. Therefore, components of solid upper or lower case are inverted in case when going between interchange form and raw form. (What the user types in a pathname string is the raw form.)

Unix allows you to specify a pathname relative to your default directory by using just a filename, or starting with the first subdirectory name; you can specify it starting from the root directory by starting with a slash. In addition, you can start with '..' as a directory name one or more times, to refer upward in the hierarchy from the default directory.

Unix pathnames on the Lisp Machine provide all these features too, but the canonicalization to a simple descending list of directory names starting from the root is done on the Lisp Machine itself when you merge the specified pathname with the defaults.

If a pathname string starts with a slash, the pathname object that results from parsing it is called "absolute". Otherwise the pathname object is called "relative".

In an absolute pathname object, the directory component is either a symbol (nil, :unspecific or :root), a string, or a list of strings. A single string is used when there is only one level of directory in the pathname.

A relative pathname has a directory that is a list of the symbol :relative followed by some strings. When the pathname is merged with defaults, the strings in the list are appended to the strings in the default directory. The result of merging is always an absolute pathname.

In a relative pathname's string form, the string ".." can be used as a directory name. It is translated to the symbol :up when the string is parsed. That symbol is processed when the relative pathname is merged with the defaults.

Restrictions on the length of Unix pathnames require abbreviations for the standard Zetalisp pathname types, just as for VMS. On Unix the preferred mappings of all canonical types are one or two characters long. We give here the mappings in raw form; they are actually specified in interchange form.

`:lisp → l`	`:text → tx`	`:qfasl → qf`	`:midas → md`
`:press → pr`	`:widths → wd`	`:patch-directory → pd`	
`:qwabl → qw`	`:babyl → bb`	`:mail → ma`	`:xmail → xm`
`:init → in`	`:unfasl → uf`	`:output → ot`	

The Multics file system is much like the Unix one; there are absolute and relative pathnames, absolute ones start with a directory delimiter, and there are no devices or versions. Alphabetic case is significant.

There are differences in details. Directory names are terminated, and absolute pathnames begun, with the character '>'. The containing directory is referred to by the character '<', which is complete in itself. It does not require a delimiter. Thus, <<FOO>BAR refers to subdirectory FOO, file BAR in the superdirectory of the superdirectory of the default directory.

The limits on filename sizes are very large, so the system canonical types all use their standard mappings. Since the mappings are specified as upper case, and then interpreted as being in interchange form, the actual file names on Multics contain lower case.

24.7.4 Lisp Machine File Systems

There are two file systems that run in the MIT Lisp Machine system. They have different pathname syntax. Both can be accessed either remotely like any other file server, or locally.

The Local-File system uses host name LM for the machine you are on. A Local-File system on another machine can be accessed using the name of that machine as a host name, provided that machine is known as a file server.

The remainder of the pathname for the Local-File system looks like "*directory*; *name.type # version*". There is no restriction on the length of names; letters are converted to upper case. Subdirectories are allowed and are specified by putting periods between the directory components, as in RMS.SUBDIR;.

The TOPS-20 pathname syntax is also accepted. In addition, if the flag **fs:*lmfs-use-twenex-syntax*** is non-nil, Local-File pathnames print out using TOPS-20 syntax. Note that since the printed representation of a pathname is cached, changing this flag's value does not change the printing of pathnames with existing representations.

The Local-File system on the filecomputer at MIT has the host name FS.

The LMFILE system is primarily for use as a file server, unless you have 512k of memory. At MIT it runs on the filecomputer and is accessed remotely with host name FC.

The remainder of an LMFILE pathname looks like "*directory*; *name type # version*". However, the directory and name can be composed of any number of subnames, separated by backslashes. This is how subdirectories are specified. FOO;BAR\X refers to the same file as FOO\BAR;X, but the two ways of specifying the file have different consequences in defaulting, getting directory listings, etc.

Case is significant in LMFILE pathnames; however, when you open a file, the LMFILE system ignores the case when it matches your pathname against the existing files. As a result, the case you use matters when you create or rename a file, and appears in directory listings, but it is ignored when you refer to an existing file, and you cannot have two files whose names differ only in case. When components are accessed in interchange form, they are always converted to upper case.

24.7.5 Logical Pathnames

There is another kind of pathname that doesn't correspond to any particular file server. It is called a *logical* pathname, and its host is called a logical host. Every logical pathname can be translated into a corresponding *physical* pathname because each logical host records a corresponding actual ("physical") host and rules for translating the other components of the pathname.

The reason for having logical pathnames is to make it easy to keep bodies of software on more than one file system. An important example is the body of software that constitutes the Lisp Machine system. Every site has a copy of all of the sources of the programs that are loaded into the initial Lisp environment. Some sites may store the sources on an ITS file system, while others may store them on a TOPS-20. However, system software (including **make-system**) wishes to be able to find a particular file independent of the name of the host a particular site stores it on, or even the kind of host it is. This is done by means of the logical host SYS; all pathnames for system files are actually logical pathnames with host SYS. At each site, SYS is defined as a logical host, but translations are different at each site. For example, at MIT the source files are stored on the TOPS-20 system named OZ, so MIT's site file says that SYS should translate to the host OZ.

Each logical host, such as SYS, has a list of translations, each of which says how to map certain pathnames for that host into pathnames for the corresponding physical host. To translate a logical pathname, the system tests each of the logical host's translations, in sequence, to see if it is applicable. (If none is applicable, an error is signaled.) A translation consists of a pair of pathnames or namestrings, typically containing wildcards. Unspecified components in them default to :wild. The *from*-pathname of the translation is used to match against the pathname to be translated; if it matches, the corresponding *to*-pathname is used to construct the translation, filling in its wild fields from the pathname being translated as in the :translate-wild-pathname operation (page 565).

Most commonly the translations contain pathnames that have only directories specified, everything else wild. Then the other components are unchanged by translation.

If the files accessed through the logical host are moved, the translations can be changed so that the same logical pathnames refer to the same files on their new physical host via physical pathnames changed to fit the restrictions and the conventions of the new physical host.

Each translation is specified as a list of two strings. The strings are parsed into pathnames and any unspecified components are defaulted to :wild. The first string of the pair is the source pattern; it is parsed with logical pathname syntax. The second string is the target pattern, and it is parsed with the pathname syntax for the specified physical host.

For example, suppose that logical host FOO maps to physical host BAR, a Tops-20, and has the following list of translations:

```
(("BACK;" "PS:<FOO.BACK>")
 ("FRONT;* QFASL" "SS:<FOO.QFASL>*.QFASL")
 ("FRONT;" "PS:<FOO.FRONT>"))
```

Then all pathnames with host FOO and directory BACK translate to host BAR, device PS and directory <FOO.BACK> with name, type and version unchanged. All pathnames with host FOO, directory FRONT and type QFASL translate to host BAR, device SS, directory <FOO.QFASL> and type QFASL, with name and version unchanged. All other pathnames with host FOO and directory FRONT map to host BAR, device PS and directory <FOO.FRONT>, with name, type and version unchanged. Note that the first translation whose pattern matches a given pathname is the one that is used.

Another site might define FOO's to map to a Unix host QUUX, with the following translation list:

```
(("BACK;" "//nd//foo//back//")
 ("FRONT;" "//nd//foo//front//"))
```

This site apparently does not see a need to store the QFASL files in a separate directory. Note that the slashes are duplicated to quote them for Lisp; the actual namestrings contain single slashes as is usual with Unix.

If the last translation's source pattern is entirely wild, it applies to any pathname not so far handled. Example:

```
(("BACK;" "//nd//foo//back//")
 ("" "//nd//fool//*//"))
```

Physical pathnames can also be *back-translated* into the corresponding logical pathname. This is the inverse transformation of ordinary translation. It is necessary to specify which logical host to back translate for, as it may be that the same physical pathname could be the translation of different logical pathnames on different hosts. Use the :back-translated-pathname operation, below.

fs:add-logical-pathname-host *logical-host physical-host translations*
fs:set-logical-pathname-host *logical-host* &key *physical-host translations*

Both create a new logical host named *logical-host*. Its corresponding physical host (that is, the host to which it should forward most operations) is *physical-host*. *logical-host* and *physical-host* should both be strings. *translations* should be a list of translation specifications, as described above. The two functions differ only in that one accepts positional arguments and the other accepts keyword arguments. Example:

```
(add-logical-pathname-host "MUSIC" "MUSIC-10-A"
          '(("MELODY;" "SS:<MELODY>")
            ("DOC;" "PS:<MUSIC-DOCUMENTATION>")))
```

This creates a new logical host called MUSIC. An attempt to open the file MUSIC:DOC;MANUAL TEXT 2 will be re-directed to the file MUSIC-10-A:PS:<MUSIC-DOCUMENTATION>MANUAL.TEXT.2 (assuming that the host MUSIC-10-A is a TOPS-20 system).

fs:make-logical-pathname-host *name*

Requests that the definition of logical host *name* be loaded from a standard place in the file system: namely, the file SYS: SITE; *name* TRANSLATIONS. This file is loaded immediately with load, in the fs package. It should contain code to create the logical host; normally, a call to fs:set-logical-pathname-host or fs:add-logical-pathname-host, above.

The same file is automatically reloaded, if it has been changed, at appropriate times: by load-patches, and whenever site information is updated.

:translated-pathname *Operation on* fs:logical-pathname

Converts a logical pathname to a physical pathname. It returns the translated pathname of this instance, a pathname whose host component is the physical host that corresponds to this instance's logical host.

If this operation is applied to a physical pathname, it simply returns that pathname unchanged.

:back-translated-pathname *pathname* *Operation on* fs:logical-pathname

Converts a physical pathname to a logical pathname. *pathname* should be a pathname whose host is the physical host corresponding to this instance's logical host. This returns a pathname whose host is the logical host and whose translation is *pathname*. If *pathname* is not the translation of any logical pathname on this instance's host, nil is returned.

Here is an example of how this would be used in connection with truenames. Given a stream that was obtained by opening a logical pathname,

```
(send stream :pathname)
```

returns the logical pathname that was opened.

```
(send stream :truename)
```

returns the true name of the file that is open, which of course is a pathname on the physical host. To get this in the form of a logical pathname, one would do

```
(send (send stream :pathname)
      :back-translated-pathname
      (send stream :truename))
```

If this operation is applied to a physical pathname, it simply returns its argument. Thus the above example works no matter what kind of pathname was opened to create the stream.

fs:unknown-logical-pathname-translation (fs:pathname-error error) *Condition*

This is signaled when a logical pathname has no translation. The condition instance supports the :logical-pathname operation, which returns the pathname that was untranslatable.

The proceed type :define-directory is supported. It expects a single argument, a pathname or a string to be parsed into one. This defines the target pattern for a translation whose source pattern is the directory from the untranslatable pathname (and all else wild). Such a translation is added to the logical host, making it possible to translate the pathname.

A logical pathname looks like "*host*: *directory*; *name type version*". There is no way to specify a device; parsing a logical pathname always returns a pathname whose device component is :unspecific. This is because devices don't have any meaning in logical pathnames.

The equivalence-sign character (≡) can be used for quoting special characters such as spaces and semicolons. The double-arrow character ('↔') can be used as a place-holder for components that are nil, and the up-horseshoe ('∪') indicates :unspecific (generic pathnames typically have :unspecific as the type and the version). All letters are mapped to upper case unless quoted. The :newest, :oldest, and :wild values for versions are written as '>', '<', and '*' respectively.

There isn't any init file naming convention for logical hosts; you can't log into them. The :string-for-host, :string-for-wholine, :string-for-dired, and :string-for-editor messages are all passed on to the translated pathname, but the :string-for-printing is handled by the **fs:logical-pathname** flavor itself and shows the logical name.

24.7.6 Editor Buffer Pathnames

The hosts ED, ED-BUFFER and ED-FILE are used in pathnames which refer to buffers in the editor. If you open such a pathname, you get a stream that reads or writes the contents of an editor buffer. The three host names differ only in the syntax of the pathname, and in how it is interpreted.

The host ED is followed by an abbreviation that should complete to the name of an existing editor buffer. For example, the pathname ED:FOO could refer to the buffer FOO.LISP PS:<ME> OZ:.

The host ED-BUFFER is followed by an exact buffer name. If there is no buffer with that name, one is created. This is most useful for creating a buffer.

The host ED-FILE is followed by an arbitrary pathname, including a host name. An ED-FILE pathname refers to a buffer visiting that file. If necessary, the file is read into the editor. For example, ED-FILE: OZ: PS:<ME>FOO.LISP would refer to the same buffer as ED: FOO. The current default defaults are used in processing the pathname that follows ED-FILE, when the pathname is parsed.

24.8 Hosts

Each host known to the Lisp Machine is represented by a flavor instance known as a host object. The host object records such things as the name(s) of the host, its operating system type, and its network address(es). Host objects print like #⊂FS:TOPS20-CHAOS-HOST "MIT-OZ"⊃, so they can be read back in.

Not all hosts support file access. Those that do support it appear on the list **fs:*pathname-host-list*** and can be the host component of pathnames. A host object is also used as an argument when you make a Chaosnet connection for any purpose.

The hosts that you can use for making network connections appear in the value of **si:host-alist**. Most of the hosts you can use for pathnames are among these; but some, such as logical hosts, are not.

24.8.1 Parsing Hostnames

si:parse-host *namestring* &optional *no-error-p* (*unknown-ok* **t**)

Returns a host object that recognizes the specified name. If the name is not recognized, it is an error, unless *no-error-p* is non-**nil**; in that case, **nil** is returned.

If *unknown-ok* is non-nil (the default), a host table server on the local network is contacted, to see if perhaps it can find the name there. If it can't, an error is signalled or **nil** is returned, according to *no-error-p*. The host instance created in this manner contains all the kinds of information that a host defined from the host table file has.

If a string of the form CHAOS|*nnn* is used, a host object is created and given *nnn* (interpreted as octal) as its Chaosnet address. This can be done regardless of the *unknown-ok* argument.

The first argument is allowed to be a host object instead of a string. In this case, that argument is simply returned.

sys:unknown-host-name *Condition*

(sys:local-network-error sys:network-error error)

This condition is signaled by **si:parse-host** when the host is not recognized, if that is an error.

The :name operation on the condition instance returns the string given to **si:parse-host**.

si:get-host-from-address *address network*

Returns a host object given an address and the name of the network which that address is for. Usually the symbol :chaos is used as the network name.

nil is returned if there is no known host with that address.

fs:get-pathname-host *name* &optional *no-error-p*

Returns a host object that can be used in pathnames. If the name is not recognized, it is an error, unless *no-error-p* is non-nil; in that case, nil is returned.

The first argument is allowed to be a host object instead of a string. In this case, that argument is simply returned.

si:parse-host and fs:get-pathname-host differ in the set of hosts searched.

fs:unknown-pathname-host (fs:pathname-error error) *Condition*

This condition is signaled by fs:get-pathname-host when the host is not recognized, if that is an error.

The :name operation on the condition instance returns the string given to fs:get-pathname-host.

fs:*pathname-host-list* *Variable*

This is a list of all the host objects that support file access.

si:host-alist *Variable*

This variable is a list of one element for each known network host. The element looks like this:

(*full-name host-object* (*nickname nickname2* ... *full-name*)
system-type machine-type site
network list-of-addresses network2 list-of-addresses2 ...)

The *full-name* is the host's official name. The :name operation on the host object returns this.

The *host-object* is a flavor instance that represents this host. It may be nil if none has been created yet; si:parse-host creates them when they are referred to.

The *nicknames* are alternate names that si:parse-host should recognize for this host, but which are not its official name.

The *system-type* is a symbol that tells what software the host runs. This is used to decide what flavor of host object to construct. Symbols now used include :lispm, :its, :tops-20, :tenex, :vms, :unix, :multics, :minits, :waits, :chaos-gateway, :dos, :rsx, :magicsix, :msdos, and others. Not all of these are specifically understood in any way by the Lisp Machine. If none of these applies to a host you wish to add, use a new symbol.

The *machine-type* is a symbol that describes the hardware of the host. Symbols in use include :lispm, :pdp10, :pdp11, :vax, :nu, :pe3230, and :ibmpc. (nil) has also been observed to appear here. Note that these machine types attempt to have wide meanings, lumping together various brands, models, etc.

The *site* does not describe anything about the host. Instead it serves to say what the Lisp Machine's site name was when the host was defined. This is so that, when a Lisp Machine system is moved to a different institution that has a disjoint set of hosts, all the old site's hosts can be deleted from the host alist by site reinitialization.

The *networks* and lists of addresses describe how to reach the host. Usually there is only one network and only one address in the list. The generality is so that hosts with multiple addresses on multiple networks can be recorded. Networks include :chaos and :arpa. The address is meaningful only to code for a specific network.

24.8.2 Host Object Operations

:name *Operation on host objects*

Returns the full, official name of the host.

:name-as-file-computer *Operation on host objects*

Returns the name to print in pathnames on this host (assuming it supports files). This is likely to be a short nickname of the host.

:short-name *Operation on host objects*

Returns the shortest known nickname for this host.

:pathname-host-namep *string* *Operation on host objects*

Returns t if *string* is recognized as a name for this host for purposes of pathname parsing. The local host will recognise LM as a pathname host name.

:system-type *Operation on host objects*

Returns the operating system type symbol for this host. See page 810.

:network-type *Operation on host objects*

Returns the symbol for one network that this host is connected to, or nil if it is not connected to any. :chaos is preferred if it is one of the possible values.

:network-typep *network* *Operation on host objects*

Returns t if the host is connected to the specified network.

:network-addresses *Operation on host objects*

Returns an alternating list of network names and lists of addresses, such as

```
(:chaos (3104) :arpa (106357002))
```

You can therefore find out all networks a host is known to be on, and its addresses on any network.

:sample-pathname *Operation on host objects*

Returns a pathname for this host, whose device, directory, name, type and version components are all nil. Sample pathnames are often useful because many file-system-dependent pathname operations depend only on the pathname's host.

:open-streams *Operation on host objects*

Returns a list of all the open file streams for files on this host.

:close-all-files *Operation on host objects*

Closes all file streams open for files on this host.

:generic-base-type *type-component* *Operation on host objects*

Returns the type component for a generic pathname assuming it is being made from a pathname whose type component is the one specified.

25. Accessing Files

The Lisp Machine can access files on a variety of remote file servers, which are typically (but not necessarily) accessed through the Chaosnet, as well as accessing files on the Lisp Machine itself, if the machine has its own file system. You are not allowed to refer to files without first logging in, and you may also need to specify a username and password for the host on which the file is stored; see page 801.

The way to read or write a file's contents is to *open* the file to get an input or output stream, use the standard stream I/O functions or operations described in chapters 22 and 23, and then close the stream. The first section of this chapter tells how to open and close the stream. The rest of the chapter describes things specific to files such as deleting and renaming, finding out the true name of the file that has been opened, and listing a directory.

Files are named with *pathnames*. There is much to know about pathnames aside from accessing files with them; all this is described in the previous chapter.

Many functions in this chapter take an argument called *file* which is intended to specify a file to be operated on. This argument may be given as a pathname (which is defaulted), a namestring (which is parsed into a pathname and then defaulted), or a stream open to a file (the same file is used).

25.1 Opening and Closing File Streams

with-open-file (*stream file options...*) *body...* *Macro*

Evaluates the *body* forms with the variable *stream* bound to a stream that reads or writes the file named by the value of *file*. The *options* forms evaluate to the file-opening options to be used; see page 582.

When control leaves the body, either normally or abnormally (via throw), the file is closed. If a new output file is being written and control leaves abnormally, the file is aborted and it is as if it were never written. Because it always closes the file, even when an error exit is taken, with-open-file is preferred over open. Opening a large number of files and forgetting to close them tends to break some remote file servers, ITS's for example.

If an error occurs in opening the file, the result depends on the values of the *error* option, the *if-exists* option, and the *if-does-not-exist* option. An error may be signaled (and possibly corrected with a new pathname), or *stream* may be bound to a condition object or even nil.

with-open-file-case (*stream file options...*) *clauses...* *Macro*

This opens and closes the file like with-open-file, but what happens afterward is determined by *clauses* that are like the clauses of a condition-case (page 702). Each clause begins with a condition name or a list of condition names and is executed if open signals a condition that possesses any of those names. A clause beginning with the symbol :no-error is executed if the file is opened successfully. This would be where the reading

or writing of the file would be done.
Example:

```
      (with-open-file-case (stream (send generic-pathname
                                        :source-pathname))
        (sys:remote-network-error (format t "~&Host down."))
        (fs:file-not-found (format t "~&(New file)"))
        (:no-error (setq list (read stream))))
```

file-retry-new-pathname *(pathname-var condition-names...) body...* *Macro*
file-retry-new-pathname-if *Macro*
cond-form (pathname-var condition-names...) body...

file-retry-new-pathname executes *body*. If *body* does not signal any of the conditions in *condition-names*, *body*'s values are simply returned. If any of *condition-names* is signaled, file-retry-new-pathname reads a new pathname, setq's *pathname-var* to it, and executes *body* again.

The user can type End instead of a pathname if he wishes to let the condition be handled by the debugger.

file-retry-new-pathname-if is similar, but the conditions are handled only if *cond-form*'s value is non-nil.

For an example, see the example of the following macro.

with-open-file-retry *Macro*
(stream (pathname-var condition-names...) options...) body...

Like with-open-file inside of a file-retry-new-pathname. If an error occurs while opening the file and it has one of the specified *condition-names*, a new pathname is read, the variable *pathname-var* is setq'd to it, and another attempt is made to open a file with the newly specified name. Example:

```
      (with-open-file-retry (instream (infile fs:file-not-found))
        ...)
```

infile should be a variable whose value is a pathname or namestring. The example is equivalent to

```
      (file-retry-new-pathname (infile fs:file-not-found)
        (with-open-file (instream infile)
          ...))
```

with-open-file-search *Macro*

Opens a file, trying various pathnames until one of them succeeds. The pathnames tried differ only in their type components. For example, load uses this macro to search for either a compiled file or a source file. The calling sequence looks like

```
      (with-open-file-search
          (streamvar (operation defaults auto-retry)
                     types-and-pathname options...)
        body...)
```

with-open-file-search tries opening various files until one succeeds; then binds *streamvar* to the stream and executes *body*, closing the stream on exit. The values of *body* are returned.

types-and-pathname specifies which files to open. It should be a form which evaluates to two values, the first being a list of types to try and the second being a pathname called the base pathname. Each pathname to try is made by merging the base pathname with the defaults *defaults* and one of the types. The types may be strings or canonical type keywords (see section 24.2.3, page 551).

options are forms whose values should be alternating to keywords and values, which are passed to open each time.

If all the names to be tried fail, a fs:multiple-file-not-found error is signaled. *operation* is provided just so that the :operation operation on the condition object can return it. Usually the value given for *operation* should be the user-level function for which the with-open-file-search is being done.

If *auto-retry* is non-nil, an error causes the user to be prompted for a new base pathname. The entire set of types specified is tried anew with the new pathname.

open *file* &rest *options*

Returns a stream that is connected to the specified file. Unlike Maclisp, the open function creates streams only for *files*; streams of other kinds are created by other functions. The *file* and *options* arguments are the same as in with-open-file; see above.

When the caller is finished with the stream, it should close the file by using the :close operation or the close function. The with-open-file special form does this automatically and so is usually preferred. open should only be used when the control structure of the program necessitates opening and closing of a file in some way more complex than the simple way provided by with-open-file. Any program that uses open should set up unwind-protect handlers (see page 82) to close its files in the event of an abnormal exit.

close *stream* &optional *option*

The close function simply sends the :close message to *stream*. If *option* is :abort for a file output stream, the file is discarded.

cli:close *stream* &key *abort*

The Common Lisp version of close is the same as close except for its calling convention. If *abort* is non-nil for a file output stream, the file is discarded.

fs:close-all-files

Closes all open files. This is useful when a program has run wild opening files and not closing them. It closes all the files in :abort mode (see page 464), which means that files open for output are deleted. Using this function is dangerous, because you may close files out from under various programs like Zmacs and ZMail; only use it if you have to and if you feel that you know what you're doing.

The *options* used when opening a file are normally alternating keywords and values, like any other function that takes keyword arguments. In addition, for compatibility with the Maclisp **open** function, if only a single option is specified it is either a keyword or a list of keywords (not alternating with values).

The file-opening options control things like whether the stream is for input from a existing file or output to a new file, whether the file is text or binary, etc.

The following keyword arguments are standardly recognized; additional keywords can be implemented by particular file system hosts.

direction
: Controls which direction of I/O can be done on the resulting stream. The possible values are **:input** (the default), **:output**, **nil**, **:probe**, **:probe-directory** and **:probe-link**. The first two should be self-explanatory. **nil** or **:probe** means that this is a "probe" opening; no data are to be transferred, the file is being opened only to verify its existence or access its properties. The stream created in this case does not permit any I/O. **nil** and **:probe** differ in causing different defaults for the argument *if-does-not-exist*. If that argument is specified explicitly, **nil** and **:probe** are equivalent.

  :probe-directory is used to see whether a directory exists. If the directory specified for the file to be opened is found, then the **open** completes (returning a non-I/O stream) as if the specified file existed whether it really exists or not.

  :probe-link is used to find out the truename of a link. If the file specified exists as a link, then the **open** completes returning a non-I/O stream which describes the link itself rather than the file linked to. If the file exists and is not a link, the **open** also completes for it as with any probe.

  Common Lisp defines the value **:io** for this argument, requesting a stream that can do input and output, but no file system supported by the Lisp Machine has this capability.

characters
: The possible values are **t** (the default), **nil**, which means that the file is a binary file, and **:default**, which means that the file system should decide whether the file contains characters or binary data and open it in the appropriate mode.

byte-size
: The possible values are **nil** (the default), a number, which is the number of bits per byte, and **:default**, which means that the file system should choose the byte size based on attributes of the file. If the file is being opened as characters, **nil** selects the appropriate system-dependent byte size for text files; it is usually not useful to use a different byte size. If the file is being opened as binary, **nil** selects the default byte size of 16 bits.

element-type
: This is the Common Lisp way to specify what kind of objects the stream wants to read or write. This combines the effect of the *characters* and *byte-size* arguments. The value is a type specifier; it must be one of the following:

  string-char Read or write characters as usual. The default.

  character Read or write characters, dealing with characters that are more than 8 bits. You can succeed in writing out any sequence of character objects and reading it back, but the file does not look anything like a text file.

(unsigned-byte *n*)
: Read or write *n*-bit bytes. Like *characters* = nil, *byte-size* = *n*.

unsigned-byte
: Similar, but uses the byte size that the file was originally written with. This is the same as *characters* = nil, *byte-size* = :default.

(signed-byte *n*)
: Read or write *n*-bit bytes, sign-extending on input. Each byte read from the file is sign-extended so that its most significant bit serves as a sign bit.

signed-byte
: Similar, but uses the byte size that the file was originally written with.

(mod *n*)
: Like unsigned-byte for a big enough byte size to hold all numbers less than *n*. bit is also accepted, and means (mod 2).

:default
: Is allowed, even though it is not a type specifier. It is the same as using :default as the value of *characters*.

if-exists
: For output opens, *if-exists* specifies what to do if a file with the specified name already exists. There are several values you can use:

:new-version
: Create a new version. This makes sense only when the pathname has :newest as its version, and it is the default in that case.

:supersede
: Make a new file which, when closed, replaces the old one.

:overwrite
: Write over the data of the existing file, starting at the beginning, and set the file's length to the length of the newly written data.

:truncate
: Like :overwrite except that it discards the old contents of the file immediately, making it empty except for what is written into it this time.

:append
: Add new data onto the existing file at the end.

:rename
: Rename the existing file and then create a new one.

:rename-and-delete
: Rename the existing file, create a new one, and delete the old file when the new one is closed.

:error
: Signal an error (fs:file-already-exists). This is the default when the pathname's version is not :newest. The further handling of the error is controlled by the *error* argument.

nil
: Return nil from open in this case. The *error* argument is irrelevant in this case.

if-does-not-exist
: Specifies what to do when the file requested does not exist. There are three allowed values:

:create	Create a file. This is the default for output opens, except when *if-exists* is :append, :overwrite or :truncate. This silly exception is part of the Common Lisp specifications.
:error	Signal an error. This is the default for input opens, and also for output opens when *if-exists* is :append, :overwrite or :truncate. The further handling of the error is controlled by the *error* argument.
nil	Return nil from open. This is the default for :probe opens. The *error* argument is irrelevant in this case.

error Specifies what to do if an error is signaled for any reason. (Note that the values of the *if-exists* and *if-does-not-exist* arguments control whether an error is signaled in certain circumstances.) The possible values are t (the default), :reprompt and nil. t means that nothing special is done, so the error invokes the debugger if the caller does not handle it. nil means that the condition object should be returned as the value of open. :reprompt means that a new file name should be read and opened.

Any caller which need not know reliably which file was ultimately opened might as well specify :reprompt for this argument. Callers which need to know if a different file is substituted should never specify :reprompt; they may use with-open-file-retry or file-retry-new-pathname (see page 581) if they wish to permit an alternative file name to be substituted.

:submit If specified as t when opening a file for output, the file is submitted as a batch job if it is closed normally. The default is nil. You must specify :direction :output as well.

deleted The default is nil. If t is specified, and the file system has the concept of deleted but not expunged files, it is possible to open a deleted file. Otherwise deleted files are invisible.

temporary If t is specified, the file is marked as temporary, if the file system has that concept. The default is nil.

preserve-dates If t is specified, the file's reference and modification dates are not updated. The default is nil.

flavor This controls the kind of file to be opened. The default is nil, a normal file. Other possible values are :directory and :link. Only certain file systems recognize this keyword.

link-to When creating a file with *flavor* :link, this argument must be specified; its value is a pathname or namestring that becomes the target of the link.

submit The value can be either nil (the default) or t. If the value is t, and the :direction is :output, the resulting file will be submitted as a batch job. Currently, this option is implemented only for Twenex and VMS.

estimated-size The value may be nil (the default), which means there is no estimated size, or a number of bytes. Some file systems use this to optimize disk allocation.

physical-volume The value may be nil (the default), or a string that is the name of a physical volume on which the file is to be stored. This is not meaningful for all file systems.

logical-volume The value may be nil (the default), or a string that is the name of a logical volume on which the file is to be stored. This is not meaningful for all file systems.

super-image The value may be nil (the default), or t, which disables the special treatment of rubout in ASCII files. Normally, rubout is an escape which causes the following character to be interpreted specially, allowing all characters from 0 through 376 (octal) to be stored. This applies to ASCII file servers only.

raw The value may be nil (the default), or t, which disables all character set translation in ASCII files. This applies to ASCII file servers only.

In the Maclisp compatibility mode, there is only one *option*, and it is either a symbol or a list of symbols. These symbols are recognized no matter what package they are in, since Maclisp does not have packages. The following symbols are recognized:

in, read Select opening for input (the default).

out, write, print
Select opening for output; a new file is to be created.

binary, fixnum Select binary mode; otherwise character mode is used. Note that fixnum mode uses 16-bit binary words and is not compatible with Maclisp fixnum mode, which uses 36-bit words. On the PDP-10, fixnum files are stored with two 16-bit words per PDP-10 word, left-justified and in PDP-10 byte order.

character, ascii
The opposite of fixnum. This is the default.

single, block Ignored for compatibility with the Maclisp open function.

byte-size Must be followed by a number in the options list, and must be used in combination with fixnum. The number is the number of bits per byte, which can be from 1 to 16. On a PDP-10 file server these bytes will be packed into words in the standard way defined by the ILDB instruction. The :tyi stream operation will (of course) return the bytes one at a time.

probe, error, noerror, raw, super-image, deleted, temporary
These are not available in Maclisp. The corresponding keywords in the normal form of file-opening options are preferred over these.

25.2 File Stream Operations

The following functions and operations may be used on file streams, in addition to the normal I/O operations which work on all streams. Note that several of these operations are useful with file streams that have been closed. Some operations use pathnames; refer to chapter 24, page 545 for an explanation of pathnames.

file-length *file-stream*

Returns the length of the file open on *file-stream*, in terms of the units in which I/O is being done on that stream. (A stream is needed, rather than just a pathname, in order to specify the units.)

file-position *file-stream* &optional *new-position*

With one argument, returns the current position in the file of *file-stream*, using the :read-pointer stream operation. It may return nil meaning that the position cannot be determined. In fact, it always returns nil for a stream open in character mode and not at the beginning of the file.

With two arguments, sets the position using the :set-pointer stream operation, if possible, and returns t if the setting was possible and nil if not. You can specify :start as the *new-position* to position to the beginning of the file, or :end to position to the end.

:pathname *Operation on file streams*

Returns the pathname that was opened to get this stream. This may not be identical to the argument to open, since missing components will have been filled in from defaults. The pathname may have been replaced wholesale if an error occurred in the attempt to open the original pathname.

:truename *Operation on file streams*

Returns the pathname of the file actually open on this stream. This can be different from what :pathname returns because of file links, logical devices, mapping of version :newest to a particular version number, etc. For an output stream the truename is not meaningful until after the stream has been closed, at least when the file server is an ITS.

:generic-pathname *Operation on file streams*

Returns the generic pathname of the pathname that was opened to get this stream. Normally this is the same as the result of sending the :generic-pathname message to the value of the :pathname operation on the stream; however, it does special things when the Lisp system is bootstrapping itself.

:qfaslp *Operation on file streams*

Returns t if the file has a magic flag at the front that says it is a QFASL file, nil if it is an ordinary file.

:length *Operation on file streams*

Returns the length of the file, in bytes or characters. For text files on ASCII file servers, this is the number of ASCII characters, not Lisp Machine characters. The numbers are different because of character-set translation; see section 25.8, page 607 for a full explanation. For an output stream the length is not meaningful until after the stream has been closed, at least when the file server is an ITS.

:creation-date *Operation on file streams*
Returns the creation date of the file, as a number that is a universal time. See the chapter on the time package (chapter 34, page 776).

:info *Operation on file streams*
Returns a cons of the file's truename and its creation date. This can be used to tell if the file has been modified between two open's. For an output stream the information is not guaranteed to be correct until after the stream has been closed.

:properties &optional (*error-p* t) *Operation on file streams*
This returns two values: a property list (like an element of the list returned by fs:directory-list), and a list of the settable properties. See the section on standard file properties (section 25.6, page 598) for a description of the ones that may possible found in the list.

:set-byte-size *new-byte-size* *Operation on file streams*
This is only allowed on binary file streams. The byte size can be changed to any number of bits from 1 to 16.

:delete &optional (*error-p* t) *Operation on file streams*
Deletes the file open on this stream. For the meaning of *error-p*, see the deletef function. The file doesn't really go away until the stream is closed.

:undelete &optional (*error-p* t) *Operation on file streams*
If you have used the :deleted option in open to open a deleted file, this operation undeletes the file.

:rename *new-name* &optional (*error-p* t) *Operation on file streams*
Renames the file open on this stream. For the meaning of *error-p*, see the renamef function.

File output streams implement the :finish and :force-output operations.

25.3 Manipulating Files

This section describes functions for doing things to files aside from reading or writing their contents.

truename *object*
Returns the truename of the file specified somehow by *object*. If *object* is a plausible stream, it is asked for the truename with the :truename operation. Otherwise, *object* is converted to a pathname and that pathname is opened to get its file's truename.

delete-file *file* &key (*error-p* t) *query?*
deletef *file* &optional (*error-p* t) *query?*

Both delete the specified file. The two functions differ in accepting keyword arguments versus positional arguments. *file* may contain wildcard characters, in which case multiple files are deleted.

If *query?* is non-nil, the user is queried about each file (whether there are wildcards or not). Only the files that the user confirms are actually deleted.

If *error-p* is t, then if an error occurs it is signaled as a Lisp error. If *error-p* is nil and an error occurs, the error message is returned as a condition object. Otherwise, the value is a list of elements, one for each file considered. The car of each element is the truename of the file, and the cadr is non-nil if the file was actually deleted (it is always t unless querying was done).

undelete-file *file* &key (*error-p* t) *query?*
undeletef *file* &optional (*error-p* t) *query?*

Both undelete the specified file. Wildcards are allowed, just as in deletef. The rest of the calling conventions are the same as well. The two functions differ in taking keyword arguments versus positional arguments.

Not all file systems support undeletion, and if it is not supported on the one you are using, it gets an error or returns a string according to *error-p*. To find out whether a particular file system supports this, send the :undeletable-p operation to a pathname. If it returns t, the file system of that pathname supports undeletion.

rename-file *file new-name* &key (*error-p* t) *query?*
renamef *file new-name* &optional (*error-p* t) *query?*

Both rename the specified file to *new-name* (a pathname or string). The two functions differ in taking keyword arguments versus positional arguments. *file* may contain wildcards, in which case multiple files are renamed. Each file's new name is produced by passing *new-name* to merge-pathname-defaults with the file's truename as the defaults. Therefore, *new-name* should be a string in this case.

If *query?* is non-nil, the user is queried about each file (whether there are wildcards or not). Only the files that the user confirms are actually renamed.

If *error-p* is t, then if an error occurs it is signaled as a Lisp error. If *error-p* is nil and an error occurs, the error message is returned as a condition object. Otherwise, the value is a list of elements, one for each file considered. The car of each element is the original truename of the file, the cadr is the name it was to be renamed to, and the caddr is non-nil if the file was renamed. The caddr is nil if the user was queried and said no.

copy-file *file new-name* &key (*error* t) (*copy-creation-date* t) (*copy-author* t) *report-stream* (*create-directories* :query) (*characters* :default) (*byte-size* :default)

Copies the file specified by *file* to the name *new-name*.

characters and *byte-size* specify what mode of I/O to use to transfer the data. *characters* can be

t	to specify character input and output.
nil	for binary input and output,
:ask	meaning ask the user which one
:maybe-ask	meaning ask if it is not possible to tell with certainty which method is best,
:default	meaning to guess as well as possible automatically.

If binary transfer is done, *byte-size* specifies the byte size to use; **:default** means to ask the file system for the byte size that the old file is stored in, just as it does in **open**.

copy-author and *copy-creation-date* say whether to set those properties of the new file to be the same as those of the old file. If a property is not copied, it is set to your login name or the current date and time.

report-stream, if non-nil, is a stream on which a message should be printed describing the file copied, where it is copied to, and which mode was used.

create-directories says what to do if the output filename specifies a directory that does not exist. It can be **t** meaning create the directory, **nil** meaning treat it as an error, or **:query** meaning ask the user which one to do. The default is **:query**.

error, if **nil**, means that if an error happens then this function should just return an error indication.

If the pathname to copy from contains wildcards, multiple files are copied. The new name for each file is obtained by merging *new-name* (parsed into a pathname) with that file's truename as a default. The mode of copy is determined for each file individually, and each copy is reported on the *report-stream* if there is one. If *error* is **nil**, an error in copying one file does not prevent the others from being copied.

There are four values. If wildcards were used, each value is a list with one element describing each file that matched; otherwise, each value describes the single file specified (though the value may be a list anyway). The values, for each file, are:

output-file	The defaulted pathname to be opened for output in copying this file.
truename	The truename of the file copied
outcome	The truename of the new file, If the file was successfully copied. A condition object, if there was an error and *error* was **nil**. **nil** if the user was asked whether to copy this file and said no.
mode	A Common Lisp type descriptor such as **string-char** or **(unsigned-byte 8)** saying how the file was copied.

probe-file *file*
probef *file*

Returns **nil** if there is no file named *file*; otherwise returns a pathname that is the true name of the file, which can be different from *file* because of file links, version numbers, etc. If *file* is a stream, this function cannot return **nil**.

Any problem in opening the file except for **fs:file-not-found** signals an error.

probef is the Maclisp name; **probe-file** is the Common Lisp name.

file-write-date *file*

Returns the creation date/time of *file*, as a universal time.

file-author *file*

Returns the name of the author of *file* (the user who wrote it), as a string.

viewf *file* &optional (*output-stream* ***standard-output***) *leader*

Copies the contents of the specified file, opened in character mode, onto **output-stream**. Normally this has the effect of printing the file on the terminal. *leader* is passed along to **stream-copy-until-eof** (see page 457).

fs:create-link *link-name link-to* &key (*error* **t**)

Creates a link named *link-name* which points to a file named *link-to*. An error happens if the host specified in *link-name* does not support links, or for any of the usual problems that can happen in creating a file.

25.3.1 Loading Files

To *load* a file is to read through the file, evaluating each form in it. Programs are typically stored in files; the expressions in the file are mostly special forms such as **defun** and **defvar** which define the functions and variables of the program.

Loading a compiled (or QFASL) file is similar, except that the file does not contain text but rather pre-digested expressions created by the compiler which can be loaded more quickly.

These functions are for loading single files. There is a system for keeping track of programs which consist of more than one file; for further information refer to chapter 28, page 660.

load *file* &key *verbose print* (*if-does-not-exist* **t**) *set-default-pathname package*

Loads the specified file into the Lisp environment. If *file* is a stream, **load** reads from it; otherwise *file* is defaulted from the default pathname defaults and the result specifies a file to be opened. If the file is a QFASL file, **fasload** is used; otherwise **readfile** is used. If *file* specifies a name but no type, **load** looks first for the canonical type **:qfasl** and then for the canonical type **:lisp**.

Normally the file is read into the package specified in its attribute list, but if *package* is supplied then the file is read in that package. If *package* is **nil** and *verbose* is **nil**, **load** prints a message saying what file is being loaded and what package is being used. *verbose* defaults to the value of ***load-verbose***.

If *if-does-not-exist* is nil, **load** just returns nil if no file with the specified name exists. Error conditions other than **fs:file-not-found** are not handled by this option.

If a file is loaded, **load** returns the file's truename.

If *print* is non-nil, the value of each expression evaluated from the file is printed on ***standard-output***.

pathname is defaulted from the default pathname defaults. If *set-default-pathname* is non-nil, the pathname defaults are set to the name of the file loaded. The default for *set-default-pathname* is **t**.

load used to be called with a different calling sequence:

```
    (load pathname pkg nonexistent-ok
          dont-set-default)
```

This calling sequence is detected and still works, but it is obsolete.

load-verbose *Variable*

Is the default value for the *verbose* argument to **load**.

readfile *file* &optional *pkg no-msg-p*

readfile is the version of **load** for text files. It reads and evaluates each expression in the file. As with **load**, *pkg* can specify what package to read the file into. Unless *no-msg-p* is t, a message is printed indicating what file is being read into what package.

fasload *file* &optional *pkg no-msg-p*

fasload is the version of **load** for QFASL files. It defines functions and performs other actions as directed by the specifications inserted in the file by the compiler. As with **load**, *pkg* can specify what package to read the file into. Unless *no-msg-p* is **t**, a message is printed indicating what file is being read into what package.

25.4 Pathname Operations That Access Files

Here are the operations that access files. Many accept an argument *error* or *error-p* which specifies whether to signal an error or to return a condition instance, if the file cannot be accessed. For these arguments, **nil** and non-**nil** are the only significant values. **:reprompt** has no special meaning as a value. That value when passed to one of the file accessing functions (**open**, **deletef**, etc.) has its special significance at a higher level.

:truename *Operation on* **pathname**

Returns a pathname object describing the exact name of the file specified by the pathname the object is sent to.

This may be different from the original pathname. For example, the original pathname may have **:newest** as the version, but the truename always has a number as the version if the file system supports versions.

:open *pathname* &rest *options* *Operation on* pathname

Opens a stream for the file named by the pathname. The argument *pathname* is what the :pathname operation on the resulting stream should return. When a logical pathname is opened, *pathname* is that logical pathname, but self is its translated pathname.

options is a list of alternating keywords and values, as would be passed to open. The old style of open keywords are not allowed; when they are used with open, open converts them to the new style before sending the :open message.

:delete &optional (*error-p* t) *Operation on* pathname
:undelete &optional (*error-p* t) *Operation on* pathname

Respectively delete or undelete the file specified by the pathname.

All file systems support :delete but not all support :undelete.

If *error-p* is nil, problems such as nonexistent files cause a string describing the problem to be returned. Otherwise, they signal an error.

:undeletable-p *Operation on* pathname

Returns t if this pathname is for a file system which allows deletion to be undone. Such pathnames support the :undelete and :expunge operations.

:rename *new-name* &optional (*error-p* t) *Operation on* pathname

Renames the file specified by the pathname. *new-name*, a string or pathname, specifies the name to rename to. If it is a string, it is parsed using self as the defaults.

If *error-p* is nil, problems such as nonexistent files cause a string describing the problem to be returned. Otherwise, they signal an error.

:complete-string *string options* *Operation on* pathname

Attempts to complete the filename *string*, returning the results. This operation is used by the function fs:complete-pathname (see page 602). The pathname the message is sent to is used for defaults. *options* is a list whose elements may include :deleted, :read (file is for input), :write (it's for output), :old (only existing files allowed), or :new-ok (new files are allowed too).

There are two values: a string, which is the completion as far as possible, and a flag, which can be :old, :new or nil. :old says that the returned string names an existing file, :new says that the returned string is no file but some completion was done, nil says that no completion was possible.

:change-properties *error-p* &rest *properties* *Operation on* pathname

Changes the properties of the file specified by the pathname. *properties* should be an alternating list of property names and values.

:directory-list *options* — *Operation on* pathname

Performs the work of (fs:directory-list *this-pathname options*...).

:properties — *Operation on* pathname

Returns a property list (in the form of a directory-list element) and a list of settable properties. See section 25.6, page 598 for more information on file properties.

:wildcard-map *function plistp dir-list-options* &rest *args* — *Operation on* pathname

Maps *function* over all the files specified by this pathname (which may contain wildcards). Each time *function* is called, its first argument is a pathname with no wildcards, or else a directory-list element (whose car is a pathname and whose cdr contains property names and values). The elements of *args* are given to *function* as additional arguments.

plistp says whether *function*'s first argument should be a directory-list element or just a pathname. t specifies a directory-list element. That provides more information, but it makes it necessary to do extra work if the specified pathname does *not* contain wildcards.

dir-list-options is passed to fs:directory-list. You can use this to get deleted files mentioned in the list, for example.

The remaining file-access operations are defined only on certain file systems.

:expunge &key (*error* t) — *Operation on* pathname

Expunges the directory specified by the host, device and directory components of the pathname.

The argument *error* says whether to signal an error if the directory does not exist. nil means just return a string instead.

:create-directory &key (*error* t) — *Operation on* pathname

Creates the directory specified in this pathname.

:remote-connect &key (*error* t) *access* — *Operation on* pathname

Performs the work of fs:remote-connect with the same arguments on this pathname's host.

25.5 File Attribute Lists

Any text file can contain an *attribute list* that specifies several attributes of the file. The above loading functions, the compiler, and the editor look at this property list. Attribute lists are especially useful in program source files, i.e. a file that is intended to be loaded (or compiled and then loaded). QFASL files also contain attribute lists, copied from their source files.

If the first non-blank line in a text file contains the three characters '-*-', some text, and '-*-' again, the text is recognized as the file's attribute list. Each attribute consists of the attribute name, a colon, and the attribute value. If there is more than one attribute they are separated by semicolons. An example of such an attribute list is:

```
	; -*- Mode:Lisp; Package:Cellophane; Base:10 -*-
```

This defines three attributes: mode, package, and base. The initial semicolon makes the line look like a comment rather than a Lisp expression. Another example is:

```
    .c Part of the Lisp Machine manual.  -*- Mode:Bolio -*-
```

An attribute name is made up of letters, numbers, and otherwise-undefined punctuation characters such as hyphens. An attribute value can be such a name, or a decimal number, or several such items separated by commas. Spaces may be used freely to separate tokens. Upper and lower-case letters are not distinguished. There is *no* quoting convention for special characters such as colons and semicolons.

If the attribute list text contains no colons, it is an old Emacs format, containing only the value of the **Mode** attribute.

The file attribute list format actually has nothing to do with Lisp; it is just a convention for placing some information into a file that is easy for a program to interpret. The Emacs editor on the PDP-10 knows how to interpret these attribute lists (primarily in order to look at the **Mode** attribute).

The Lisp Machine handles the attribute list stored in the file by parsing it into a Lisp data structure, a property list. Attribute names are interpreted as Lisp symbols and are interned on the keyword package. Numbers are interpreted as Lisp fixnums and are read in decimal. If a attribute value contains any commas, then the commas separate several expressions that are formed into a list.

When a file is compiled, its attribute list data structure is stored in the QFASL file. It can be loaded back from the QFASL file as well. The representation in the QFASL file resembles nothing described here, but when the attribute list is extracted from there, the same Lisp data structure described above is obtained.

When a file is edited, loaded, or compiled, its file attribute list is read in and the properties are stored on the property list of the generic pathname (see section 24.5, page 561) for that file, where they can be retrieved with the **:get** and **:plist** messages. This is done using the function **fs:read-attribute-list**, below. So the way you examine the properties of a file is usually to use messages to a pathname object that represents the generic pathname of a file. Note that there are other properties there, too.

Here the attribute names with standard meanings:

Mode The editor major mode to be used when editing this file. This is typically the name of the language in which the file is written. The most common values are **Lisp** and **Text**.

Package This attribute specifies the package in which symbols in the file should be interned. The attribute may be either the name of a package, or a list that specifies both the package name and how to create the package if it does not exist. If it is a list, it should look like (*name superpackage initial-size ...options...*). See chapter 27, page 636 for more information about packages.

Base The number base in which the file is written (remember, it is always parsed in decimal). This affects both ***read-base*** and ***print-base***, since it is confusing to have the input and output bases be different. The most common values are 8 and 10.

Readtable	The value specifies the syntax (that is, the choice of readtable) to use for reading Lisp objects from this file. The defined values are **t** or **traditional** for traditional Lisp Machine syntax, and **cl** or **common-lisp** for Common Lisp syntax. If you do not specify this option, the objects in the file are read using whatever readtable is current in the program that reads them.
Lowercase	If the attribute value is not **nil**, the file is written in lower-case letters and the editor does not translate to upper case. (The editor does not translate to upper case by default unless the user enables Electric Shift Lock mode.)
Fonts	The attribute value is a list of font names, separated by commas. The editor uses this for files that are to be displayed in a specific font, or contain multiple fonts. If this attribute is present, the file is actually stored in the file system with font-change indicators. A font-change indicator is an epsilon (ε) followed by a digit or *. ε*n* means to enter font *n*. The previous font is saved on a stack and ε* means to pop the stack, returning to the previous font. If the file includes an epsilon as part of its contents, it is stored as εε. When expressions are read from such files, font-change indicators are ignored, and εε is treated as a single ε.
Backspace	If the attribute value is not **nil**, **Overstrike** characters in the file should cause characters to overprint on each other. The default is to disallow overprinting and display **Overstrike** the way other special function keys are displayed. This default is to prevent the confusion that can be engendered by overstruck text.
Patch-File	If the attribute value is not **nil**, the file is a *patch file*. When it is loaded the system will not complain about function redefinitions. In a patch file, the **defvar** special-form turns into **defconst**; thus patch files always reinitialize variables. Patch files are usually created by special editor commands described in section 28.8, page 672.
Cold-Load	A non-**nil** value for this attribute identifies files that are part of the cold load, the core from which a new system version is built. Certain features that do not work in the cold load check this flag to give an error or a compiler warning if used in such files, so that the problem can be detected sooner.

You are free to define additional file attributes of your own. However, to avoid accidental name conflicts, you should choose names that are different from all the names above, and from any names likely to be defined by anybody else's programs.

The following functions are used to examine file attribute lists:

fs:file-attribute-list *pathname*

Returns the attribute list of the file specified by the pathname. This works on both text files and QFASL files.

fs:extract-attribute-list *stream*

Returns the attribute list read from the specified stream, which should be pointing to the beginning of a file. This works on both text streams and QFASL file binary streams. After the attribute list is read, the stream's pointer is set back to the beginning of the file using the :set-pointer file stream operation (see page 468).

fs:read-attribute-list *pathname stream*

pathname should be a pathname object (*not* a string or namelist, but an actual pathname); usually it is a generic pathname (see section 24.5, page 561). *stream* should be a stream that has been opened and is pointing to the beginning of the file whose file attribute list is to be parsed. The attribute list is read from the stream and then corresponding properties are placed on the specified *pathname*. The attribute list is also returned.

The fundamental way that programs in the Lisp Machine notice the presence of properties on a file's attribute list is by examining the property list in the generic pathname. However, there is another way that is more convenient for some applications. File attributes can cause special variables to be bound whenever Lisp expressions are being read from the file—when the file is being loaded, when it is being compiled, when it is being read from by the editor, and when its QFASL file is being loaded. This is how the Package and Base attributes work. You can also deal with attributes this way, by using the following function:

fs:file-attribute-bindings *pathname*

Returns values describing the special variables that should be bound before reading expressions from file *pathname*. It examines the property list of *pathname* and finds all those property names that have fs:file-attribute-bindings properties. Each such property name specifies a set of variables to bind and a set of values to which to bind them. This function returns two values, a list of all the variables and a list of all the corresponding values. Usually you use this function by calling it on a generic pathname that has had fs:read-attribute-list done on it, and then you use the two returned values as the first two arguments of a progv special form (see page 32). Inside the body of the progv the specified bindings will be in effect.

pathname may be anything acceptable as the first argument of get. Usually it is a generic pathname.

Of the standard attribute names, the following ones have fs:file-attribute-bindings, with the following effects. Package binds the variable package (see page 637) to the package. Base binds the variables *print-base* (see page 514) and *read-base* (see page 517) to the value. Readtable binds the variable readtable to a value computed from the specified attribute. Patch-file binds fs:this-is-a-patch-file to the value. Cold-load binds si:file-in-cold-load to the value. Fonts binds si:read-discard-font-changes to t.

Any properties whose names do not have fs:file-attribute-bindings properties are ignored completely.

You can also add your own attribute names that affect bindings. If an indicator symbol has an fs:file-attribute-bindings property, the value of that property is a function that is called when a file with a file attribute of that name is going to be read from. The function is given three arguments: the file pathname, the attribute name, and the attribute value. It must return two values: a list of variables to be bound and a list of values to bind them to. The function for the **Base** keyword could have been defined by:

```
(defun (:base file-attribute-bindings) (file ignore bse)
  (if (not (and (typep bse 'fixnum)
                (> bse 1)
                (< bse 37.)))
      (ferror 'fs:invalid-file-attrbute
              "File ~A has an illegal -*- Base:~D -*-"
              file bse))
  (values (list 'base 'ibase) (list bse bse)))
```

fs:extract-attribute-bindings *stream*

Returns two values: a list of variables, and a corresponding list of values to bind them to, giving the attribute bindings of the attribute list found on *stream*

fs:invalid-file-attribute (error) *Condition*

An attribute in the file attribute list had a bad value. This is detected within **fs:file-attribute-bindings.**

25.6 Accessing Directories

To understand the functions in this section, it is vital to have read the chapter on *pathnames*. The *filespec* argument in many of these functions may be a pathname or a namestring; its name, type and version default to **:wild.**

listf *filespec*

Prints on ***standard-output*** the names of the files that match *filespec*, and their sizes, creation dates, and other information that comes in the directory listing.

fs:directory-list *filespec* &rest *options*

Finds all the files that match *filespec* and returns a list with one element for each file. Each element is a list whose car is the pathname of the file and whose cdr is a list of the properties of the file; thus the element is a disembodied property list and **get** may be used to access the file's properties. The car of one element is **nil**; the properties in this element are properties of the file system as a whole rather than of a specific file.

filespec normally contains wildcards, and the data returned describe all existing files that match it. If it contains no wildcards, it specifies a single file and only that file is described in the data that are returned.

The *options* are keywords which modify the operation. The following options are currently defined:

:noerror	If a file-system error (such as no such directory) occurs during the operation, normally an error is signaled and the user is asked to supply a new pathname. However, if **:noerror** is specified then, in the event of an error, a condition object describing the error is returned as the result of **fs:directory-list**. This is identical to the **:noerror** option to **open.**

:deleted This is for file servers on which deletion is not permanent. It specifies that deleted (but not yet expunged) files are to be included in the directory listing.

:sorted This requests that the directory list be sorted by filenames before it is returned.

The properties that may appear in the list of property lists returned by fs:directory-list are host-dependent to some extent. The following properties are those that are defined for both ITS and TOPS-20 file servers. This set of properties is likely to be extended or changed in the future.

:length-in-bytes
The length of the file expressed in terms of the basic units in which it is written (characters in the case of a text file).

:byte-size The number of bits in one of those units.

:length-in-blocks
The length of the file in terms of the file system's unit of storage allocation.

:block-size The number of bits in one of those units.

:creation-date The date the file was created, as a universal time. See chapter 34, page 776.

:reference-date
The most recent date on which the file was used, as a universal time or nil, meaning the file was never referenced.

:modification-date
The most recent date on which the file's contents were changed, as a universal time.

:author The name of the person who created the file, as a string.

:reader The name of the person who last read the file, as a string.

:not-backed-up
t if the file exists only on disk, nil if it has been backed up on magnetic tape.

:directory t if this file is actually a directory.

:temporary t if this file is temporary.

:deleted t if this file is deleted. Deleted files are included in the directory list only if you specify the :deleted option.

:dont-delete t indicates that the file is not allowed to be deleted.

:dont-supersede
t indicates that the file may not be superseded; that is, a file with the same name and higher version may not be created.

:dont-reap t indicates that this file is not supposed to be deleted automatically for lack of use.

:dont-dump	t indicates that this file is not supposed to be dumped onto magnetic tape for backup purposes.
:characters	t indicates that this file contains characters (that is, text). nil indicates that the file contains binary data. This property, rather than the file's byte size, should be used to decide whether it is a text file.
:link-to	If the file is a link, this property is a string containing the name that the link points to.
:offline	T if the file's contents are not online.
:incremental-dump-date	The last time this file was dumped during an incremental dump (a universal time).
:incremental-dump-tape	The tape on which the last was saved in that incremental dump (a string).
:complete-dump-date	The last time this file was dumped during an full dump (a universal time).
:complete-dump-tape	The tape on which the last was saved in that full dump (a string).
:generation-retention-count	The number of files differing in version that are kept around.
:default-generation-retention-count	The generation-retention-count that a file ordinarily gets when it is created in this directory.
:auto-expunge-interval	The interval at which files are expunged from this directory, in seconds.
:date-last-expunged	The last (universal) time this directory was expunged, or nil.
:account	The account to which the file belongs, a string.
:protection	A system-dependent description of the protection of this file as a string.
:physical-volume	A string naming the physical volume on which the file is found.
:volume-name	A string naming the logical volume on which the file is found.
:pack-number	A string describing the pack on which this file is found.
:disk-space-description	A system-dependent description of the space usage on the file system. This usually appears in the plist that applies to the entire directory list.

The element in the directory list that has nil instead of a file's pathname describes the directory as a whole.

:physical-volume-free-blocks
: This property is an alist in which each element maps a physical volume name (a string) into a number, that is the number of free blocks on that volume.

:settable-properties
: This property is a list of file property names that may be set. This information is provided in the directory list because it is different for different file systems.

:pathname
: This property is the pathname from which this directory list was made.

:block-size
: This is the number of words in a block in this directory. It can be used to interpret the numbers of free blocks.

fs:directory-list-stream *filespec* &rest *options*

This is like fs:directory-list but returns the information in a different form. Instead of returning the directory list all at once, it returns a special kind of stream which gives out one element of the directory list at a time.

The directory list stream supports two operations: :entry and :close. :entry asks for the next element of the directory stream. :close closes any connection to a remote file server.

The purpose of using fs:directory-list-stream instead of fs:directory-list is that, when communicating with a remote file server, the directory list stream can give you some of the information without waiting for it to all be transmitted and parsed. This is desirable if the directory is being printed on the console.

directory *filespec*

Returns a list of pathnames (truenames) of the files in the directory specified by *filespec*. Wildcards are allowed. This is the Common Lisp way to find the contents of a directory.

fs:expunge-directory *filespec* &key (*error* t)

Expunges the directory specified in *filespec*; that is, permanently eliminates any deleted files in that directory. If *error* is nil, there is no error if the directory does not exist.

Note that not all file systems support this function. To find out whether a particular one does, send the :undeletable-p operation to a pathname. If it returns t, the file system of that pathname supports undeletion (and therefore expunging).

fs:create-directory *filespec* &key (*error* t)

Creates the directory specified in *filespec*. If *error* is nil, there is no error if the directory cannot be created; instead an error string is returned. Not all file servers support creation of directories.

fs:remote-connect *filespec* &key (*error* t) *access*

Performs the TOPS-20 "connect" or "access" function, or their equivalents, in a remote file server. Access is done if *access* is non-nil; otherwise, connect is done.

The connect operation grants you full access to the specified directory. The access operation grants you whatever access to all files and directories you would have if logged in on the specified directory. Both operations affect access only, since the connected directory of the remote server is never used by the Lisp Machine in choosing which file to operate on.

This function may ask you for a password if one is required for the directory you specify. If the operation cannot be performed, then if *error* is **nil**, an error object is returned.

File Properties:

fs:change-file-properties *file error-p* &rest *properties*

Changes one or more properties of the file *file*. The *properties* arguments are alternating keywords and values. If an error occurs accessing the file or changing the properties, the *error-p* argument controls what is done; if it is **nil**, a condition object describing the error is returned; if it is **t** a Lisp error is signaled. If no error occurs, **fs:change-file-properties** returns **t**.

Only some of the properties of a file may be changed; for instance, its creation date or its author. Exactly which properties may be changed depends on the host file system; a list of the changeable property names is the **:settable-properties** property of the file system as a whole, returned by **fs:directory-list** as explained above.

fs:file-properties *file* &optional (*error-p* **t**)

Returns a disembodied property list for a single file (compare this to **fs:directory-list**). The car of the returned list is the truename of the file and the cdr is an alternating list of indicators and values. The *error-p* argument is the same as in **fs:change-file-properties**.

Filename Completion:

fs:complete-pathname *defaults string type version* &rest *options*

string is a partially-specified file name. (Presumably it was typed in by a user and terminated with the **Altmode** key or the **End** key to request completion.) **fs:complete-pathname** looks in the file system on the appropriate host and returns a new, possibly more specific string. Any unambiguous abbreviations are expanded out in a host-dependent fashion.

defaults, *type*, and *version* are the arguments to be given to **fs:merge-pathname-defaults** (see page 558) when the user's input is eventually parsed and defaulted.

options are keywords (without following values) that control how the completion is performed. The following option keywords are allowed:

:deleted Looks for files which have been deleted but not yet expunged.

:read or **:in** The file is going to be read. This is the default.

:print or **:write** or **:out**
The file is going to be written (i.e. a new version is going to be created).

:old	Looks only for files that already exist. This is the default.
:new-ok	Allows either a file that already exists or a file that does not yet exist. An example of the use of this is the C-X C-F (Find File) command in the editor.

The first value returned is always a string containing a file name, either the original string or a new, more specific string. The second value returned indicates the success or failure of the completion. It is nil if an error occurred. One possible error is that the file is on a file system that does not support completion, in which case the original string is returned unchanged. Other possible second values are :old, which means that the string completed to the name of a file that exists, :new, which means that the string completed to the name of a file that could be created, and nil again, which means that there is no possible completion.

Balance Directories:

fs:balance-directories *filespec1 filespec2* &rest *options*

fs:balance-directories is a function for maintaining multiple copies of a directory. Often it is useful to maintain copies of your files on more than one machine; this function provides a simple way of keeping those copies up to date.

The function first parses *filespec1*, filling in missing components with wildcards (except for the version, which is :newest). Then *filespec2* is parsed with *filespec1* as the default. The resulting pathnames are used to generate directory lists using fs:directory-list. Note that the resulting directory lists need not be entire directories; any subset of a directory that fs:directory-list can produce will do.

First the directory lists are matched up on the basis of file name and type. All of the files in either directory list which have both the same name and the same type are grouped together.

The directory lists are next analyzed to determine if the directories are consistent, meaning that two files with the same name and type have equal creation-dates when their versions match, and greater versions have later creation-dates. If any inconsistencies are found, a warning message is printed on the console.

If the version specified for both *filespec1* and *filespec2* was :newest (the default), then the newest version of each file in each directory is copied to the other directory if it is not already there. The result is that each directory has the newest copy of every file in either of the two directories.

If one or both of the specified versions is not :newest, then *every* version that appears in one directory list and not in the other is copied. This has the result that the two directories are completely the same. (Note that this is probably not the right thing to use to *copy* an entire directory. Use copy-file with a wildcard argument instead.)

The *options* are keywords arguments which modify the operation. The following options are currently defined:

:ignore	This option takes one argument, which is a list of file names to ignore when making the directory lists. The default value is **nil**.
:error	This option is identical to the :error option to **open**.
:query-mode	This option takes one argument, which indicates whether or not the user should be asked before files are transferred. If the argument is **nil**, no querying is done. If it is :1->2, then only files being transferred from *filespec2* to *filespec1* are queried, while if it is :2->1, then files transferred from *filespec1* to *filespec2* are queried. If the argument is :always, then the user is asked about all files.
:copy-mode	This option is identical to the :copy-mode option of **copy-file**, and is used to control whether files are treated as binary or textual data.
:direction	This option specifies transfer of files in one direction only. If the value is :1->2 then files are transfered only from *filespec1* to *filespec2*, never in the other direction. If the value is :2->1 then files are transferred only from *filespec2* to *filespec1*. **nil**, the default, means transfer in either direction as appropriate.

25.7 Errors in Accessing Files

fs:file-error (error) *Condition Flavor*

This flavor is the basis for all errors signaled by the file system.

It defines two special operations, :pathname and :operation. Usually, these return the pathname of the file being operated on, and the operation used. This operation was performed either on the pathname object itself, or on a stream.

It defines prompting for the proceed types :retry-file-operation and :new-pathname, both of which are provided for many file errors. :retry-file-operation tries the operation again exactly as it was requested by the program; :new-pathname expects on argument, a pathname, and tries the same operation on this pathname instead of the original one.

fs:file-operation-failure (fs:file-error) *Condition*

This condition name signifies a problem with the file operation requested. It is an alternative to **fs:file-request-failure** (page 609), which means that the file system was unable to consider the operation properly.

All the following conditions in this section are always accompanied by **fs:file-operation-failure**, **fs:file-error**, and **error**, so they will not be mentioned.

fs:file-open-for-output *Condition*

The request cannot be performed because the file is open for output.

fs:file-locked *Condition*

The file cannot be accessed because it is already being accessed. Just which kinds of simultaneous access are allowed depends on the file system.

fs:circular-link *Condition*

A link could not be opened because it pointed, directly or indirectly through other links, to itself. In fact, some systems report this condition whenever a chain of links exceeds a fixed length.

fs:invalid-byte-size *Condition*

In open, the specified byte size was not valid for the particular file server or file.

fs:no-more-room *Condition*

Processing a request requires resources not available, such as space in a directory, or free disk blocks.

fs:filepos-out-of-range *Condition*

The :set-pointer operation was used with a pointer value outside the bounds of the file.

fs:not-available *Condition*

A requested pack, file, etc. exists but is currently off line or not available to users.

fs:file-lookup-error *Condition*

This condition name categorizes all sorts of failure to find a specified file, for any operation.

fs:device-not-found (fs:file-lookup-error) *Condition*

The specified device does not exist.

fs:directory-not-found (fs:file-lookup-error) *Condition*

The specified directory does not exist.

fs:file-not-found (fs:file-lookup-error) *Condition*

There is no file with the specified name, type and version. This implies that the device and directory do exist, or one of the errors described above would have been signaled.

fs:multiple-file-not-found (fs:file-lookup-error) *Condition*

There is no file with the specified name and any of the specified types, in with-open-file-search. Three special operations are defined:

:operation	Returns the function which used with-open-file-search, such as load.
:pathname	The base pathname used.
:pathnames	A list of all the pathnames that were looked for.

fs:link-target-not-found (fs:file-lookup-error) *Condition*

The file specified was a link, but the link's target filename fails to be found.

fs:access-error *Condition*
The operation is possible, but the file server is insubordinate and refuses to obey you.

fs:incorrect-access-to-file (access-error). *Condition*
fs:incorrect-access-to-directory (access-error). *Condition*
The file server refuses to obey you because of protection attached to the file (or, the directory).

fs:invalid-wildcard *Condition*
A pathname had a wildcard in a place where the particular file server does not support them. Such pathnames are not created by pathname parsing, but they can be created with the :new-pathname operation.

fs:wildcard-not-allowed *Condition*
A pathname with a wildcard was used in an operation that does not support it. For example, opening a file with a wildcard in its name.

fs:wrong-kind-of-file *Condition*
An operation was done on the wrong kind of file. If files and directories share one name space and it is an error to open a directory, the error possesses this condition name.

fs:creation-failure *Condition*
An attempt to create a file or directory failed for a reason specifically connected with creation.

fs:file-already-exists (fs:creation-failure) *Condition*
The file or directory to be created already exists.

fs:superior-not-directory (fs:creation-failure fs:wrong-kind-of-file) *Condition*
In file systems where directories and files share one name space, this error results from an attempt to create a file using a filename specifying a directory whose name exists in the file system but is not a directory.

fs:delete-failure *Condition*
A file to be deleted exists, but for some reason cannot be deleted.

fs:directory-not-empty (fs:delete-failure) *Condition*
A file could not be deleted because it is a directory and has files in it.

fs:dont-delete-flag-set (fs:delete-failure) *Condition*
A file could not be deleted because its "don't delete" flag is set.

fs:rename-failure *Condition*
A file to be renamed exists, but the renaming could not be done. The :new-pathname operation on the condition instance returns the specified new pathname, which may be a pathname or a string.

fs:rename-to-existing-file (fs:rename-failure) *Condition*
Renaming cannot be done because there is already a file with the specified new name.

fs:rename-across-directories (fs:rename-failure) *Condition*
Renaming cannot be done because the new pathname contains a different device or directory from the one the file is on. This may not always be an error—some file systems support it in certain cases—but when it is an error, it has this condition name.

fs:unknown-property (fs:change-property-failure) *Condition*
A property name specified in a :change-properties operation is not supported by the file server. (Some file servers support only a fixed set of property names.) The :property operation on the condition instance returns the problematical property name.

fs:invalid-property-value (fs:change-property-failure) *Condition*
In a :change-properties operation, some property was given a value that is not valid for it. The :property operation on the condition instance returns the property name, and the :value operation returns the specified value.

fs:invalid-property-name (fs:change-property-failure) *Condition*
In a :change-properties operation, a syntactically invalid property name was specified. This may be because it is too long to be stored. The :property operation on the condition instance returns the property name.

25.8 File Servers

Files on remote file servers are accessed using *file servers* over the Chaosnet. Normally connections to servers are established automatically when you try to use them, but there are a few ways you can interact with them explicitly.

When characters are written to a file server computer that normally uses the ASCII character set to store text, Lisp Machine characters are mapped into an encoding that is reasonably close to an ASCII transliteration of the text. When a file is written, the characters are converted into this encoding; the inverse transformation is done when a file is read back. No information is lost. Note that the length of a file, in characters, is not the same measured in original Lisp Machine characters as it is measured in the encoded ASCII characters. In the currently implemented ASCII file servers, the following encoding is used. All printing characters and any characters not mentioned explicitly here are represented as themselves. Codes 010 (lambda), 011 (gamma), 012 (delta), 014 (plus-minus), 015 (circle-plus), 177 (integral), 200 through 207 inclusive, 213 (Delete), and 216 and anything higher, are preceded by a 177; that is, 177 is used as a quoting character for these codes. Codes 210 (Overstrike), 211 (Tab), 212 (Line), and 214 (Page), are converted to their ASCII cognates, namely 010 (backspace), 011 (horizontal tab), 012 (line feed), and 014 (form feed) respectively. Code 215 (Return) is converted into 015 (carriage return) followed by 012 (line feed). Code 377 is ignored completely, and so cannot be stored in files.

When a file server is first created for you on a particular host, you must tell the server how to log in on that host. This involves specifying a *username*, and, if the obstructionists are in control of your site, a password. The Lisp Machine prompts you for these on the terminal when they are needed.

Logging in a file server is not the same thing as logging in on the Lisp Machine (see **login**, page 801). The latter identifies you as a user in general and involves specifying one host, your login host. The former identifies you to a particular file server host and must be done for each host on which you access files. However, logging in on the Lisp Machine does specify the username for your login host and logs in a file server there.

The Lisp Machine uses your username (or the part that follows the last period) as a first guess for your password (this happens to take no extra time). If that does not work, you are asked to type a password, or else a username and a password, on the keyboard. You do not have to give the same user name that you are logged in as, since you may have or use different user names on different machines.

Once a password is recorded for one host, the system uses that password as the guess if you connect to a file server on another host.

fs:user-unames *Variable*

This is an alist matching host names with the usernames you have specified on those hosts. Each element is the cons of a host object and the username, as a string.

For hosts running ITS, the symbol **fs:its** is used instead of a host object. This is because every user has the same username on all ITS hosts.

fs:user-host-password-alist *Variable*

Once you have specified a password for a given username and host, it is remembered for the duration of the session in this variable. The value is a list of elements, each of the form

((*username hostname*) *password*)

All three data are strings.

The remembered passwords are used if more than one file server is needed on the same host, or if the connection is broken and a new file server needs to be created.

If you are very scared of your password being known, you can turn off the recording by setting this variable:

fs:record-passwords-flag *Variable*

Passwords are recorded when typed in if this variable is non-**nil**.

You should set the variable at the front of your init file, and also set **fs:user-host-password-alist** to **nil**, since it will already have recorded your password when you logged in.

If you do not use a file server for a period of time, it is killed to save resources on the server host.

fs:host-unit-lifetime *Variable*

This is the length of time after which an idle file server connection should be closed, in 60ths of a second. The default is 20 minutes.

Some hosts have a caste system in which all users are not equal. It is sometimes necessary to enable one's privileges in order to exercise them. This is done with these functions:

fs:enable-capabilities *host* &rest *capabilities*
Enables the named capabilities on file servers for the specified host. *capabilities* is a list of strings, whose meanings depend on the particular file system that is available on *host*. If *capabilities* is nil, a default list of capabilities is enabled; the default is also dependent on the operating system type.

fs:disable-capabilities *host* &rest *capabilities*
Disables the named capabilities on file servers for the specified host. *capabilities* is a list of strings, whose meanings depend on the particular file system that is available on *host*. If *capabilities* is nil, a default list of capabilities is disabled; the default is also dependent on the operating system type.

The PEEK utility has a mode that displays the status of all your file connections, and of the *host unit* data structures that record them. Clicking on a connection with the mouse gets a menu of operations, of which the most interesting is reset. Resetting a host unit may be useful if the connection becomes hung.

25.8.1 Errors in Communication with File Servers

fs:file-request-failure (fs:file-error error) *Condition*
This condition name categorizes errors that prevent the file system from processing the request made by the program.

The following condition names are always accompanied by the more general classifications fs:file-request-failure, fs:file-error, and error.

fs:data-error *Condition*
This condition signifies inconsistent data found in the file system, indicating a failure in the file system software or hardware.

fs:host-not-available *Condition*
This condition signifies that the file server host is up, but refusing connections for file servers.

fs:network-lossage *Condition*
This condition signifies certain problems in the use of the Chaosnet by a file server, such as failure to open a data connection when it is expected.

fs:not-enough-resources *Condition*
This condition signifies a shortage of resources needed to consider processing a request, as opposed to resources used up by the request itself. This may include running out of network connections or job slots on the server host. It does not include running out of space in a directory or running out of disk space, because these are resources whose requirements come from processing the request.

fs:unknown-operation *Condition*

This condition signifies that the particular file system fails to implement a standardly defined operation; such as, expunging or undeletion on ITS.

26. The Chaosnet

The purpose of the basic software protocol of Chaosnet is to allow high-speed communication among processes on different machines, with no undetected transmission errors.

26.1 Chaosnet Overview

The principal service provided by Chaosnet is a *connection* between two user processes. This is a full-duplex reliable packet-transmission channel. The network undertakes never to garble, lose, duplicate, or resequence the packets; in the event of a serious error it may break the connection off entirely, informing both user processes. User programs may deal explicitly in terms of packets. They may also ignore packet boundaries and treat the connection as two uni-directional streams of 8-bit or 16-bit bytes, but this really works by means of packets.

If you just want to ask a question of another process or host and receive a reply, you can use a *simple transaction*: You send only one packet to the other host, and it sends one packet back. This is more efficient than establishing a connection and using it only briefly. In a simple transaction, the server cannot tell whether the user received the answer; and if the user does not receive the answer, it cannot tell whether the server received the question. In fact, the server might receive the question more than once. If this is unacceptable, a connection must be used.

Each node (or host) on the network is identified by an *address*, which is a 16-bit number. These addresses are used in the routing of packets. There is a table (the system host table, SYS: CHAOS; HOSTS TXT) that relates symbolic host names to numeric host addresses. The host table can record addresses on any number of different networks, and in certain contexts a host address is meaningful only together with the name of the network it is for.

The data transmitted over connections are in units called *packets*. Each packet contains an 8-bit number, the *opcode*, which indicates what its function is. Opcode numbers are always given in octal. Opcodes less than 200 (octal) are special purpose. Each such opcode that is used has an assigned name and a specific function. Users need not know about all of them. Opcodes 200 through 277 (octal) are used for 8-bit user data. Opcodes 300 through 377 (octal) are used for 16-bit user data.

Each packet also contains some number of data bytes, whose meaning depends on the opcode. If the opcode is for user data, then it is up to the application user software to decide on the interpretation.

Establishing a connection:

A connection is created because one process sends a request to a host. The request is a packet containing the special-purpose opcode RFC. The data contains a *contact name* which is used to find the process to connect to. There may be a process on the target host *listening* on this contact name. If so, it decides whether to agree to the connection. Alternatively, the contact name can be the name of a standard service such as TELNET. In this case, the receiving host creates a process to respond, loaded with the program for that service.

Once a connection has been established, there is no more need for the contact name and it is discarded. The Lisp Machine remembers what contact name was used to open a connection, but this is only for the user's information.

In the case where two existing processes that already know about each other want to establish a connection, they must agree on a contact name, and then one of them must send the request while the other listens. They must agree between themselves which is to do which.

Contact names are restricted to strings of upper-case letters, numbers, and ASCII punctuation. The maximum length of a contact name is limited only by the packet size, although on ITS hosts the names of automatically-started servers are limited by the file-system to six characters. The contact name is terminated by a space. If the RFC packet contains data beyond the contact name, it is just for interpretation by the listening process, which can also use it in deciding whether to accept the connection.

A simple transaction is also begun with an RFC packet. There is nothing in the RFC packet which indicates whether it is intended to start a connection or a simple transaction. The server has the option of doing either one. But normally any given server always does one or the other, and the requestor knows which one to expect.

The server accepts the request for a connection by sending an OPN packet (a packet with opcode OPN) to the requestor. It can also refuse the connection by sending a CLS packet. The data in the CLS packet is a string explaining the reason for the refusal. Another alternative is to tell the requestor to try a different host or a different contact name. This is called *forwarding* the request, and is done with a FWD packet.

The server can also respond with an answer, an ANS packet, which is the second half of a simple transaction. (Refusing and forwarding are also meaningful when a simple transaction is intended, just as when a connection is intended).

Once the connection is open:

Data transmitted through Chaosnet generally follow Lisp Machine standards. Bits and bytes are numbered from right to left, or least-significant to most-significant. The first 8-bit byte in a 16-bit word is the one in the arithmetically least-significant position. The first 16-bit word in a 32-bit double-word is the one in the arithmetically least-significant position. This is the "little-endian" convention.

Big-endian machines such as the PDP-10 need to reorder the characters in a word in order to access them conveniently. For their sake, some packet opcodes imply 8-bit data and some imply 16-bit data. Packets known to contain 8-bit bytes, including opcodes 200 through 277, are stored in the big-endian machine's memory a character at a time, whereas packets containing 16-bit data are stored 16 bits at a time.

The character set used is dictated by the higher-level protocol in use. Telnet and Supdup, for example, each specifies its own ASCII-based character set. The default character set—used for new protocols and for text that appears in the basic Chaosnet protocol, such as contact names—is the Lisp Machine character set.

If one process tries to send data faster than the other can process it, the buffered packets could devour lots of memory. Preventing this is the purpose of *flow control*. Each process specifies a *window size*, which is the number of packets that are allowed to be waiting for that process to read. Attempting to send on a connection whose other side's window is full waits until the other side reads some packets. The default window size is 13, but for some applications you might wish to specify a larger value (see chaos:connect, page 615). There is little reason ever to specify a smaller value.

Breaking a connection:

Either end of a connection can break the connection abruptly by sending a CLS packet. The data in this packet is a string describing why the connection was broken.

To break a connection gently, it is necessary to verify that all the data transmitted was received properly before sending a CLS. This matters in some applications and is unnecessary in others. When it is needed, it is done by sending a special packet, an EOF packet, which is mostly like a data packet except for its significance with regard to closing the connection. The EOF packet is like the words "the end" at the end of a book: it tells the recipient that it has received all the data it is supposed to receive, that there are no missing pages that should have followed. When the sender of the EOF sees the acknowledgement for the EOF packet, indicating that the EOF was received and understood, it can break the connection with a CLS.

If a process that expects to receive an EOF gets a CLS with no EOF, it takes this to mean that the connection was broken before the transmission was finished. If the process does receive an EOF, it does not break the connection itself immediately. It waits to see the sender of the EOF break it. If this does not happen in a few seconds, the EOF recipient can break the connection.

It is illegal to put data in an EOF packet; in other words, the byte count should always be zero. Most Chaosnet implementations simply ignore any data that is present in an EOF.

If both sides are sending data and both need to know for certain where "the end" is, they must do something a little more complicated. Arbitrarily call one party the user and the other the server. The protocol is that after sending all its data, each party sends an EOF and waits for it to be acknowledged. The server, having seen its EOF acknowledged, sends a second EOF. The user, having seen its EOF acknowledged, looks for a second EOF and *then* sends a CLS and goes away. The server goes away when it sees the user's CLS, or after a brief timeout has elapsed. This asymmetrical protocol guarantees that each side gets a chance to know that both sides agree that all the data have been transferred. The first CLS is sent only after both sides have waited for their (first) EOF to be acknowledged.

Clearing up inconsistencies:

If a host crashes, it is supposed to forget all the connections that it had. When a packet arrives on one of the former connections, the host will report "no such connection" to the sender with a LOS packet, whose data is a string explaining what happened. The same thing happens if a CLS packet is lost; the intended recipient may keep trying to use the connection that the other side (which sent the CLS) no longer believes should exist. LOS packets are used whenever a host receives a packet that it should not be getting; the recipient of the LOS packet knows that the connection it thought it was using does not exist any more.

26.2 Conns

On the Lisp Machine, your handle on a connection is a named structure of type chaos:conn. The conn may have an actual connection attached to it, or it may have a connection still being made, or record that a connection was refused, closed or broken.

chaos:inactive-state
: This conn is not really in use at all.

chaos:rfc-sent-state
: This conn was used to request a connection to another process, but no reply has been received. When the reply is received, it may change the conn's state to chaos:answered-state, chaos:cls-received-state, or chaos:open-state.

chaos:listening-state
: This conn is being used to listen with. If a RFC packet is received for the contact name you are listening on, the state changes to chaos:rfc-received-state.

chaos:rfc-received-state
: This means that your listen has "heard" an RFC packet that matches it. You can accept, reject, forward or answer the request. Accepting goes to state chaos:open-state; refusing or forwarding goes to to state chaos:inactive-state.

chaos:open-state
: This conn is one end of an open connection. You can receive any data packets that are waiting and you can transmit data.

chaos:answered-state
: This conn was used to send an RFC packet and an ANS packet was received in response (a simple transaction answer arrived). You can read the ANS packet, that is all.

chaos:cls-received-state
: This conn has received a CLS packet (the connection was closed or refused). You can read any data packets that came in before the CLS; after them you can read the CLS.

chaos:los-received-state
: This conn's connection was broken and the other end sent a LOS packet to say so. The LOS packet is the only packet available to be read.

chaos:host-down-state
: The host at the other end of this conn's connection has not responded to anything for a significant time.

chaos:foreign-state
: The connection is being used with a foreign protocol encapsulated in UNC packets (see the MIT AI Lab memo entitled "Chaosnet" for more information on this).

These are the fields of a conn that you might be interested in:

chaos:conn-state *conn*
This slot holds the state of *conn*. It is one of the symbols listed above.

chaos:conn-foreign-address *conn*
Returns the address of the host at the other end of this connection. Use si:get-host-from-address to find out which host this is (see page 577).

chaos:conn-read-pkts *conn*
Internally threaded chain of incoming packets available to be read from *conn*.

Its main use for the applications programmer is to test whether there are any incoming packets.

chaos:conn-window-available *conn*
Returns the number of packets you may transmit before the network software forces you to wait for the receiver to read some. This is just a minimum. By the time you actually send this many packets, the receiver may already have said he has room for some more.

chaos:conn-plist *conn*
This slot is used to store arbitrary properties on *conn*. You can store properties yourself; use property names that are not in the chaos package to avoid conflict.

chaos:contact-name *conn*
Returns the contact name with which *conn* was created. The contact name is not significant to the functioning of the connection once an RFC and LSN have matched, but it is remembered for the sake of debugging.

chaos:wait *conn state timeout* &optional *whostate*
Waits until the state of *conn* is not the symbol *state*, or until *timeout* 60ths of a second have elapsed. If the timeout occurs, nil is returned; otherwise t is returned. *whostate* is the process state to put in the who-line; it defaults to "Chaosnet wait".

26.3 Opening and Closing Connections

26.3.1 User-Side

chaos:connect *host contact-name* &optional *window-size timeout*
Opens a stream connection; returns a conn if it succeeds or else a string giving the reason for failure. *host* may be a number or the name of a known host. *contact-name* is a string containing the contact name and any additional arguments to go in the RFC packet. If *window-size* is not specified it defaults to 13. If *timeout* is not specified it defaults to 600 (ten seconds).

chaos:simple *host contact-name* &optional *timeout*

Taking arguments similar to those of chaos:connect, this performs the user side of a simple-transaction. The returned value is either an ANS packet or a string containing a failure message. The ANS packet should be disposed of (using chaos:return-pkt, see below) when you are done with it.

chaos:remove-conn *conn*

Makes *conn* null and void. It becomes inactive, all its buffered packets are freed, and the corresponding Chaosnet connection (if any) goes away. This is called *removing* the connection. *conn* itself is marked for reuse for another Chaosnet connection, so you should not do anything else with it after it is removed.

chaos:close-conn *conn* &optional *reason*

Closes and removes the connection. If it is open, a CLS packet is sent containing the string *reason*. Don't use this to reject RFC's; use chaos:reject for that.

chaos:open-foreign-connection *host index* &optional *pkt-allocation distinguished-port*

Creates a conn that may be used to transmit and receive foreign protocols encapsulated in UNC packets. *host* and *index* are the destination address for packets sent with chaos:send-unc-pkt. *pkt-allocation* is the 'window size', i.e. the maximum number of input packets that may be buffered. It defaults to 10. If *distinguished-port* is supplied, the local index is set to it. This is necessary for protocols that define the meanings of particular index numbers.

See the MIT AI Lab memo entitled "Chaosnet" for more information on using foreign protocols.

26.3.2 Server-Side

chaos:listen *contact-name* &optional *window-size wait-for-rfc*

Waits for an RFC for the specified contact name to arrive, then returns a conn that is in chaos:rfc-received-state. If *window-size* is not specified it defaults to 13. If *wait-for-rfc* is specified as nil (it defaults to t) then the conn is returned immediately without waiting for an RFC to arrive.

chaos:server-alist *Variable*

Contains an entry for each server that always exists. When an RFC arrives for one of these servers, the specified form is evaluated in the background process; typically it creates a process that will then do a chaos:listen. Use the add-initialization function to add entries to this list.

chaos:accept *conn*

conn must be in chaos:rfc-received-state. An OPN packet is transmitted and *conn* enters the chaos:open-state. If the RFC packet has not already been read with chaos:get-next-pkt, it is discarded. You should read it before accepting, if it contains arguments in addition to the contact name.

chaos:reject *conn reason*
conn must be in chaos:rfc-received-state. A CLS packet containing the string *reason* is sent and *conn* is removed from the connection table.

chaos:forward-all *contact-name host*
Causes all future requests for connection to this host on *contact-name* to be forwarded to the same contact name at host *host*.

chaos:answer-string *conn string*
conn must be in chaos:rfc-received-state. An ANS packet containing the string *string* is sent and *conn* is removed from the connection table.

chaos:answer *conn pkt*
conn must be in chaos:rfc-received-state. *pkt* is transmitted as an ANS packet and *conn* is removed. Use this function when the answer is some binary data rather than a text string.

chaos:fast-answer-string *contact-name string*
If a pending RFC exists to *contact-name*, an ANS containing *string* is sent in response to it and t is returned. Otherwise nil is returned. This function involves the minimum possible overhead. No conn is created.

26.4 Stream Input and Output

chaos:open-stream *host contact-name* &key *window-size timeout error direction characters ascii-translation*
Opens a Chaosnet connection and returns a stream that does I/O to it. *host* is the host to connect to; *contact-name* is the contact name at that host. These two arguments are passed along to chaos:connect.

If *host* is nil, a connection to *contact-name* is listened for, and a stream is returned as soon as a request comes in for that contact name. At this time, you must accept or reject the connection by invoking the stream operation :accept or :reject. Before you decide which to do, you can use the :foreign-host operation to find out where the connection came from.

The remaining arguments are:

window-size *timeout*	These two arguments specify two arguments for chaos:connect.
error	If the value is non-nil, a failure to connect causes a Lisp error. Otherwise, it causes a string describing the error to be returned.
direction *characters* *ascii-translation*	These three arguments are passed along to chaos:make-stream.

chaos:make-stream *conn* &key *direction characters ascii-translation*
Creates and returns a stream that does I/O on the connection *conn*, which should be open as a stream connection. *direction* may be :input, :output or :bidirectional.

If *characters* is non-nil (which is the default), the stream reads and writes 8-bit bytes. If *characters* is nil, the stream reads and writes 16-bit bytes.

If *ascii-translation* is non-nil, characters written to the stream are translated to standard ASCII before they are sent, and characters read are translated from ASCII to the Lisp Machine character set.

:foreign-host *Operation on* chaos:basic-stream
Returns the host object for the host at the other end of this stream's connection.

:accept *Operation on* chaos:basic-stream
Accepts the request for a connection which this stream received. Used only for streams made by chaos:open-stream with nil as the *host* argument.

:reject *reason-string* *Operation on* chaos:basic-stream
Rejects the request for a connection which this stream received, sending *reason-string* in the CLS packet as the reason. Used only for streams made by chaos:open-stream with nil as the *host* argument.

:close &optional *abort-p* *Operation on* chaos:basic-stream
Sends a CLS packet and removes the connection. For output connections and bidirectional connections, the :eof operation is performed first, if *abort-p* is nil.

:force-output *Operation on* chaos:basic-output-stream
Any buffered output is transmitted. Normally output is accumulated until a full packet's worth of bytes are available, so that maximum-size packets are transmitted.

:finish *Operation on* chaos:basic-output-stream
Waits until either all packets have been sent and acknowledged, or the connection ceases to be open. If successful, returns t; if the connection goes into a bad state, returns nil.

:eof *Operation on* chaos:basic-output-stream
Forces out any buffered output, sends an EOF packet, and does a :finish.

:clear-eof *Operation on* chaos:basic-input-stream
Allows you to read past an EOF packet on input. Normally, each :tyi done at eof returns nil or signals the specified eof error. If you do :clear-eof on the stream, you can then read more data (assuming there are data packets following the EOF packet).

26.5 Packet Input and Output

Input and output on a Chaosnet connection can be done at the whole-packet level, using the functions in this section. A packet is represented by a chaos:pkt data structure. Allocation of pkts is controlled by the system; each pkt that it gives you must be given back. There are functions to convert between pkts and strings. A pkt is an art-16b array containing the packet header and data; the leader of a pkt contains a number of fields used by the system.

chaos:first-data-word-in-pkt *Constant*
This is the index in any pkt of the element that is the first 16-bit word of user data. (Preceding elements are used to store a header used by the hardware.)

chaos:max-data-words-per-pkt *Constant*
The maximum number of 16-bit data words allowed in a packet.

chaos:pkt-opcode *pkt*
Accessor for the opcode of the packet *pkt*. To set the opcode, do

```
(setf (chaos:pkt-opcode my-pkt) my-opcode)
```

The system provides names for all the opcodes standardly used. The names useful to the applications programmer appear at the end of this section.

chaos:pkt-nbytes *pkt*
Accessor for the number-of-data-bytes field of *pkt*'s. This field says how much of *pkt*'s contects are valid data, measured in 8-bit bytes. This field can be set with setf also.

chaos:pkt-string *pkt*
An indirect array that is the data field of *pkt* as a string of 8-bit bytes. The length of this string is equal to (chaos:pkt-nbytes *pkt*). If you wish to record the contects of *pkt* permanently, you must copy this string.

chaos:set-pkt-string *pkt* &rest *strings*
Copies the *strings* into the data field of *pkt*, concatenating them, and sets (chaos:pkt-nbytes *pkt*) accordingly.

chaos:get-pkt
Allocates a pkt for use by the user.

chaos:return-pkt *pkt*
Returns *pkt* to the system for reuse. The packets given to you by chaos:get-pkt, chaos:get-next-pkt and chaos:simple should be returned to the system in this way when you are finished with them.

chaos:send-pkt *conn pkt* &optional (*opcode* chaos:dat-op)
Transmits *pkt* on *conn*. *pkt* should have been allocated with chaos:get-pkt and then had its data field and n-bytes filled in. *opcode* must be a data opcode (#o200 or more) or EOF. An error is signaled, with condition chaos:not-open-state, if *conn* is not open.

Giving a pkt to chaos:send-pkt constitutes giving it back to the system. You do not need to call chaos:return-pkt.

chaos:send-string *conn* &rest *strings*
Sends a data packet containing the concatenation of *strings* as its data.

chaos:send-unc-pkt *conn pkt* &optional *pkt-number ack-number*
Transmits *pkt*, an UNC packet, on *conn*. The opcode, packet number, and acknowledge number fields in the packet header are filled in (the latter two only if the optional arguments are supplied).

See the MIT AI Lab memo entitled "Chaosnet" for more information on using foreign protocols.

chaos:may-transmit *conn*
A predicate that returns t if there is any space in the window for transmitting on *conn*. If the value is nil, you may have to wait if you try to transmit. If the value is t, you certainly do not have to wait.

chaos:finish-conn *conn* &optional (*whostate* "Net Finish")
Waits until either all packets have been sent and acknowledged, or the connection ceases to be open. If successful, returns t; if the connection goes into a bad state, returns nil. *whostate* is the process state to display in the who-line while waiting.

chaos:conn-finished-p *conn*
t unless *conn* is open and has sent packets which have not been acknowledged.

chaos:get-next-pkt *conn* &optional (*no-hang-p* nil) *whostate check-conn-state*
Returns the next input packet from *conn*. When you are done with the packet you must give it back to the system with chaos:return-pkt. This can return an RFC, CLS, or ANS packet, in addition to data, UNC, or EOF.

If no packets are available, nil is returned if *no-hang-p* is t. Otherwise, chaos:get-next-pkt waits for a packet to come in or for the state to change. *whostate* is displayed in the who line; it defaults to "Chaosnet Input".

If *check-conn-state* is non-nil, the connection state is checked for validity before anything else is done, and an error is signaled if the connection is in a bad state, with condition name chaos:host-down, chaos:los-received-state, or chaos:read-on-closed-connection. If *check-conn-state* is nil and *no-hang-p* is t, nil is returned. *check-conn-state* defaults to (not *no-hang-p*).

chaos:data-available *conn*
A predicate that returns t if there any input packets available from *conn*.

Here are symbolic names for the opcodes that an applications programmer needs to know about:

chaos:rfc-op *Constant*
This special-purpose opcode is used for requesting a connection. The data consists of the contact name terminated by a space character, followed optionally by additional data whose meaning is up to the server for that contact name.

chaos:lsn-op *Constant*
This special-purpose opcode is used when you ask to listen on a contact name. The data is just the contact name. This packet is never actually sent over the network, just kept in the Chaosnet software and compared with the contact names in RFC packets that arrive.

chaos:opn-op *Constant*
This special-purpose opcode is used by the server process to accept the request for a connection conveyed by an RFC packet. Its data serves only internal functions.

chaos:ans-op *Constant*
This special-purpose opcode is used to send a simple reply. The simple reply is sent back in place of opening a connection.

chaos:los-op *Constant*
This special-purpose packet is what you receive if you try to use a connection that has been broken. Its data is a message explaining the situation, which you can print for the user.

chaos:cls-op *Constant*
This special-purpose packet is used to close a connection. Its data is a message explaining the reason, and it can be printed for the user. Note that you cannot count on receiving a CLS packet because it is not retransmitted if it is lost. If that happens you get a LOS when you try to use the connection (thinking it is still open).

CLS packets are also used for refusing to open a connection in the first place.

chaos:eof-op *Constant*
This special-purpose opcode is used to indicate the end of the data that you really want to transmit. When this packet is acknowledged by the other process, you know that all the real data was received properly. You can wait for this with **chaos:finish**. The EOF packet carries no data itself.

chaos:dat-op *Constant*
This is opcode 200 (octal), which is the normal opcode used for 8-bit user data. Some protocols use multiple data opcodes in the range 200 through 277, but simple protocols that do not need to distinguish types of packets just use opcode 200.

26.6 Connection Interrupts

chaos:interrupt-function *conn*

This attribute of a conn is a function to be called when certain events occur on this connection. Normally this is nil, which means not to call any function, but you can use setf to store a function here. Since the function is called in the Chaosnet background process, it should not do any operations that might have to wait for the network, since that could permanently hang the background process.

The function's first argument is one of the following symbols, giving the reason for the interrupt. The function's second argument is *conn*. Additional arguments may be present depending on the reason. The possible reasons are:

:input
: A packet has arrived for the connection when it had no input packets queued. It is now possible to do chaos:get-next-pkt without having to wait. There are no additional arguments.

:output
: An acknowledgement has arrived for the connection and made space in the window when formerly it was full. Additional output packets may now be transmitted with chaos:send-pkt without having to wait. There are no additional arguments.

:change-of-state
: The state of the connection has changed. The third argument to the function is the symbol for the new state.

chaos:read-pkts *conn*

Some interrupt functions want to look at the queued input packets of a connection when they get a :input interrupt. chaos:read-pkts returns the first packet available for reading. Successive packets can be found by following chaos:pkt-link.

chaos:pkt-link *pkt*

Lists of packets in the NCP are threaded together by storing each packet in the chaos:pkt-link of its predecessor. The list is terminated with nil.

26.7 Chaosnet Errors

sys:network-error (error) *Condition Flavor*

All errors from the Chaosnet code use flavors built on this one.

26.7.1 Local Problems

sys:local-network-error (sys:network-error error) *Condition Flavor*
This flavor is used for problems in connection with the Chaosnet that have entirely to do with what is going on in this Lisp Machine.

sys:network-resources-exhausted *Condition*
(sys:local-network-error sys:network-error error)
Signaled when some local resource in the NCP was exhausted. Most likely, there are too many Chaosnet connections and the connection table is full.

sys:unknown-address (sys:local-network-error sys:network-error error) *Condition*
The *address* argument to chaos:connect or some similar function was not recognizable. The :address operation on the condition instance returns the address that was supplied.

26.7.2 Problems Involving Other Machines' Actions

sys:remote-network-error (sys:network-error error) *Condition Flavor*
This flavor is used for network problems that involve the actions (or lack of them) of other machines. It is often useful to test for as a condition name.

The operations :connection and :foreign-host return the chaos:conn object and the host object for the foreign host.

All the condition names listed below imply the presence of sys:remote-network-error, sys:network-error and error. For brevity, these are not mentioned in the individual descriptions.

Every instance of sys:remote-network-error is either a sys:connection-error or a sys:bad-connection-state.

sys:connection-error *Condition*
This condition name categorizes failure to complete a connection.

sys:bad-connection-state *Condition*
This condition name categorizes errors where an existing, valid connection becomes invalid. The error is not signaled until you try to use the connection.

sys:host-not-responding *Condition*
This condition name categorizes errors where no packets whatever are received from the foreign host, making it seem likely that that host or the network is down.

sys:host-not-responding-during-connection *Condition*
(sys:connection-error sys:host-not-responding)
This condition is signaled when a host does not respond while it is being asked to make a connection.

sys:no-server-up (sys:connection-error) *Condition*

This condition is signaled by certain functions which request service from any available machine which can provide it, if no such machine is responding.

sys:host-stopped-responding *Condition*
(sys:bad-connection-state sys:host-not-responding)

This condition is signaled when a host does not respond even though a connection to it already exists.

sys:connection-refused (sys:connection-error) *Condition*

This is signaled when a connection is refused.

The :reason operation on the condition instance returns the reason specified in the CLS packet (a string) or nil if no reason was given.

sys:connection-closed (sys:bad-connection-state) *Condition*

This is signaled when you try to send on a connection which has been closed by the other host.

The :reason operation on the condition instance returns the reason specified in the CLS packet (a string) or nil if no reason was given.

sys:connection-lost (sys:bad-connection-state) *Condition*

This is signaled when you try to use a connection on which a LOS packet was received.

The :reason operation on the condition instance returns the reason specified in the CLS packet (a string) or nil if no reason was given.

sys:connection-no-more-data (sys:bad-connection-state) *Condition*

This is signaled when you try to read from a connection which has been closed by the other host, when there are no packets left to be read. (It is no error to read from a connection which has been closed, if you have not yet read all the packets which arrived, including the CLS packet).

The :reason operation on the condition instance returns the reason specified in the CLS packet (a string) or nil if no reason was given.

26.8 Information and Control

chaos:host-up-p *host* &optional (*timeout* 180.)

t if *host* responds over the Chaosnet within *timeout* sixtieths of a second, otherwise nil. The value is always nil if *host* is not on the Chaosnet.

chaos:up-hosts *host-list* &optional *number-of-hosts* (*timeout* 250.)

Returns a list of all the hosts in *host-list* which are currently responding over the Chaosnet. *host-list* is a list of host names and/or host objects. The value is always a list of host objects, possibly nil for none of them.

If *number-of-hosts* is non-nil, it should be a positive integer; when that many hosts have responded, chaos:up-hosts returns right away without bothering to listen for replies from the rest.

timeout is an integer; if a host fails to respond for that many sixtieths of a second, it is assumed to be down.

chaos:host-data &optional *host*
host may be a number or a known host name, and defaults to the local host. Two values are returned. The first value is the host name and the second is the host number. If the host is a number not in the table, it is asked its name using the STATUS protocol; if no response is received the name "Unknown" is returned.

chaos:print-conn *conn* &optional (*short* t)
Prints everything the system knows about the connection. If *short* is nil it also prints everything the system knows about each queued input and output packet on the connection.

chaos:print-pkt *pkt* &optional (*short* nil)
Prints everything the system knows about the packet, except its data field. If *short* is t, only the first line of the information is printed.

chaos:print-all-pkts *pkt* &optional (*short* t)
Calls chaos:print-pkt on *pkt* and all packets on the threaded list emanating from it.

chaos:status
Prints the hardware status.

chaos:reset
Resets the hardware and software and turns off Chaosnet communication.

chaos:assure-enabled
Turns on Chaosnet communication if it is not already on. It is normally always on unless you call one of the functions in this section.

chaos:enable
Resets the hardware and turns on Chaosnet communication.

chaos:disable
Resets the hardware and turns off Chaosnet communication.

chaos:show-routing-table *host* &optional (*stream* *standard-output)
Print out *host*'s routing table onto *stream*.

chaos:show-routing-path &key (*from si:local-host*) *to* (*stream* *standard-output)
Show how a packet would get from *from* to *to*. For this to work when the hosts are on different subnets, the bridge must respond to the DUMP-ROUTING-TABLE request.

The PEEK program has a mode that displays the status of all of the Lisp Machine's Chaosnet connections, and various other information, in a continuously updating fashion.

26.9 Higher-Level Protocols

This section briefly documents some of the higher-level protocols of the most general interest. There are quite a few other protocols which are too specialized to mention here. All protocols other than the STATUS protocol are optional and are only implemented by those hosts that need them. All hosts are required to implement the STATUS protocol since it is used for network maintenance.

The site files tell the Lisp Machine which hosts at your site implement certain higher-level protocols. See section 35.12, page 810.

26.9.1 Status

All network nodes, even bridges, are required to answer RFC's with contact name STATUS, returning an ANS packet in a simple transaction. This protocol is primarily used for network maintenance. The answer to a STATUS request should be generated by the Network Control Program, rather than by starting up a server process, in order to provide rapid response.

The STATUS protocol is used to determine whether a host is up, to determine whether an operable path through the network exists between two hosts, to monitor network error statistics, and to debug new Network Control Programs and new Chaosnet hardware. The **hostat** function on the Lisp Machine uses this protocol.

The first 32 bytes of the ANS contain the name of the node, padded on the right with zero bytes. The rest of the packet contains blocks of information expressed in 16-bit and 32-bit words, low byte first (little-endian convention). The low-order half of a 32-bit word comes first. Since ANS packets contain 8-bit data (not 16-bit), big-endian machines such as PDP-10s have to shuffle the bytes explicitly when using this protocol. The first 16-bit word in a block is its identification. The second 16-bit word is the number of 16-bit words to follow. The remaining words in the block depend on the identification.

This is the only block type currently defined. All items are optional, according to the count field, and extra items not defined here may be present and should be ignored. Note that items after the first two are 32-bit words.

word 0	A number between 400 and 777 octal. This is 400 plus a subnet number. This block contains information on this host's direct connection to that subnet.
word 1	The number of 16-bit words to follow, usually 16.
words 2-3	The number of packets received from this subnet.
words 4-5	The number of packets transmitted to this subnet.
words 6-7	The number of transmissions to this subnet aborted by collisions or because the receiver was busy.

words 8-9	The number of incoming packets from this subnet lost because the host had not yet read a previous packet out of the interface and consequently the interface could not capture the packet.
words 10-11	The number of incoming packets from this subnet with CRC errors. These were either transmitted wrong or damaged in transmission.
words 12-13	The number of incoming packets from this subnet that had no CRC error when received, but did have an error after being read out of the packet buffer. This error indicates either a hardware problem with the packet buffer or an incorrect packet length.
words 14-15	The number of incoming packets from this subnet that were rejected due to incorrect length (typically not a multiple of 16 bits).
words 16-17	The number of incoming packets from this subnet rejected for other reasons (e.g. too short to contain a header, garbage byte-count, forwarded too many times.)

If word 0, the identification, is a number between 0 and 377 octal, this is an obsolete format of block. The identification is a subnet number and the counts are as above except that they are only 16 bits instead of 32, and consequently may overflow. This format should no longer be sent by any hosts.

Identification numbers of 1000 octal and up are reserved for future use.

26.9.2 Routing Information

For network and NCP debugging, this RFC/ANS protocol should be implemented. The contact name is `DUMP-ROUTING-TABLE`, and the response is an ANS packet whose words alternate contain a method to getting to a subnet, and the cost. If the method is zero, then the machine knows of know what to get to that subnet. If the method is positive and less than 400 (octal), it is an interface of some kind to that subnet. If the method is 400 (octal) or greater, this is actually a bridge (host) off which the machine is bouncing packets destined for the subnet.

26.9.3 Telnet and Supdup

The Telnet and Supdup protocols of the Arpanet exist in identical form in Chaosnet. These protocols allow access to a computer system as an interactive terminal from another network node.

The contact names are `TELNET` and `SUPDUP`. The direct borrowing of the Telnet and Supdup protocols was eased by their use of 8-bit byte streams and of only a single connection. Note that these protocols define their own character sets, which differ from each other and from the Chaosnet standard character set.

For the Telnet protocol, refer to the Arpanet Protocol Handbook. For the Supdup protocol, see MIT AI Lab memo 644.

Chaosnet contains no counterpart of the INR/INS attention-getting feature of the Arpanet. The Telnet protocol sends a packet with opcode 201 octal in place of the INS signal. This is a controlled packet and hence does not provide the "out of band" feature of the Arpanet INS, however it is satisfactory for the Telnet 'interrupt process' and 'discard output' operations on the kinds of hosts attached to Chaosnet.

26.9.4 File Access

The FILE protocol is primarily used by Lisp Machines to access files on network file servers. ITS and TOPS-20 are equipped to act as file servers. A user end for the file protocol also exists for TOPS-20 and is used for general-purpose file transfer. For complete documentation on the file protocol, see SYS: DOC; FILE TEXT. The Arpanet file transfer protocols have not been implemented on the Chaosnet (except through the Arpanet gateway described below).

26.9.5 Mail

The MAIL protocol is used to transmit inter-user messages through the Chaosnet. The Arpanet mail protocol was not used because of its complexity and poor state of documentation. This simple protocol is by no means the last word in mail protocols; however, it is adequate for the mail systems we presently possess.

The sender of mail connects to contact name MAIL and establishes a stream connection. It then sends the names of all the recipients to which the mail is to be sent at (or via) the server host. The names are sent one to a line and terminated by a blank line (two carriage returns in a row). The Lisp Machine character set is used. A reply (see below) is immediately returned for each recipient. A recipient is typically just the name of a user, but it can be a user-atsign-host sequence or anything else acceptable to the mail system on the server machine. After sending the recipients, the sender sends the text of the message, terminated by an EOF. After the mail has been successfully swallowed, a reply is sent. After the sender of mail has read the reply, both sides close the connection.

In the MAIL protocol, a reply is a signal from the server to the user (or sender) indicating success or failure. The first character of a reply is a plus sign for success, a minus sign for permanent failure (e.g. no such user exists), or a percent sign for temporary failure (e.g. unable to receive message because disk is full). The rest of a reply is a human-readable character string explaining the situation, followed by a carriage return.

The message text transmitted through the mail protocol normally contains a header formatted in the Arpanet standard fashion. Refer to the Arpanet Protocols Handbook.

26.9.6 Send

The SEND protocol is used to transmit an interactive message (requiring immediate attention) between users. The sender connects to contact name `SEND` at the machine to which the recipient is logged in. The remainder of the RFC packet contains the name of the person being sent to. A stream connection is opened and the message is transmitted, followed by an EOF. Both sides close after following the end-of-data protocol described in page 613. The fact that the RFC was responded to affirmatively indicates that the recipient is in fact present and accepting messages. The message text should begin with a suitable header, naming the user that sent the message. The standard for such headers, not currently adhered to by all hosts, is one line formatted as in the following example:

```
        Moon@MIT-MC 6/15/81 02:20:17
```

Automatic reply to the sender can be implemented by searching for the first '@' and using the SEND protocol to the host following the '@' with the argument preceding it.

26.9.7 Name

The Name/Finger protocol of the Arpanet exists in identical form on the Chaosnet. Both Lisp Machines and timesharing machines support this protocol and provide a display of the user(s) currently logged in to them.

The contact name is `NAME`, which can be followed by a space and a string of arguments like the "command line" of the Arpanet Name protocol. A stream connection is established and the "finger" display is output in Lisp Machine character set, followed by an EOF.

Lisp Machines also support the FINGER protocol, a simple-transaction version of the NAME protocol. An RFC with contact name `FINGER` is transmitted and the response is an ANS containing the following items of information separated by carriage returns: the logged-in user ID, the location of the terminal, the idle time in minutes or hours-colon-minutes, the user's full name, and the user's group affiliation.

26.9.8 Time

The Time protocol allows a host such as a Lisp Machine that has no long-term timebase to ask the time of day. An RFC to contact name `TIME` evokes an ANS containing the universal time as a 32-bit number in four 8-bit bytes, least-significant byte first.

26.9.9 Uptime

This is similar to the TIME protocol, except that the contact name is UPTIME, and the time returned is actually an interval (in seconds) describing how long the host has been up.

26.9.10 Arpanet Gateway

This protocol allows a Chaosnet host to access almost any service on the Arpanet. The gateway server runs on each ITS host that is connected to both networks. It creates an Arpanet connection and a Chaosnet connection and forwards data bytes from one to the other. It also provides for a one-way auxiliary connection, used for the data connection of the Arpanet File Transfer Protocol.

The RFC packet contains a contact name of TCP, a space, the name of the Arpanet host to be connected to, optionally followed by a space and the contact-socket number in octal, which defaults to 1 if omitted. The name of host can also be an Internet-format address. The bi-directional 8-bit connection is made by connecting to the host with TCP.

If a data packet with opcode 201 (octal) is received, an Arpanet INS signal is transmitted. Any data bytes in this packet are transmitted normally. (This does nothing in the current server, since TCP does not define an interrupt signal.)

If a data packet with opcode 210 (octal) is received, an auxiliary connection on each network is opened. The first eight data bytes are the Chaosnet contact name for the auxiliary connection; the user should send an RFC with this name to the server. The next four data bytes are the TCP socket number to be connected to, in the wrong order, most-significant byte first. The byte-size of the auxiliary connection is 8 bits.

The normal closing of an TCP connection corresponds to an EOF packet. Closing due to an error, such as Host Dead, corresponds to a CLS packet.

26.9.11 Host Table

The HOSTAB protocol may be used to access tables of host addresses on other networks, such as the Arpanet or Internet. Servers for this protocol currently exist for Tenex, TOPS-20, ITS, and Lisp Machines.

The user connects to contact name HOSTAB, undertakes a number of transactions, then closes the connection. Each transaction is initiated by the user transmitting a host name followed by a carriage return. The server responds with information about that host, terminated with an EOF, and is then ready for another transaction. The server's response consists of a number of attributes of the host. Each attribute consists of an identifying name, a space character, the value of the attribute, and a carriage return. Values may be strings (free of carriage returns and *not* surrounded by double-quotes) or octal numbers. Attribute names and most values are in upper case. There can be more than one attribute with the same name; for example, a host may have more than one name or more than one network address.

The standard attribute names defined now are as follows. Note that more are likely to be added in the future.

`ERROR`	The value is an error message. The only error one might expect to get is "no such host".
`NAME`	The value is a name of the host. There may be more than one `NAME` attribute; the first one is always the official name, and any additional names are nicknames.
`MACHINE-TYPE`	The value is the type of machine, such as `LISPM`, `PDP10`, etc.
`SYSTEM-TYPE`	The value is the type of software running on the machine, such as `LISPM`, `ITS`, etc.
`ARPA`	The value is an address of the host on the Arpanet, in the form *host/imp*. The two numbers are decimal.
`CHAOS`	The value is an address of the host on Chaosnet, as an octal number.
`DIAL`	The value is an address of the host on Dialnet, as a telephone number.
`LCS`	The value is an address of the host on the LCSnet, as two octal numbers separated by a slash.
`SU`	The value is an address of the host on the SUnet, in the form *net#host*. The two numbers are octal.

26.9.12 Dover

A press file may be sent to the Dover printer at MIT by connecting to contact name DOVER at host AI-CHAOS-11. This host provides a protocol translation service that translates from Chaosnet stream protocol to the EFTP protocol spoken by the Dover printer. Only one file at a time can be sent to the Dover, so an attempt to use this service may be refused by a CLS packet containing the string "BUSY". Once the connection has been established, the press file is transmitted as a sequence of 8-bit bytes in data packets (opcode 200). It is necessary to provide packets rapidly enough to keep the Dover's program (Spruce) from timing out; a packet every five seconds suffices. Of course, packets are normally transmitted much more rapidly.

Once the file has been transmitted, an EOF packet must be sent. The transmitter must wait for that EOF to be acknowledged, then send a second one, and then close the connection. The two EOF's are necessary to provide the proper connection-closing sequence for the EFTP protocol. Once the press file has been transmitted to the Dover in this way and stored on the Dover's local disk, it will be processed, prepared for printing, and printed.

If an error message is returned by the Dover while the press file is being transmitted, it is reported back through the Chaosnet as a LOS containing the text of the error message. Such errors are fairly common; the sender of the press file should be prepared to retry the operation a few times.

Most programs that send press files to the Dover first wait for the Dover to be idle, using the Foreign Protocol mechanism of Chaosnet to check the status of the Dover. This is optional, but is courteous to other users since it prevents printing from being held up while additional files are sent to the Dover and queued on its local disk.

It would be possible to send to a press file to the Dover using its EFTP protocol through the Foreign Protocol mechanism, rather than using the AI-CHAOS-11 gateway service. This is not usually done because EFTP, which requires a handshake for every packet, tends to be very slow on a timesharing system.

26.9.13 Remote Disk

The Remote Disk server exists on Lisp Machines to allow other machines to refer to or modify the contents of the Lisp Machine's disk. Primarily this is used for printing and editing the disk label.

After first establishing a connection to contact name REMOTE-DISK, the user process sends commands as packets which contain a line of text, ending with a Return character. The text consists of a command name, a space, and arguments interpreted according to the command. The server processing the command may send disk data to the user, or it may read successive packets and write them to the disk. It is up to the user to know how many packets of disk data to read or send after each command. The commands are:

READ *unit block n-blocks*
: Reads *n-blocks* of data from disk *unit* starting at *block* and transmits their contents to the user process.

WRITE *unit block n-blocks*
: Reads data from the net connection and stores it into *n-blocks* disk blocks on disk *unit* starting at *block*.

SAY *text*
: Prints *text*, which is simply all the rest of the line following SAY, on the screen of the server host as a notification.

Each disk block is transmitted as three packets, the first two containing the data for 121 (decimal) Lisp Machine words, and the third containing the data for the remaining 14 (decimal) words of the disk block. Each packet's data ends with a checksum made by adding together all the 8-bit bytes of the actual disk data stored in the packet.

26.9.14 The Eval Server

The Eval server is available on Lisp Machines with contact name EVAL. It provides a read-eval-print loop which reads and prints using the Chaosnet connection. The data consists of text in the ASCII character set.

Each time a complete s-expression arrives, the Eval server reads it, evaluates it and prints the list of values back onto the network connection, followed by a CRLF. There is no way for the user process to tell the end of the output for a particular s-expression; the usual application is simply to copy all the output to a user's terminal asynchronously.

The Eval server is disabled when the Lisp Machine is logged in, unless the user requests to enable it.

chaos:eval-server-on *mode*

Turn the Eval server on this Lisp Machine on or off. *mode* can be t (on), nil (off), or :notify (on, but notify the user when a connection is made).

26.10 Using Higher Level Protocols

qsend *user* &optional *text*

Sends a message to another user. qsend is different from mail because it sends the message immediately; it will appear within seconds on the other user's screen, rather than being saved in her mail file.

user should be a string of the form "*username*@*hostname*"; *host* is the name of the Lisp Machine or timesharing system the user is currently logged-in to. Multiple recipients separated by commas are also allowed. *text* is a string which is the message. If *text* is not specified, you are prompted to type in a message.

Unlike mail and bug, qsend does not put up a window to allow you to compose the message; it just reads it from the input stream. Use Converse if you wish to compose sends in the editor. Converse can be invoked by typing System C. If you have started typing in a message to qsend, you can switch to Converse by typing Control-Meta-E ("Edit"). The text you have typed so far is transferred into Converse.

qsend does give you the ability to insert the text of the last message you received. Type Control-Meta-Y to do this.

reply &optional *text*
qreply &optional *text*

Sends *text* as a message to the last user who sent a message to you, like qsend with an appropriate first argument provided. The two names are synonymous.

chaos:shout &optional *message*

Sends *message* to every Lisp Machine at your site. If you do not specify *message*, it is read from *standard-input*.

print-sends

Reprints any messages that have been received. This is useful if you want to see a message again.

supdup &optional *host*

host may be a string or symbol, which is taken as a host name, or a number, which is taken as a host number. If no *host* is given, the machine you are logged-in to is assumed. This function opens a connection to the host over the Chaosnet using the Supdup protocol, and allows the Lisp Machine to be used as a terminal for any ITS, UNIX or TOPS-20 system.

To give commands to **supdup**, type the **Network** key followed by one character. Type **Network** followed by **Help** for documentation.

telnet &optional *host simulate-imlac*

telnet is similar to **supdup** but uses the Arpanet-standard Telnet protocol, simulating a printing terminal rather than a display terminal.

hostat &rest *hosts*

Asks each of the *hosts* for its status using the STATUS protocol, and prints the results. If no hosts are specified, all hosts on the Chaosnet are asked. Hosts can be specified either by name or by number.

For each host, a line is output that either says that the host is not responding or gives metering information for the host's network attachments. If a host is not responding, that usually means that it is down or there is no such host at that address. A Lisp Machine can fail to respond if it is looping inside **without-interrupts** or paging extremely heavily, such that it is simply unable to respond within a reasonable amount of time.

finger &optional *spec* (*stream* ***standard-output***)
whois &optional *spec* (*stream* ***standard-output***)

Prints brief (finger) or verbose (whois) information about a user or users specified by *spec*, on *stream*. *spec* can be a user name, @ followed by a host name, or a user name, @, and a host name. If there is no host name, the default login host is used. If there is no user name, all users on the host are described.
Examples:

```
(finger "@OZ")
(whois "RMS@OZ")
```

chaos:finger-all-lms &optional *stream print-free return-free hosts*

Prints a line of information about the user of each Lisp Machine in *hosts* (the default is all Lisp Machines at this site) on *stream* (default is ***standard-output***).

If *print-free* is non-**nil**, information on free Lisp Machines and nonresponding Lisp Machines is also printed.

If *return-free* is non-**nil**, then this function returns two values, the first a list of host objects of free Lisp Machines, the second a list of host objects of nonresponding Lisp Machines.

chaos:user-logged-into-host-p *username host*

Returns **t** if there is a user named *username* logged in on *host* (a host name or host object).

chaos:find-hosts-or-lispms-logged-in-as-user *user hosts*

Return a list of host objects for hosts on which *user* is logged in. All Lisp Machines at this site are checked, and so are *hosts* (which are presumably non-Lisp machines).

tv:close-all-servers *reason*

Close the connections of all network servers on this Lisp Machine, giving *reason* (a string) as the reason in the CLS packet.

Note that PEEK has a mode that displays information on the active network servers.

27. Packages

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 a session must be loaded into the same environment. Each of these programs is composed of a group of functions; apparently 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 which 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 there can't be two function definitions on the same symbol. This means that separate *name spaces*—mappings between names and symbols—must be provided for the two programs. The package system is designed to do just that.

Under the package system, the author of a program or a group of closely related programs identifies them together as a *package*. The package system associates a distinct name space with each package.

Here is an 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 (or something). 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 feature can be used to provide a separate name space 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, its symbols would be entered in the **chaos** package's name space. So when the Chaosnet program's **get-packet** referred to **allocate-pbuf**, the **allocate-pbuf** in the **chaos** name space would be found, which would be 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**.

In order to have multiple name spaces, the function **intern**, which searches for a name, must allow the name space to be specified. **intern** accepts an optional second argument which is the package to search.

It's obvious that every file has to 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. Luckily, this usually happens automatically.

The system can get the package of a source file from its `-*-` line. For instance, you can put at the front of your file a line such as

```
        ; -*- Mode:Lisp; Package:System-Internals -*-
```

The compiler puts the package name into the QFASL file for use when it is loaded. If a file doesn't have such a package specification in it, the system loads it into the current package and tells you what it did.

27.1 The Current Package

At any time, one package is the *current package*. By default, symbol lookup happens in the current package.

package *Variable*
package *Variable*

The value of the this variable is the current package. intern searches this package if it is not given a second argument. Many other functions for operating on packages also use this as the default.

Setting or binding the variable changes the current package. May the Goddess help you if you set it to something that isn't a package!

The two names are synonymous.

Each process or stack group can have its own setting for the current package by binding *package* with let. The actual current package at any time is the value bound by the process which is running. The bindings of another process are irrelevant until the process runs.

pkg-bind *pkg body...* *Macro*

pkg may be a package or a package name. The forms of the *body* are evaluated sequentially with the variable *package* bound to the package named by *pkg*.
Example:

```
        (pkg-bind "ZWEI"
          (read-from-string function-name))
```

When a file is loaded, *package* is bound to the correct package for the file (the one named in the file's `-*-` line). The Chaosnet program file has Package: Chaos; in the `-*-` line, and therefore its symbols are looked up in the chaos package. A QFASL file has an encoded representation of the `-*-` line of the source file; it looks different, but it serves the same purpose.

The current package is also relevant when you type Lisp expressions on the keyboard; it controls the reading of the symbols that you type. Initially it is the package user. You can select a different package using pkg-goto, or even by setqing *package*. If you are working with the Chaosnet program, it might be useful to type (pkg-goto 'chaos) so that your symbols are found in the chaos package by default. The Lisp listen loop binds *package* so that pkg-goto in one Lisp listener does not affect others, or any other processes whatever.

pkg-goto *package* &optional *globally*

Sets *package* to *package*. if *package* is suitable. (Autoexporting packages used by other packages are not suitable because it you could cause great troubles by interning new symbols in them). *package* may be specified as a package object or the name of one. If globally is non-nil, then this function also calls pkg-goto-globally (see below)

The Zmacs editor records the correct package for each buffer; it is determined from the file's -*- line. This package is used whenever expressions are read from the buffer. So if you edit the definition of the Chaosnet get-packet and recompile it, the new definition is read in the chaos package. The current buffer's package is also used for all expressions or symbols typed by the user. Thus, if you type Meta-. allocate-pbuf while looking at the Chaosnet program, you get the definition of the allocate-pbuf function in the chaos package.

The variable *package* also has a global binding, which is in effect in any process or stack group which does not rebind the variable. New processes that do bind *package* generally use the global binding to initialize their own bindings, doing (let ((*package* *package*)) ...). Therefore, it can be useful to set the global binding. But you cannot do this with setq or pkg-goto from a Lisp listener, or in a file, because that will set the local binding of *package* instead. Therefore you must use setq-globally (page 35) or pkg-goto-globally.

pkg-goto-globally *package*

Sets the global binding of *package* to *package*. An error is signaled if *package* is not suitable. Bindings of *package* other than the the global one are not changed, including the current binding if it is not the global one.

The name of the current package is always displayed in the middle of the who line, with a colon following it. This describes the process which the who line in general is describing; normally, the process of the selected window. No matter how the current package is changed, the who line will eventually show it (at one-second intervals). Thus, while a file is being loaded, the who line displays that file's package; in the editor, the who line displays the package of the selected buffer.

27.2 Package Prefixes

The separation of name spaces is not an uncrossable gulf. Consider a program for accessing files, using the Chaosnet. It may be useful to put it in a distinct package file-access, not chaos, so that the programs are protected from accidental name conflicts. But the file program cannot exist without referring to the functions of the Chaosnet program.

The colon character (':') has a special meaning to the Lisp reader. When the reader sees a colon preceded by the name of a package, it reads the next Lisp object with *package* bound to that package. Thus, to refer to the symbol connect in package chaos, we write chaos:connect. Some symbols documented in this manual require package prefixes to refer to them; they are always written with an appropriate prefix.

Similarly, if the **chaos** program wanted to refer to the **arpa** program's **allocate-pbuf** function (for some reason), it could use **arpa:allocate-pbuf**.

Package prefixes are printed on output also. If you would need a package prefix to refer to a symbol on input, then the symbol is printed with a suitable package prefix if it supposed to be printed readably (**prin1**, as opposed to **princ**). Just as the current package affects how a symbol is read, it also affects how the symbol is printed. A symbol available in the current package is never printed with a package prefix.

The printing of package prefixes makes it possible to print list structure containing symbols from many packages and read the text to produce an equal list with the same symbols in it—provided the current package when the text is read is the same one that was current when the text was printed.

The package name in a package prefix is read just like a symbol name. This means that escape characters can be used to include special characters in the package name. Thus, **foo/:bar:test** refers to the symbol **test** in the package whose name is "FOO:BAR", and so does **|FOO:BAR|:test**. Also, letters are converted to upper case unless they are escaped. For this reason, the actual name of a package is normally all upper case, but you can use either case when you write a package prefix.

In Common Lisp programs, simple colon prefixes are supposed to be used only for referring to external symbols (see page 642). To refer to other symbols, one is supposed to use two colons, as in **chaos::lose-it-later**. The Lisp machine tradition is to allow reference to any symbol with a single colon. Since this is upward compatible with what is allowed in Common Lisp, single-colon references are always allowed. However, double-colon prefixes are printed for internal symbols when Common Lisp syntax is in use, so that data printed on a Lisp Machine can be read by other Common Lisp implementations.

27.3 Home Packages of Symbols

Each symbol remembers one package which it belongs to: normally, the first one it was ever interned in. This package is available as **(symbol-package** *symbol***)**.

With **make-symbol** (see page 133) it is possible to create a symbol that has never been interned in any package. It is called an *uninterned symbol*, and it remains one as long as nobody interns it. The package cell of an uninterned symbol contains **nil**. Uninterned symbols print with **#:** as a prefix, as in **#:foo**. This syntax can be used as input to create an uninterned symbol with a specific name; but a new symbol is created each time you type it, since the mechanism which normally makes symbols unique is interning in a package. Thus, **(eq #:foo #:foo)** returns **nil**.

symbol-package *symbol*

Returns the contents of *symbol*'s package cell, which is the package which owns *symbol*, or **nil** if *symbol* is uninterned.

package-cell-location *symbol*

Returns a locative pointer to *symbol*'s package cell. It is preferable to write

```
(locf (symbol-package symbol))
```

rather than calling this function explicitly.

Printing of package prefixes is based on the contents of the symbol's package cell. If the cell contains the **chaos** package, then **chaos:** is printed as the prefix when a prefix is necessary. As a result of obscure actions involving interning and uninterning in multiple packages, the symbol may not actually be present in **chaos** any more. Then the printed prefix is inaccurate. This cannot be helped. If the symbol is not where it claims to be, there is no easy way to find wherever it might be.

27.4 Keywords

Distinct name spaces are useful for symbols which have function definitions or values, to enable them to be used independently by different programs.

Another way to use a symbol is to check for it with **eq**. Then there is no possibility of name conflict. For example, the function **open**, part of the file system, checks for the symbol **:error** in its input using **eq**. A user function might do the same thing. Then the symbol **:error** is meaningful in two contexts, but these meanings do not affect each other. The fact that a user program contains the code **(eq sym :error)** does not interfere with the function of system code which contains a similar expression.

There is no need to separate name spaces for symbols used in this way. In fact, it would be a disadvantage. If both the Chaosnet program and the Arpanet program wish to recognize a keyword named "address", for similar purposes (naturally), it is very useful for programs that can call either one if it is the *same* keyword for either program. But which should it be? **chaos:address**? **arpa:address**?

To avoid this uncertainty, one package called **keyword** has been set aside for the keywords of all programs. The Chaosnet and Arpanet programs would both look for **keyword:address**, normally written as just **:address**.

Symbols in **keyword** are the normal choice for names of keyword arguments; if you use **&key** to process them, code is automatically generated to look for for symbols in **keyword**. They are also the normal choice for flavor operation names, and for any set of named options meaningful in a specific context.

keyword and the symbols belonging to it are treated differently from other packages in a couple of ways designed to make them more convenient for this usage.

* Symbols belonging to **keyword** are constants; they always evaluate to themselves. (This is brought about by storing the symbol in its own value cell when the symbol is placed in the package). So you can write just **:error** rather than **':error**. The nature of the application of keywords is such that they would always be quoted if they were not constant.

* A colon by itself is a sufficient package prefix for **keyword**. This is because keywords are the most frequent application of package prefixes.

keywordp *object*
t if *object* is a symbol which belongs to the keyword package.

There are certain cases when a keyword should *not* be used for a symbol to be checked for with **eq**. Usually this is when the symbol 1) does not need to be known outside of a single program, and 2) is to be placed in shared data bases such as property lists of symbols which may sometimes be in **global** or **keyword**. For example, if the Chaosnet program were to record the existence of a host named CAR by placing an **:address** property on the symbol **:car**, or the symbol **car** (notice that **chaos:car** *is* **car**), it would risk conflicts with other programs that might wish to use the **:address** property of symbols in general. It is better to call the property **chaos:address**.

27.5 Inheritance between Name Spaces

In the simplest (but not the default) case, a package is independent of all other packages. This is not the default because it is not usually useful. Consider the standard Lisp function and variables names, such as **car**: how can the Chaosnet program, using the **chaos** package, access them? One way would be to install all of them in the **chaos** package, and every other package. But it is better to have one table of the standard Lisp symbols and refer to it where necessary. This is called *inheritance*. The single package **global** is the only one which actually contains the standard Lisp symbols; other packages such as **chaos** contain directions to "search **global** too".

Each package has a hash table of the symbols. The symbols in this table are said to be *present* (more explicitly, *present directly*) in the package, or *interned* in it. In addition, each package has a list of other packages to inherit from. By default, this list contains the package **global** and no others; but packages can be added and removed at any time with the functions **use-package** and **unuse-package**. We say that a package *uses* the packages it inherits from. Both the symbols present directly in the package and the symbols it inherits are said to be *available* in the package.

Here's how this works in the above example. When the Chaosnet program is read into the Lisp world, the current package would be the **chaos** package. Thus all of the symbols in the Chaosnet program would be interned in the **chaos** package. If there is a reference to a standard Lisp symbol such as **append**, nothing is found in the **chaos** package's own table; no symbol of that name is present directly in **chaos**. Therefore the packages used by **chaos** are searched, including **global**. Since **global** contains a symbol named **append**, that symbol is found. If, however, there is a reference to a symbol that is not standard, such as **get-packet**, the first time it is used it is not found in either **chaos** or **global**. So **intern** makes a new symbol named **get-packet**, and installs it in the **chaos** package. When **get-packet** is referred to later in the Chaosnet program, **intern** finds **get-packet** immediately in the **chaos** package. **global** does not need to be searched.

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 one that the Chaosnet program got. However, if it refers to **get-packet**, it does *not* get the same one the Chaosnet program got, because the **chaos** package is presumably not used by **arpa**. The **get-packet** in **chaos** not being available, no symbol is found, so a new one is created and placed in the **arpa** package. Further references in the Arpanet program find that **get-packet**.

This is the desired result: the packages share the standard Lisp symbols only.

Inheritance between other packages can also be useful, but it must be restricted: inheriting only some of the symbols of the used package. If the file access program refers frequently to the advertised symbols of the Chaosnet program—the connection states, such as **open-state**, functions such as **connect**, **listen** and **open-stream**, and others—it might be convenient to be able to refer to these symbols from the **file-access** package without need for package prefixes.

One way to do this is to place the appropriate symbols of the **chaos** package into the **file-access** package as well. Then they can be accessed by the file access program just like its own symbols. Such sharing of symbols between packages never happens from the ordinary operation of packages, but it can be requested explicitly using **import**.

import *symbols* &optional (*package* ***package***)

> Is the standard Common Lisp way to insert a specific symbol or symbols into a package. *symbols* is a symbol or a list of symbols. Each of the specified symbols becomes present directly in *package*.
>
> If a symbol with the same name is already present (directly or by inheritance) in *package*, an error is signaled. On proceeding, you can say whether to leave the old symbol there or replace it with the one specified in **import**.

But importing may not be the best solution. All callers of the Chaosnet program probably want to refer to the same set of symbols: the symbols described in the documentation of the Chaosnet program. It is simplest if the Chaosnet program, rather than each caller, says which symbols they are.

Restricted inheritance allows the **chaos** package to specify which of its symbols should be inheritable. Then **file-access** can use package **chaos** and the desired symbols are available in it.

The inheritable symbols of a package such as **chaos** in this example are called *external*; the other symbols are *internal*. Symbols are internal by default. The function **export** is how symbols are made external. Only the external symbols of a package are inherited by other packages which use it. This is true of **global** as well; Only external symbols in **global** are inherited. Since **global** exists only for inheritance, every symbol in it is external; in fact, any symbol placed in **global** is automatically made external. **global** is said to be *autoexporting*. A few other packages with special uses, such as **keyword** and **fonts**, are autoexporting. Ordinary packages such as **chaos**, which programs are loaded in, should not be.

If a request is made to find a name in a package, first the symbols present directly in that package are searched. If the name is not found that way, then all the packages in the used-list are searched; but only external symbols are accepted. Internal symbols found in the used packages are ignored. If a new symbol needs to be created and put into the name space, it is placed directly in the specified package. New symbols are never put into the inherited packages.

The used packages of a package are not in any particular order. It does not make any difference which one is searched first, because they are *not allowed* to have any conflicts among them. If you attempt to set up an inheritance situation where a conflict would exist, you get an error immediately. You can then specify explicitly how to resolve the conflict. See section 27.7, page 647.

The packages used by the packages used are *not* searched. If package **file-access** uses package **chaos** and file **mypackage** uses package **file-access**, this does not cause **mypackage** to inherit anything from **chaos**. This is desirable: the Chaosnet functions for whose sake **file-access** uses **chaos** are not needed in the programs in **mypackage** simply to enable them to communicate with **file-access**. If it is desirable for **mypackage** to inherit from **chaos**, that can be requested explicitly.

These functions are used to set up and control package inheritance.

use-package *packages* &optional (*in-package* ***package***)
: Makes *in-package* inherit symbols from *packages*, which should be either a single package or name for a package, or a list of packages and/or names for packages.

  This can cause a name conflict, if any of *packages* has a symbol whose name matches a symbol in *in-package*. In this case, an error is signaled, and you must resolve the conflict or abort.

unuse-package *packages* &optional (*in-package* ***package***)
: Makes *in-package* cease to inherit symbols from *packages*.

package-use-list *package*
: Returns the list of packages used by *package*.

package-used-by-list *package*
: Returns the list of packages which use *package*.

You can add or remove inheritance paths at any time, no matter what else you have done with the package.

These functions are used to make symbols external or internal in a package. By default, they operate on the current package.

export *symbols* &optional (*package* ***package***)
: Makes *symbols* external in *package*. *symbols* should be a symbol or string or a list of symbols and/or strings. The specified symbols or strings are interned in *package*, and the symbols found are marked external in *package*.

  If one of the specified symbols is found by inheritance from a used package, it is made directly present in *package* and then marked external there. (We know it was already external in the package it was inherited from.)

  Note that if a symbol is present directly in several packages, it can be marked external or internal in each package independently. Thus, it is the symbol's presence in a particular package which is external or not, rather than the symbol itself. **export** makes symbols external in whichever package you specify; if the same symbols are present directly in any other package, their status as external or internal in the other package is not affected.

unexport *symbols* &optional (*package* *package*)

Makes *symbols* not be external in *package*. An error occurs if any of the symbols fails to be directly present in *package*.

package-external-symbols *package*

Returns a list of all the external symbols of *package*.

globalize *name-or-symbol* &optional (*into-package* "GLOBAL")

Sometimes it will be discovered that a symbol which ought to be in global is not there, and the file defining it has already been loaded, thus mistakenly creating a symbol with that name in some other package. Creating a symbol in global would not fix the problem, since pointers to the misbegotten symbol already exist. Even worse, similarly named symbols may have been created mistakenly in other packages by code attempting to refer to the global symbol, and those symbols also are already pointed to. globalize is designed for use in correcting such a situation.

globalize *symbol-or-string* &optional (*package* "GLOBAL")

If *name-or-symbol* is a name (a string), interns the name in *into-package* and then forwards together all symbols with the same name in all the packages that use *into-package* as well as in *into-package* itself. These symbols are forwarded together so that they become effectively one symbol as far as the value, function definition and properties are concerned. The value of the composite is taken from whichever of the symbols had a value; a proceedable error is signaled if multiple, distinct values were found. The function definition is treated similarly, and so is each property that any of the symbols has.

If *name-or-symbol* is a symbol, globalize interns that symbol in *into-package* and then forwards the other symbols to that one.

The symbol which ultimately is present in *into-package* is also exported.

27.6 Packages and Interning

The most important service of the package system is to look up a name in a package and return the symbol which has that name in the package's name space. This is done by the function intern, and is called *interning*. When you type a symbol as input, read converts your characters to the actual symbol by calling intern.

The function intern allows you to specify a package as the second argument. It can be specified by giving either the package object itself or a string or symbol that is a name for the package. intern returns three values. The first is the interned symbol. The second is a keyword that says how the symbol was found. The third is the package in which the symbol was actually found. This can be either the specified package or one of its used packages.

When you don't specify the second argument to intern, the current package, which is the value of the symbol *package*, is used. This happens, in particular, when you call read and read calls intern. To specify the package for such functions to use, bind the symbol *package* temporarily to the desired package with pkg-bind.

There are actually four forms of the **intern** function: regular **intern**, **intern-soft**, **intern-local**, and **intern-local-soft**. **-soft** means that the symbol should not be added to the package if there isn't already one; in that case, all three values are **nil**. **-local** turns off inheritance; it means that the used packages should not be searched. Thus, **intern-local** can be used to cause shadowing. **intern-local-soft** is right when you want complete control over what packages to search and when to add symbols. All four forms of **intern** return the same three values, except that the **soft** forms return **nil nil nil** when the symbol isn't found.

intern *string-or-symbol* &optional (*pkg* ***package***)

The simplest case of **intern** is where *string-or-symbol* is a string. (It makes a big difference which one you use.) **intern** searches *pkg* and its used packages sequentially, looking for a symbol whose print-name is equal to *string-or-symbol*. If one is found, it is returned. Otherwise, a new symbol with *string-or-symbol* as print name is created, placed in package *pkg*, and returned.

The first value of **intern** is always the symbol found or created. The second value tells whether an existing symbol was found, and how. It is one of these four values:

:internal	A symbol was found present directly in *pkg*, and it was internal in *pkg*.
:external	A symbol was found present directly in *pkg*, and it was external in *pkg*.
:inherited	A symbol was found by inheritance from a package used by *pkg*. You can deduce that the symbol is external in that package.
nil	A new symbol was created

The third value returned by **intern** says which package the symbol found or created is present directly in. This is different from *pkg* if and only if if the second value is **:inherited.**

If *string-or-symbol* is a symbol, the search goes on just the same, using the print-name of *string-or-symbol* as the string to search for. But if no existing symbol is found, *string-or-symbol* itself is placed directly into *pkg*, just as **import** would do. No new symbol is created; *string-or-symbol itself* is the "new" symbol. This is done even if *string-or-symbol* is already present in another package. You can create arbitrary arrangements of sharing of symbols between packages this way.

Note: **intern** is sensitive to case; that is, it will consider two character strings different even if the only difference is one of upper-case versus lower-case. The reason that symbols get converted to upper-case when you type them in is that the reader converts the case of characters in symbols; the characters are converted to upper-case before **intern** is ever called. So if you call **intern** with a lower-case **"foo"** and then with an upper-case **"FOO"**, you won't get the same symbol.

intern-local *string-or-symbol* &optional (*pkg* ***package***)

Like **intern** but ignores inheritance. If a symbol whose name matches *string-or-symbol* is present directly in *pkg*, it is returned; otherwise *string-or-symbol* (if it is a symbol) or a new symbol (if *string-or-symbol* is a string) is placed directly in *pkg*.

intern-local returns second and third values with the same meaning as those of **intern**. However, the second value can never be **:inherited**, and the third value is always *pkg*.

The function **import** is implemented by passing the symbol to be imported to **intern-local**.

intern-soft *string* &optional (*pkg* ***package***)
find-symbol *string* &optional (*pkg* ***package***)

Like **intern** but never creates a symbol or modifies *pkg*. If no existing symbol is found, **nil** is returned for all three values. It makes no important difference if you pass a symbol instead of a string.

intern-soft returns second and third values with the same meaning as those of **intern**. However, if the second value is **nil**, it does not mean that a symbol was created, only that none was found. In this case, the third value is **nil** rather than a package.

find-symbol is the Common Lisp name for this function. The two names are synonymous.

intern-local-soft *string* &optional (*pkg* ***package***)

Like **intern-soft** but without inheritance. If a matching symbol is found directly present in *pkg*, it is returned; otherwise, the value is **nil**.

intern-local-soft returns second and third values with the same meaning as those of **intern**. However, if the second value is **nil**, it does not mean that a symbol was created, only that none was found. Also, it can never be **:inherited**. The third value is rather useless as it is either *pkg*, or **nil** if the second value is **nil**.

remob *symbol* &optional (*package* (**symbol-package** *symbol*))
unintern *symbol* &optional (*package* ***package***)

Both remove *symbol* from *package*. *symbol* itself is unaffected, but **intern** will no longer find it in *package*. *symbol* is not removed from any other package, even packages used by *package*, if it should be present in them. If *symbol* was present in *package* (and therefore, was removed) then the value is **t**; otherwise, the value is **nil**.

In **remob**, *package* defaults to the contents of the symbol's package cell, the package it belongs to. In **unintern**, *package* defaults to the current package. **unintern** is the Common Lisp version and **remob** is the traditional version.

If *package* is the package that *symbol* belongs to, then *symbol* is marked as uninterned: **nil** is stored in its package cell.

If a shadowing symbol is removed, a previously-hidden name conflict between distinct symbols with the same name in two used packages can suddenly be exposed, like a discovered check in chess. If this happens, an error is signaled.

27.7 Shadowing and Name Conflicts

In a package that uses **global**, it may be desirable to avoid inheriting a few standard Lisp symbols. Perhaps the user has defined a function **copy-list**, knowing that this symbol was not in **global**, and then a system function **copy-list** was created as part of supporting Common Lisp. Rather than changing the name in his program, he can *shadow* **copy-list** in the program's package. Shadowing a symbol in a package means putting a symbol in that package which hides any symbols with the same name which could otherwise have been inherited there. The symbol is explicitly marked as a *shadowing symbol* so that the name conflict does not result in an error.

Shadowing of symbols and shadowing of bindings are quite distinct. The same word is used for them because they are both examples of the general abstract concept of shadowing, which is meaningful whenever there is inheritance.

Shadowing can be done in the definition of a package (see page 652) or by calling the function **shadow**. **(shadow "COPY-LIST")** creates a new symbol named **copy-list** in the current package, regardless of any symbols with that name already available through inheritance. Once the new symbol is present directly in the package and marked as a shadowing symbol, the potentially inherited symbols are irrelevant.

shadow *names* &optional (*package* ***package***)

> Makes sure that shadowing symbols with the specified names exist in *package*. *names* is either a string or symbol or a list of such. If symbols are used, only their names matter; they are equivalent to strings. Each name specified is handled independently as follows:
>
> If there is a symbol of that name present directly in *package*, it is marked as a shadowing symbol, to avoid any complaints about name conflicts.
>
> Otherwise, a new symbol of that name is created and interned in *package*, and marked as a shadowing symbol.

Shadowing must be done before programs are loaded into the package, since if the programs are loaded without shadowing first they will contain pointers to the undesired inherited symbol. Merely shadowing the symbol at this point does not alter those pointers; only reloading the program and rebuilding its data structures from scratch can do that.

If it is necessary to refer to a shadowed symbol, it can be done using a package prefix, as in **global:copy-list**.

Shadowing is not only for symbols inherited from **global**; it can be used to reject inheritance of any symbol. Shadowing is the primary means of resolving *name conflicts* in which there multiple symbols with the same name are available, due to inheritance, in one package.

Name conflicts are not permitted to exist unless a resolution for the conflict has been stated in advance by specifying explicitly which symbol is actually to be seen in package. If no resolution has been specified, any command which would create a name conflict signals an error instead.

For example, a name conflict can be created by **use-package** if it adds a new used package with its own symbol **foo** to a package which already has or inherits a different symbol with the same name **foo**. **export** can cause a name conflict if the symbol becoming external is now supposed to be inherited by another package which already has a conflicting symbol. On either occasion, if shadowing has not already been performed to control the outcome, an error is signaled and the useage or exportation does not occur.

The conflict is resolved—in advance, always—by placing the preferred choice of symbol in the package directly, and marking it as a shadowing symbol. This can be done with the function shadowing-import. (Actually, you can proceed from the error and specify a resolution, but this works by shadowing and retrying. From the point of view of the retried operation, the resolution has been done in advance.)

shadowing-import *symbols* &optional (*package* *package*)

Interns the specified symbols in *package* and marks them as shadowing symbols. *symbols* must be a list of symbols or a single symbol; strings are not allowed.

Each symbol specified is placed directly into *package*, after first removing any symbol with the same name already interned in *package*. This is rather drastic, so it is best to use shadowing-import right after creating a package, when it is still empty.

shadowing-import is primarily useful for choosing one of several conflicting external symbols present in packages to be used.

Once a package has a shadowing symbol named foo in it, any other potentially conflicting external symbols with name foo can come and go in the inherited packages with no effect. It is therefore possible to perform the use-package of another package containing another foo, or to export the foo in one of the used packages, without getting an error.

In fact, shadow also marks the symbol it creates as a shadowing symbol. If it did not do so, it would be creating a name conflict and would always get an error.

package-shadowing-symbols *package*

Returns the list of shadowing symbols of *package*. Each of these is a symbol present directly in *package*. When a symbol is present directly in more than one package, it can be a shadowing symbol in one and not in another.

27.8 Styles of Using Packages

The unsophisticated user need never be aware of the existence of packages when writing his programs. His files are loaded into package user by default, and keyboard input is also read in user by default. Since all the functions that unsophisticated users are likely to need are provided in the global package, which user inherits from, they are all available without special effort. In this manual, functions that are not in the global package are documented with colons in their names, and they are all external, so typing the name the way it is documented does work in both traditional and Common Lisp syntax.

However, if you are writing a generally useful tool, you should put it in some package other than user, so that its internal functions will not conflict with names other users use. If your program contains more than a few files, it probably should have its own package just on the chance that someone else will use it someday along with other programs.

If your program is large, you can use multiple packages to help keep its modules independent. Use one package for each module, and export from it those of the module's symbols which are reasonable for other modules to refer to. Each package can use the packages of other modules that it refers to frequently.

27.9 Package Naming

A package has one name, also called the *primary name* for extra clarity, and can have in addition any number of *nicknames*. All of these names are defined globally, and all must be unique. An attempt to define a package with a name or nickname that is already in use is an error.

Either the name of a package or one of its nicknames counts as a *name for* the package. All of the functions described below that accept a package as an argument also accept a name for a package (either as a string, or as a symbol whose print-name is the name). Arguments that are lists of packages may also contain names among the elements.

When the package object is printed, its primary name is used. The name is also used by default when printing package prefixes of symbols. However, when you create the package you can specify that one of the nicknames should be used instead for this purpose. The name to be used for this is called the *prefix name*.

Case is significant in package name lookup. Usually package names should be all upper case. **read** converts package prefixes to upper case except for quoted characters, just as it does to symbol names, so the package prefix will match the package name no matter what case you type it in, as long as the actual name is upper case: TV:FOO and tv:foo refer to the same symbol. |tv|:foo is different from them, and normally erroneous since there is no package initially whose name is 'tv' in lower case.

In the functions find-package and pkg-find-package, and others which accept package names in place of packages, if you specify the name as a string you must give it in the correct case:

```
    (find-package "TV") => the tv package
    (find-package "tv") => nil
```

You can alternatively specify the name as a symbol; then the symbol's pname is used. Since **read** converts the symbol's name to upper case, you can type the symbol in either upper or lower case:

```
    (find-package 'TV) => the tv package
    (find-package 'tv) => the tv package
```

since both use the symbol whose pname is "TV".

Relevant functions:

package-name *package*
Returns the name of *package* (as a string).

package-nicknames *package*
Returns the list of nicknames (strings) of *package*. This does not include the name itself.

package-prefix-print-name *package*
Returns the name to be used for printing package prefixes that refer to *package*.

rename-package *package new-name* &optional *new-nicknames*
Makes *new-name* be the name for *package*, and makes *new-nicknames* (a list of strings, possibly nil) be its nicknames. An error is signaled if the new name or any of the new nicknames is already in use for some other package.

find-package *name* &optional *use-local-names-package*
Returns the package which *name* is a name for, or nil if there is none. If *use-local-names-package* is non-nil, the local nicknames of that package are checked first. Otherwise only actual names and nicknames are accepted. *use-local-names-package* should be supplied only when interpreting package prefixes.

If *name* is a package, it is simply returned.

If a list is supplied as *name*, it is interpreted as a specification of a package name and how to create it. The list should look like

(*name super-or-use size*)

or

(*name options*)

If *name* names a package, it is returned. Otherwise a package is created by passing *name* and the *options* to make-package.

pkg-find-package *name* &optional *create-p use-local-names-package*
Invokes find-package on *name* and returns the package that finds, if any. Otherwise, a package may be created, depending on *create-p* and possibly on how the user answers. These values of *create-p* are meaningful:

nil	An error is signaled if an existing package is not found.
t	A package is created, and returned.
:find	nil is returned.
:ask	The user is asked whether to create a package. If he answers Yes, a package is created and returned. If he answers No, nil is returned.

If a package is created, it is done by calling make-package with *name* as the only argument.

This function is not quite for historical compatibility only, since certain values of *create-p* provide useful features.

sys:package-not-found (error) *Condition*

is signaled by pkg-find-package with second argument :error, nil or omitted, when the package does not exist.

The condition instance supports the operations :name and :relative-to: these return whatever was passed as the first and third arguments to pkg-find-package (the package name, and the package whose local nicknames should be searched).

The proceed types that may be available include

:retry	says to search again for the specified name in case it has become defined; if it is still undefined, the error occurs again.
:create-package	says to search again for the specified name, and create a package with that name (and default characteristics) if none exists yet.
:new-name	is accompanied by a name (a string) as an argument. That name is used instead, ignoring any local nicknames. If that name too is not found, another error occurs.
:no-action	(available on errors from within read) says to continue with the entire read as well as is possible without having a valid package.

27.9.1 Local Nicknames for Packages

Suppose you wish to test new versions of the Chaosnet and file access programs. You could create new packages test-chaos and test-file-access, and use them for loading the new versions of the programs. Then the old, installed versions would not be affected; you could still use them to edit and save the files of the new versions. But one problem must be solved: when the new file access program says "chaos:connect" it must get test-chaos:connect rather than the actual chaos:connect.

This is accomplished by making "CHAOS" a local nickname for "TEST-CHAOS" in the context of the package test-file-access. This means that the when a chaos: prefix is encountered while reading in package test-file-access, it refers to test-chaos rather than chaos.

Local nicknames are allowed to conflict with global names and nicknames; in fact, they are rarely useful unless they conflict. The local nickname takes precedence over the global name.

It is necessary to have a way to override local nicknames. If you (pkg-goto 'test-file-access), you may wish to call a function in chaos (to make use of the old, working Chaosnet program). This can be done using #: as the package prefix instead of just :. #: inhibits the use of local nicknames when it is processed. It always refers to the package which is globally the owner of the name that is specified.

#: prefixes are printed whenever the package name printed is also a local nickname in the current package; that is, whenever an ordinary colon prefix would be misunderstood when read back

These are the functions which manage local nicknames.

pkg-add-relative-name *in-pkg name for-pkg*
Defines *name* as a local nickname in *in-pkg* for *for-pkg*. *in-pkg* and *for-pkg* may be packages, symbols or strings.

pkg-delete-relative-name *in-pkg name*
Eliminates *name* as a local nickname in *in-pkg*.

Looking up local nicknames is done with find-package, by providing a non-nil *use-local-names-package* argument.

27.10 Defining Packages

Before any package can be referred to or made current, it must be defined. This is done with the special form defpackage, which tells the package system all sorts of things, including the name of the package, what packages it should use, its estimated size, and some of the symbols which belong in it. The defpackage form is recognized by Zmacs as a definition of the package name.

defpackage *name* &key ... *Macro*
Defines a package named *name*. The alternating keywords and values are passed, unevaluated, to *make-package* to specify the rest of the information about how to construct the package.

If a package named *name* already exists, it is modified insofar as this is possible to correspond to the new definition.

Here are the possible options and their meanings

nicknames	A list of nicknames for the new package. The nicknames should be specified as strings.
size	A number; the new package is initially made large enough to hold at least this many symbols before a rehash is needed.
use	A list of packages or names for packages which the new package should inherit from, or a single name or package. It defaults to just the **global** package.
prefix-name	Specifies the name to use for printing package prefixes that refer to this package. It must be equal to either the package name or one of the nicknames. The default is to use the name.
invisible	If non-nil, means that this package should not be put on the list ***all-packages***. As a result, find-package will not find this package, not by its name and not by any of its nicknames. You can make normal use of the package in all other respects (passing it as the second argument to intern, passing it to use-package to make other packages inherit from it or it from others, and so on).

export
import
shadow
shadowing-import

If any of these arguments is non-nil, it is passed to the function of the same name, to operate on the package. Thus, if *shadow* is ("FOO" "BAR"), then
```
      (shadow this-package '("FOO" "BAR"))
```
is done.

You could accomplish as much by calling export, import, shadow or shadowing-import yourself, but it is clearer to specify all such things in one central place, the defpackage.

import-from If non-nil, is a list containing a package (or package name) followed by names of symbols to import from that package. Specifying *import-from* as (chaos "CONNECT" "LISTEN") is nearly the same as specifying *import* as (chaos:connect chaos:listen), the difference being that with *import-from* the symbols connect and listen are not looked up in the chaos package until it is time to import them.

super If non-nil, should be a package or name to be the superpackage of the new package. This means that the new package should inherit from that package, and also from all the packages that package inherits from. In addition, the superpackage is marked as autoexporting. Superpackages are obsolete and are implemented for compatibility only.

relative-names An alist specifying the local nicknames to have in this package for other packages. Each element looks like (*localname package*), where *package* is a package or a name for one, and *localname* is the desired local nickname.

relative-names-for-me

An alist specifying local nicknames by which this package can be referred to from other packages. Each element looks like (*package localname*), where *package* is a package name and *localname* is the name to refer to this package by from *package*.

For example, the system package eh could have been defined this way:
```
    (defpackage "EH" :size 1200
      :use ("GLOBAL" "SYS") :nicknames ("DBG" "DEBUGGER")
      :shadow ("ARG"))
```
It has room initially for at least 1200. symbols, nicknames dbg and debugger, uses system as well as global, and contains a symbol named arg which is not the same as the arg in global. You may note that the function eh:arg is documented in this manual (see page 734), as is the function arg (see page 238).

The packages of our inheritance example (page 642) might have been defined by

```
(defpackage 'chaos :size 1000 :use '(sys global)
   :export ("CONNECT" "OPEN-STREAM" "LISTEN" ...
            "OPEN-STATE" "RFC-RECEIVED-STATE" ...))

(defpackage 'file-access :size 1500
   :use '(chaos global)
   :export ("OPEN-FILE" "CLOSE-FILE" "DELETE-FILE" ...)
   :import (chaos:connect chaos:open-state))

(defpackage 'mypackage :size 400
   :use '(file-access global))
```

It is usually best to put the package definition in a separate file, which should be loaded into the user package. (It cannot be loaded into the package it is defining, and no other package has any reason to be preferred.) Often the files to be loaded into the package belong to one or a few systems; then it is often convenient to put the system definitions in the same file (see chapter 28, page 660).

A package can also be defined by the package attribute in a file's `-*-` line. Normally this specifies which (existing) package to load, compile or edit the file in. But if the attribute value is a list, as in

```
    -*-Package: (foo :size 300 :use (global system)); ...-*-
```

then loading, compiling or editing the file automatically creates package foo, if necessary with the specified options (just like defpackage options). No defpackage is needed. It is wise to use this feature only when the package is used for just a single file. For programs containing multiple files, it is good to make a system for them, and then convenient to put a defpackage near the defsystem.

make-package *name* &key *nicknames size use prefix-name invisible export shadow import shadowing-import import-from super relative-names relative-names-for-me*

Creates and returns new package with name *name*.

The meanings of the keyword arguments are described under defpackage (page 652).

pkg-create-package *name* &optional (*super* *package*) (*size* #o 200)

Creates a new package named *name* of size *size* with superpackage *super*. This function is obsolete.

kill-package *name-or-package*

Kills the package specified or named. It is removed from the list which is searched when package names are looked up.

package-declare *Macro*

package-declare is an older way of defining a package, obsolete but still used.

(package-declare *name superpackage size* nil
                        option-1 option-2 ...)

creates a package named *name* with initial size *size*.

super specifies the *superpackage* to use for this package. Superpackages were an old way of specifying inheritance; it was transitive, all symbols were inherited, and only one inheritance path could exist. If *super* is global, nothing special needs to be done; otherwise, the old superpackage facility is simulated using the *super* argument to make-package.

body is now allowed to contain only these types of elements:

(shadow *names*)
: Passes the names to the function SHADOW.

(intern *names*)
: Converts each name to a string and interns it in the package.

(refname *refname packagename*)
: Makes *refname* a local nickname in this package for the package named *packagename*.

(myrefname *packagename refname*)
: Makes *refname* a local nickname in the package named *packagename* for this package. If *packagename* is "GLOBAL", makes *refname* a global nickname for this package.

(external *names*)
: Does nothing. This controlled an old feature that no longer exists.

27.11 Operating on All the Symbols in a Package

To find and operate on every symbol present or available in a package, you can choose between iteration macros that resemble dolist and mapping functionals that resemble mapcar.

Note that all constructs that include inherited symbols in the iteration can process a symbol more than once. This is because a symbol can be directly present in more than one package. If it is directly present in the specified package and in one or more of the used packages, the symbol is processed once each time it is encountered. It is also possible for the iteration to include a symbol that is not actually available in the specified package. If that package shadows symbols present in the packages it uses, the shadowed symbols are processed anyway. If this is a problem, you can explicitly use intern-soft to see if the symbol handed to you is really available in the package. This test is not done by default because it is slow and rarely needed.

do-symbols (*var package result-form*) *body...* *Macro*

Executes *body* once for each symbol findable in *package* either directly or through inheritance. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

do-local-symbols (*var package result-form*) *body...* *Macro*

Executes *body* once for each symbol present directly in *package*. Inherited symbols are not considered. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

do-external-symbols (*var package result-form*) *body...* *Macro*
Executes *body* once for each external symbol findable in *package* either directly or through inheritance. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

do-local-external-symbols (*var package result-form*) *body...* *Macro*
Executes *body* once for each external symbol present directly in *package*. Inherited symbols are not considered. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

do-all-symbols (*var result-form*) *body...* *Macro*
Executes *body* once for each symbol present in any package. On each iteration, the variable *var* is bound to the next such symbol. Finally the *result-form* is executed and its values are returned.

Since a symbol can be directly present in more than one package, it is possible for the same symbol to be processed more than once.

mapatoms *function* &optional (*package* *package*) (*inherited-p* t)
function should be a function of one argument. mapatoms applies *function* to all of the symbols in *package*. If *inherited-p* is non-nil, then the function is applied to all symbols available in *package*, including inherited symbols.

mapatoms-all *function* &optional (*package* "GLOBAL")
function should be a function of one argument. mapatoms-all applies *function* to all of the symbols in *package* and all other packages which use *package*.

It is used by such functions as apropos and who-calls (see page 791)
Example:

```
(mapatoms-all
  #'(lambda (x)
      (and (alphalessp 'z x)
           (print x))))
```

27.12 Packages as Lisp Objects

A package is a conceptual name space; it is also a Lisp object which serves to record the contents of that name space, and is passed to functions such as intern to identify a name space.

packagep *object*
t if object is a package.

all-packages *Variable*
The value is a list of all packages, except for invisible ones (see the *invisble* argument to make-package, page 654).

list-all-packages
A Common Lisp function which returns *all-packages*.

pkg-global-package *Constant*
pkg-system-package *Constant*
pkg-keyword-package *Constant*
Respectively, the packages named global, system and keyword.

describe-package *package*
Prints everything there is to know about *package*, except for all the symbols interned in it. *package* can be specified as a package or as the name of one.

To see all the symbols interned in a package, do
```
(mapatoms 'print package)
```

27.13 Common Lisp and Packages

Common Lisp does not have defpackage or -*- lines in files. One is supposed to use the function in-package to specify which package a file is loaded in.

in-package *name* &key *nicknames use*
Creates a package named *name*, with specified nicknames and used packages, or modifies an existing package named *name* to have those nicknames and used packages.

Then *package* is set to this package.

Writing a call to in-package at the beginning of the file causes *package* to be set to that package for the rest of the file.

If you wish to use this technique for the sake of portability, it is best to have a -*- line with a package attribute also. While in-package does work for loading and compilation of the file, Zmacs does not respond to it.

In Common Lisp, the first argument to intern or find-symbol is required to be a symbol.

27.14 Initialization of the Package System

This section describes how the package system is initialized when generating a new software release of the Lisp Machine system; none of this should affect users.

The cold load, which contains the irreduceable minimum of the Lisp system needed for loading the rest, contains the code for packages, but no packages. Before it begins to read from the keyboard, it creates all the standard packages based on information in si:initial-packages, applying make-package to each element of it. At first all of the packages are empty. The symbols which belong in the packages global and system are recorded on lists which are made from the files SYS: SYS2; GLOBAL LISP and SYS: SYS2; SYSTEM LISP. Symbols referred to in the cold load which belong in packages other than si have strings (package names) in their package slots; scanning through the area which contains all the symbols, the package initializer puts each such symbol into the package it specifies, and all the rest into **si** unless they are already in **global** or **system**.

27.15 Initial Packages

The initially present packages include:

global	Contains advertised global functions.
user	The default current package for the user's type-in.
sys or **system**	Contains internal global symbols used by various system programs. Many system packages use **system**.
si or **system-internals**	Contains subroutines of many advertised system functions. Many files of the Lisp system are loaded in **si**.
compiler	Contains the compiler. **compiler** uses **sys**.
fs or **file-system**	Contains the code that deals with pathnames and accessing files. **fs** uses **sys**.
eh or **dbg**	Contains the error handler and the debugger. Uses **sys**.
cc or **cadr**	Contains the program that is used for debugging another machine. Uses **sys**.
chaos	Contains the Chaosnet controller. Uses **sys**.
tv	Contains the window system. Uses **sys**.
zwei	Contains the editor.
format	Contains the function **format** and its associated subfunctions.
cli	(Common Lisp Incompatible) contains symbols such as **cli:member** which the same pname as symbols in **global** but incompatible definitions.

There are quite a few others, but it would be pointless to list them all.

Packages that are used for special sorts of data:

fonts	Contains the names of all fonts.
format	Contains the keywords for **format**, as well as the code.
keyword	Contains all keyword symbols, symbols always written with a plain colon as a prefix. These symbols are peculiar in that they are automatically given themselves as values.

Here is a picture depicting the initial package inheritance structure

```
                      global                      keyword
                         |
  /------------------------------------\          fonts
   |     |          |          |       |
user  zwei      system      format   (etc)         cli
                    |
           /-----------------------------------\
           |           |     |     |    |      |
    system-internals  eh  chaos  cadr  fs  compiler
```

28. Maintaining Large Systems

When a program gets large, it is often desirable to split it up into several files. One reason for this is to help keep the parts of the program organized, to make things easier to find. It's also useful to have the program broken into small pieces that are more convenient to edit and compile. It is particularly important to avoid the need to recompile all of a large program every time any piece of it changes; if the program is broken up into many files, only the files that have changes in them need to be recompiled.

The apparent drawback to splitting up a program is that more commands are needed to manipulate it. To load the program, you now have to load several files separately, instead of just loading one file. To compile it, you have to figure out which files need compilation, by seeing which have been edited since they were last compiled, and then you have to compile those files.

What's even more complicated is that files can have interdependencies. You might have a file called DEFS that contains some macro definitions (or flavor or structure definitions), and functions in other files might use those macros. This means that in order to compile any of those other files, you must first load the file DEFS into the Lisp environment so that the macros will be defined and can be expanded at compile time. You have to remember this whenever you compile any of those files. Furthermore, if DEFS has changed, other files of the program may need to be recompiled because the macros may have changed and need to be re-expanded.

This chapter describes the *system* facility, which takes care of all these things for you. The way it works is that you define a set of files to be a *system*, using the defsystem special form, described below. This system definition says which files make up the system, which ones depend on the presence of others, and so on. You put this system definition into its own little file, and then all you have to do is load that file and the Lisp environment will know about your system and what files are in it. You can then use the make-system function (see page 666) to load in all the files of the system, recompile all the files that need compiling, and so on.

The system facility is very general and extensible. This chapter explains how to use it and how to extend it. This chapter also explains the *patch* facility, which lets you conveniently update a large program with incremental changes.

28.1 Defining a System

defsystem *name* (*keyword args...*)... *Macro*

Defines a system named *name*. The options selected by the keywords are explained in detail later. In general, they fall into two categories: properties of the system and *transformations*. A transformation is an operation such as compiling or loading that takes one or more files and does something to them. The simplest system is a set of files and a transformation to be performed on them.

Here are a few examples.

```
(defsystem mysys
  (:compile-load ("OZ:<GEORGE>PROG1.LISP" "OZ:<GEORGE2>PROG2.LISP")))

(defsystem zmail
  (:name "ZMail")
  (:pathname-default "SYS: ZMAIL;")
  (:package zwei)
  (:module defs "DEFS")
  (:module mult "MULT" :package tv)
  (:module main ("TOP" "COMNDS" "MAIL" "USER" "WINDOW"
                 "FILTER" mult "COMETH"))
  (:compile-load defs)
  (:compile-load main (:fasload defs)))

(defsystem bar
  (:module reader-macros "BAR:BAR;RDMAC")
  (:module other-macros "BAR:BAR;MACROS")
  (:module main-program "BAR:BAR;MAIN")
  (:compile-load reader-macros)
  (:compile-load other-macros (:fasload reader-macros))
  (:compile-load main-program (:fasload reader-macros
                                        other-macros)))
```

The first example defines a new *system* called **mysys**, which consists of two files, stored on a Tops-20 host names OZ, both of which are to be compiled and loaded. The second example is somewhat more complicated. What all the options mean is described below, but the primary difference is that there is a file **DEFS** which must be loaded before the rest of the files (**main**) can be compiled. Also, the files are stored on logical host SYS and directory ZMAIL.

The last example has two levels of dependency. **reader-macros** must be compiled and loaded before **other-macros** can be compiled. Both **reader-macros** and **other-macros** must then be loaded before **main-program** can be compiled. All the source files are stored on host BAR, presumably a logical host defined specifically for this system. It is desirable to use a logical host for the files of a system if there is a chance that people at more than one site will be using it; the logical host allows the identical **defsystem** to be valid at all sites. See section 24.7.5, page 572 for more on logical hosts and logical pathnames.

Note that The **defsystem** options other than transformations are:

:name Specifies a "pretty" version of the name for the system, for use in printing.

:short-name
Specified an abbreviated name used in constructing disk label comments and in patch file names for some file systems.

:component-systems
Specifies the names of other systems used to make up this system. Performing an operation on a system with component systems is equivalent to performing the same operation on all the individual systems. The format is (**:component-systems** *names*...).

:package
Specifies the package in which transformations are performed. A package specified here overrides one in the -*- line of the file in question.

:pathname-default
Gives a local default within the definition of the system for strings to be parsed into pathnames. Typically this specifies the directory, when all the files of a system are on the same directory.

:warnings-pathname-default
Gives a default for the file to use to store compiler warnings in, when **make-system** is used with the **:batch** option.

:patchable
Makes the system be a patchable system (see section 28.8, page 672). An optional argument specifies the directory to put patch files in. The default is the **:pathname-default** of the system.

:initial-status
Specifies what the status of the system should be when **make-system** is used to create a new major version. The default is **:experimental**. See section 28.8.5, page 679 for further details.

:not-in-disk-label
Make a patchable system not appear in the disk label comment. This should probably never be specified for a user system. It is used by patchable systems internal to the main Lisp system, to avoid cluttering up the label.

:default-binary-file-type
Specifies the file type to use for compiled Lisp files. The value you specify should be a string. If you do not specify this, the standard file type **:qfasl** is used.

:module
Allows assigning a name to a set of files within the system. This name can then be used instead of repeating the filenames. The format is (**:module** *name files options...*). *files* is usually a list of filenames (strings). In general, it is a *module-specification*, which can be any of the following:

a string
This is a file name.

a symbol
This is a module name. It stands for all of the files which are in that module of this system.

an *external module component*
This is a list of the form (*system-name module-names...*), to specify modules in another system. It stands for all of the files which are in all of those modules.

a list of *module components*
A module component is any of the above, or the following:

a list of file names
This is used in the case where the names of the input and output files of a transformation are not related according to the standard naming conventions, for

example when a QFASL file has a different name or resides on a different directory than the source file. The file names in the list are used from left to right, thus the first name is the source file. Each file name after the first in the list is defaulted from the previous one in the list.

To avoid syntactic ambiguity, this is allowed as a module component but not as a module specification.

The currently defined options for the :module clause are

:package	Overrides any package specified for the whole system for transformations performed on just this module.

In the second defsystem example above, there are three modules. Each of the first two has only one file, and the third one (main) is made up both of files and another module. To take examples of the other possibilities,

```
(:module prog (("SYS: GEORGE; PROG" "SYS: GEORG2; PROG")))
(:module foo (defs (zmail defs)))
```

The prog module consists of one file, but it lives in two directories, GEORGE and GEORG2. If this were a Lisp program, that would mean that the file SYS: GEORGE; PROG LISP would be compiled into SYS: GEORG2; PROG QFASL. The foo module consists of two other modules the defs module in the same system, and the defs module in the zmail system. It is not generally useful to compile files that belong to other systems; thus this foo module would not normally be the subject of a transformation. However, *dependencies* (defined below) use modules and need to be able to refer to (depend on) modules of other systems.

si:set-system-source-file *system-name filename*

This function specifies which file contains the defsystem for the system *system-name*. *filename* can be a pathname object or a string.

Sometimes it is useful to say where the definition of a system can be found without taking time to load that file. If make-system, or require (page 672), is ever used on that system, the file whose name has been specified will be loaded automatically.

28.2 Transformations

Transformations are of two types, simple and complex. A simple transformation is a single operation on a file, such as compiling it or loading it. A complex transformation takes the output from one transformation and performs another transformation on it, such as loading the results of compilation.

The general format of a simple transformation is (*name input dependencies condition*). *input* is usually a module specification or another transformation whose output is used. The transformation *name* is to be performed on all the files in the module, or all the output files of the other transformation.

dependencies and *condition* are optional.

dependencies is a *transformation specification*, either a list (*transformation-name module-names...*) or a list of such lists. A *module-name* is either a symbol that is the name of a module in the current system, or a list (*system-name module-names...*). A dependency declares that all of the indicated transformations must be performed on the indicated modules before the current transformation itself can take place. Thus in the zmail example above, the defs module must have the :fasload transformation performed on it before the :compile transformation can be performed on main.

The dependency has to be a tranformation that is explicitly specified as a transformation in the system definition, not just an action that might be performed by anything. That is, if you have a dependency (:fasload foo), it means that (fasload foo) is a tranformation of your system and you depend on that tranformation; it does not simply mean that you depend on foo's being loaded. Furthermore, it doesn't work if (:fasload foo) is an implicit piece of another tranformation. For example, the following works:

```
(defsystem foo
  (:module foo "FOO")
  (:module bar "BAR")
  (:compile-load (foo bar)))
```

but this doesn't work:

```
(defsystem foo
  (:module foo "FOO")
  (:module bar "BAR")
  (:module blort "BLORT")
  (:compile-load (foo bar))
  (:compile-load blort (:fasload foo)))
```

because foo's :fasload is not mentioned explicitly (i.e. at top level) but is only implicit in the (:compile-load (foo bar)). One must instead write:

```
(defsystem foo
  (:module foo "FOO")
  (:module bar "BAR")
  (:module blort "BLORT")
  (:compile-load foo)
  (:compile-load bar)
  (:compile-load blort (:fasload foo)))
```

condition is a predicate which specifies when the transformation should take place. Generally it defaults according to the type of the transformation. Conditions are discussed further on page 671.

The defined simple transformations are:

:fasload Calls the fasload function to load the indicated files, which must be QFASL files whose pathnames have canonical type :qfasl (see section 24.2.3, page 551). The *condition* defaults to si:file-newer-than-installed-p, which is t if a newer version of the file exists on the file computer than was read into the current environment.

:readfile Calls the readfile function to read in the indicated files, whose names must have canonical type :lisp. Use this for files that are not to be compiled. *condition* defaults to si:file-newer-than-installed-p.

:compile Calls the compile-file function to compile the indicated files, whose names must have canonical type :lisp. *condition* defaults to si:file-newer-than-file-p, which returns t if the source file has been written more recently than the binary file.

A special simple transformation is

:do-components
(:do-components *dependencies*) inside a system with component systems causes the *dependencies* to be done before anything in the component systems. This is useful when you have a module of macro files used by all of the component systems.

The defined complex transformations are

:compile-load (:compile-load *input compile-dependencies load-dependencies compile-condition load-condition*) is the same as (:fasload (:compile *input compile-dependencies compile-condition*) *load-dependencies load-condition*). This is the most commonly-used transformation. Everything after *input* is optional.

:compile-load-init
See page 671.

As was explained above, each filename in an input specification can in fact be a list of strings when the source file of a program differs from the binary file in more than just the file type. In fact, every filename is treated as if it were an infinite list of filenames with the last filename, or in the case of a single string the only filename, repeated forever at the end. Each simple transformation takes some number of input filename arguments and some number of output filename arguments. As transformations are performed, these arguments are taken from the front of the filename list. The input arguments are actually removed and the output arguments left as input arguments to the next higher transformation. To make this clearer, consider the prog module above having the :compile-load transformation performed on it. This means that prog is given as the input to the :compile transformation and the output from this transformation is given as the input to the :fasload transformation. The :compile transformation takes one input filename argument, the name of a Lisp source file, and one output filename argument, the name of the QFASL file. The :fasload transformation takes one input filename argument, the name of a QFASL file, and no output filename arguments. So, for the first and only file in the prog module, the filename argument list looks like ("SYS: GEORGE; PROG" "SYS: GEORG2; PROG" "SYS: GEORG2; PROG" ...). The :compile transformation is given arguments of "SYS: GEORGE; PROG" and "SYS: GEORG2; PROG" and the filename argument list which it outputs as the input to the :fasload transformation is ("SYS: GEORG2; PROG" "SYS: GEORG2; PROG" ...). The :fasload transformation then is given its one argument of "SYS: GEORG2; PROG".

Note that dependencies are not transitive or inherited. For example, if module a depends on macros defined in module b, and therefore needs b to be loaded in order to compile, and b has a similar dependency on c, c need not be loaded for compilation of a. Transformations with these dependencies would be written

```
        (:compile-load a (:fasload b))
        (:compile-load b (:fasload c))
```
To say that compilation of a depends on both b and c, you would instead write
```
        (:compile-load a (:fasload b c))
        (:compile-load b (:fasload c))
```
If in addition a depended on c (but not b) during loading (perhaps a contains defvars whose initial values depend on functions or special variables defined in c) you would write the transformations
```
        (:compile-load a (:fasload b c) (:fasload c))
        (:compile-load b (:fasload c))
```

28.3 Making a System

make-system *name* &rest *keywords*

The make-system function does the actual work of compiling and loading. In the example above, if PROG1 and PROG2 have both been compiled recently, then
```
        (make-system 'mysys)
```
loads them as necessary. If either one might also need to be compiled, then
```
        (make-system 'mysys :compile)
```
does that first as necessary.

The very first thing make-system does is check whether the file which contains the defsystem for the specified system has changed since it was loaded. If so, it offers to load the latest version, so that the remainder of the make-system can be done using the latest system definition. (This only happens if the filetype of that file is LISP.) After loading this file or not, make-system goes on to process the files that compose the system.

If the system name is not recognized, make-system attempts to load the file SYS: SITE; *system-name* SYSTEM, in the hope that that contains a system definition or a call to si:set-system-source-file.

make-system lists what transformations it is going to perform on what files, then asks the user for confirmation. If the user types S when confirmation is requested, then make-system asks about each file individually so that the user can decide selectively which transformations should be performed; then collective reconfirmation is requested. This is like what happens if the :selective keyword is specified. If the user types Y, the transformations are performed. Before each transformation a message is printed listing the transformation being performed, the file it is being done to, and the package. This behavior can be altered by *keywords*.

If the system being made is patchable, and if loading has not been inhibited, then the system's patches are loaded afterward. Loading of patches is silent if the make-system is, and requires confirmation if the make-system does.

These are the keywords recognized by the make-system function and what they do.

:noconfirm	Assumes a yes answer for all questions that would otherwise be asked of the user.

:selective	Asks the user whether or not to perform each transformation that appears to be needed for each file.
:silent	Avoids printing out each transformation as it is performed.
:reload	Bypasses the specified conditions for performing a transformation. Thus files are compiled even if they haven't changed and loaded even if they aren't newer than the installed version.
:noload	Does not load any files except those required by dependencies. For use in conjunction with the :compile option.
:compile	Compiles files also if need be. The default is to load but not compile.
:recompile	This is equivalent to a combination of :compile and :reload: it specifies compilation of all files, even those whose sources have not changed since last compiled.
:no-increment-patch	When given along with the :compile option, disables the automatic incrementing of the major system version that would otherwise take place. See section 28.8, page 672.
:increment-patch	Increments a patchable system's major version without doing any compilations. See section 28.8, page 672.
:no-reload-system-declaration	Turns off the check for whether the file containing the defsystem has been changed. Then the file is loaded only if it has never been loaded before.
:batch	Allows a large compilation to be done unattended. It acts like :noconfirm with regard to questions, turns off more-processing and fdefine-warnings (see inhibit-fdefine-warnings, page 240), and saves the compiler warnings in an editor buffer and a file (it asks you for the name).
:defaulted-batch	This is like :batch except that it uses the default for the pathname to store warnings in and does not ask the user to type a pathname.
:print-only	Just prints out what transformations would be performed; does not actually do any compiling or loading.
:noop	Is ignored. This is useful mainly for programs that call make-system, so that such programs can include forms like `(make-system 'mysys (if compile-p :compile :noop))`

28.4 Adding New Keywords to make-system

make-system keywords are defined as functions on the si:make-system-keyword property of the keyword. The functions are called with no arguments. Some of the relevant variables they can use are

si:*system-being-made* *Variable*
The internal data structure that represents the system being made.

si:*make-system-forms-to-be-evaled-before* *Variable*
A list of forms that are evaluated before the transformations are performed.

si:*make-system-forms-to-be-evaled-after* *Variable*
A list of forms that are evaluated after the transformations have been performed. Transformations can push entries here too.

si:*make-system-forms-to-be-evaled-finally* *Variable*
A list of forms that are evaluated by an unwind-protect when the body of make-system is exited, whether it is completed or not. Closing the batch warnings file is done here. Unlike the si:*make-system-forms-to-be-evaled-after* forms, these forms are evaluated outside of the "compiler warnings context".

si:*query-type* *Variable*
Controls how questions are asked. Its normal value is :normal. :noconfirm means ask no questions and :selective means asks a question for each individual file transformation.

si:*silent-p* *Variable*
If t, no messages are printed out.

si:*batch-mode-p* *Variable*
If t, :batch was specified.

si:*redo-all* *Variable*
If t, all transformations are performed, regardless of the condition functions.

si:*top-level-transformations* *Variable*
A list of the types of transformations that should be performed, such as (:fasload :readfile). The contents of this list are controlled by the keywords given to make-system. This list then controls which transformations are actually performed.

si:*file-transformation-function* *Variable*
The actual function that gets called with the list of transformations that need to be performed. The default is si:do-file-transformations.

si:define-make-system-special-variable *variable value* [*defvar-p*] *Macro*
Causes *variable* to be bound to *value* during the body of the call to make-system. This allows you to define new variables similar to those listed above. *value* is evaluated on entry to make-system. If *defvar-p* is specified as (or defaulted to) t, *variable* is defined with defvar. It is not given an initial value. If *defvar-p* is specified as *nil*, *variable* belongs to some other program and is not defvar'ed here.

The following simple example adds a new keyword to make-system called :just-warn, which means that fdefine warnings (see page 239) regarding functions being overwritten should be printed out, but the user should not be queried.

```
(si:define-make-system-special-variable
   inhibit-fdefine-warnings inhibit-fdefine-warnings nil)

(defun (:just-warn si:make-system-keyword) ()
  (setq inhibit-fdefine-warnings :just-warn))
```

(See the description of the inhibit-fdefine-warnings variable, on page 240.)

make-system keywords can do something directly when called, or they can have their effect by pushing a form to be evaluated onto si:*make-system-forms-to-be-evaled-after* or one of the other two similar lists. In general, the only useful thing to do is to set some special variable defined by si:define-make-system-special-variable. In addition to the ones mentioned above, user-defined transformations may have their behavior controlled by new special variables, which can be set by new keywords. If you want to get at the list of transformations to be performed, for example, the right way is to set si:*file-transformation-function* to a new function, which then can call si:do-file-transformations with a possibly modified list. That is how the :print-only keyword works.

28.5 Adding New Options for defsystem

Options to defsystem are defined as macros on the si:defsystem-macro property of the option keyword. Such a macro can expand into an existing option or transformation, or it can have side effects and return nil. There are several variables they can use; the only one of general interest is

si:*system-being-defined* *Variable*
The internal data structure that represents the system that is currently being constructed.

si:define-defsystem-special-variable *variable value* *Macro*
Causes *value* to be evaluated and *variable* to be bound to the result during the expansion of the defsystem special form. This allows you to define new variables similar to the one listed above.

si:define-simple-transformation *Macro*
This is the most convenient way to define a new simple transformation. The form is

(si:define-simple-transformation *name function*
default-condition input-file-types output-file-types
pretty-names compile-like load-like)

For example,

```
(si:define-simple-transformation :compile si:qc-file-1
          si:file-newer-than-file-p (:lisp) (:qfasl))
```

input-file-types and *output-file-types* are how a transformation specifies how many input filenames and output filenames it should receive as arguments, in this case one of each. They also, obviously, specify the default file type for these pathnames. The si:qc-file-1 function is mostly like compile-file, except for its interface to packages. It takes input-file and output-file arguments.

pretty-names, *compile-like*, and *load-like* are optional.

pretty-names specifies how messages printed for the user should print the name of the transformation. It can be a list of the imperative ("Compile"), the present participle ("Compiling"), and the past participle ("compiled"). Note that the past participle is not capitalized, because when used it does not come at the beginning of a sentence. *pretty-names* can be just a string, which is taken to be the imperative, and the system will conjugate the participles itself. If *pretty-names* is omitted or nil it defaults to the name of the transformation.

compile-like and *load-like* say when the transformation should be performed. Compile-like transformations are performed when the :compile keyword is given to make-system. Load-like transformations are performed unless the :noload keyword is given to make-system. By default *compile-like* is t but *load-like* is nil.

Complex transformations are defined as normal macro expansions, for example,

```
(defmacro (:compile-load si:defsystem-macro)
                (input &optional com-dep load-dep
                                 com-cond load-cond)
  '(:fasload (:compile ,input ,com-dep ,com-cond)
             ,load-dep ,load-cond))
```

28.6 More Esoteric Transformations

It is sometimes useful to specify a transformation upon which something else can depend, but which is performed not by default, but rather only when requested because of that dependency. The transformation nevertheless occupies a specific place in the hierarchy. The :skip defsystem macro allows specifying a transformation of this type. For example, suppose there is a special compiler for the read table which is not ordinarily loaded into the system. The compiled version should still be kept up to date, and it needs to be loaded if ever the read table needs to be recompiled.

```
(defsystem reader
  (:pathname-default "SYS: IO;")
  (:package system-internals)
  (:module defs "RDDEFS")
  (:module reader "READ")
  (:module read-table-compiler "RTC")
  (:module read-table "RDTBL")
  (:compile-load defs)
  (:compile-load reader (:fasload defs))
  (:skip :fasload (:compile read-table-compiler))
  (:rtc-compile-load read-table (:fasload read-table-compiler)))
```

Assume that there is a complex transformation :rtc-compile-load, which is like :compile-load except that is is built on a transformation called something like :rtc-compile, which uses the read table compiler rather than the Lisp compiler. In the above system, then, if the :rtc-compile transformation is to be performed, the :fasload transformation must be done on read-table-compiler first, that is the read table compiler must be loaded if the read table is to be recompiled. If you say (make-system 'reader :compile), then the :compile transformation is done on the read-table-compiler module despite the :skip, compiling the read table compiler if need be. If you say (make-system 'reader), the reader and the read table are loaded, but the :skip keeps this from happening to the read table compiler.

So far nothing has been said about what can be given as a *condition* for a transformation except for the default functions, which check for conditions such as a source file being newer than the binary. In general, any function that takes the same arguments as the transformation function (e.g. compile-file) and returns t if the transformation needs to be performed, can be in this place as a symbol, including for example a closure. To take an example, suppose there is a file that contains compile-flavor-methods for a system and that should therefore be recompiled if any of the flavor method definitions change. In this case, the condition function for compiling that file should return t if either the source of that file itself or any of the files that define the flavors have changed. This is what the :compile-load-init complex transformation is for. It is defined like this:

```
(defmacro (:compile-load-init si:defsystem-macro)
               (input add-dep &optional com-dep load-dep
                &aux function)
  (setq function (let-closed ((*additional-dependent-modules*
                               add-dep))
                  'compile-load-init-condition))
  `(:fasload (:compile ,input ,com-dep ,function) ,load-dep))

(defun compile-load-init-condition (source-file qfasl-file)
  (or (si:file-newer-than-file-p source-file qfasl-file)
      (local-declare ((special *additional-dependent-modules*))
        (si:other-files-newer-than-file-p
                        *additional-dependent-modules*
                        qfasl-file))))
```

The condition function generated when this macro is used returns t either if si:file-newer-than-file-p would with those arguments, or if any of the other files in add-dep, which presumably is a *module specification*, are newer than the QFASL file. Thus the file (or module) to which the :compile-load-init transformation applies will be compiled if it or any of the source files it depends on has been changed, and will be loaded under the normal conditions. In most (but not all cases), com-dep is a :fasload transformation of the same files as add-dep specifies, so that all the files this one depends on will be loaded before compiling it.

28.7 Common Lisp Modules

In Common Lisp, a *module* is a name given to a group of files of code. Modules are not like systems because nothing records what the "contents" of any particular module may be. Instead, one of the files which defines the module contains a provide form which says, when that file is loaded, "Module foo is now present." Other files may say, using require, "I want to use module foo."

Normally the require form also specifies the files to load if foo has not been provide'd already. This is where the information of which files are in a module is stored. If the require does not have file names in it, the module name foo is used in an implementation-dependent manner to find files to load. The Lisp Machine does this by using it as a system name in **make-system**.

provide *module-name*
Adds *module-name* to the list ***modules*** of modules already loaded. *module-name* should be a string; case is significant.

require *module-name* &rest *files*
If module *module-name* is not already loaded (on ***modules***), *files* are loaded in order to make the module available. *module-name* should be a string; case is significant. The elements of *files* should be pathnames or namestrings. If *files* is **nil**, **(make-system** *module-name* **:noconfirm)** is done. Note, however, that case is not significant in the argument to **make-system**.

modules *Variable*
A list of names (strings) of all modules **provide**'d so far.

28.8 The Patch Facility

The patch facility allows a system maintainer to manage new releases of a large system and issue patches to correct bugs. It is designed to be used to maintain both the Lisp Machine system itself and applications systems that are large enough to be loaded up and saved on a disk partition.

When a system of programs is very large, it needs to be maintained. Often problems are found and need to be fixed, or other little changes need to be made. However, it takes a long time to load up all of the files that make up such a system, and so rather than having every user load up all the files every time he wants to use the system, usually the files just get loaded once into a Lisp world, which is then saved away on a disk partition. Users then use this disk partition, copies of which may appear on many machines. The problem is that since the users don't load up the system every time they want to use it, they don't get all the latest changes.

The purpose of the patch system is to solve this problem. A *patch* file is a little file that, when you load it, updates the old version of the system into the new version of the system. Most often, patch files just contain new function definitions; old functions are redefined to do their new thing. When you want to use a system, you first use the Lisp environment saved on the disk, and then you load all the latest patches. Patch files are very small, so loading them doesn't take much time. You can even load the saved environment, load up the latest patches, and then save it away, to save future users the trouble of even loading the patches. (Of course, new patches may be made later, and then these will have to be loaded if you want to get the very latest version.)

For every system, there is a series of patches that have been made to that system. To get the latest version of the system, you load each patch file in the series, in order. Sooner or later, the maintainer of a system wants to stop building more and more patches, and recompile everything, starting afresh. A complete recompilation is also necessary when a system is changed in a far-reaching way, that can't be done with a small patch; for example, if you completely reorganize a program, or change a lot of names or conventions, you might need to completely recompile it to make it work again. After a complete recompilation has been done, the old patch files are no longer suitable to use; loading them in might even break things.

The way all this is kept track of is by labelling each version of a system with a two-part number. The two parts are called the *major version number* and the *minor version number*. The minor version number is increased every time a new patch is made; the patch is identified by the major and minor version number together. The major version number is increased when the program is completely recompiled, and at that time the minor version number is reset to zero. A complete system version is identified by the major version number, followed by a dot, followed by the minor version number. Thus, patch 93.9 is for major version 93 and minor version 9; it is followed by patch 93.10.

To clarify this, here is a typical scenario. A new system is created; its initial version number is 1.0. Then a patch file is created; the version of the program that results from loading the first patch file into version 1.0 is called 1.1. Then another patch file might be created, and loading that patch file into system 1.1 creates version 1.2. Then the entire system is recompiled, creating version 2.0 from scratch. Now the two patch files are irrelevant, because they fix old software; the changes that they reflect are integrated into system 2.0.

Note that the second patch file should only be loaded into system 1.1 in order to create system 1.2; you shouldn't load it into 1.0 or any other system besides 1.1. It is important that all the patch files be loaded in the proper order, for two reasons. First, it is very useful that any system numbered 1.1 be exactly the same software as any other system numbered 1.1, so that if somebody reports a bug in version 1.1, it is clear just which software is being complained about. Secondly, one patch might patch another patch; loading them in some other order might have the wrong effect.

The patch facility keeps track of all the patch files that exist, remembering which version each one creates. There is a separate numbered sequence of patch files for each major version of each system. All of them are stored in the file system, and the patch facility keeps track of where they all are. In addition to the patch files themselves, there are *patch directory* files that contain the patch facility's data base by which it keeps track of what minor versions exist for a major version, and what the last major version of a system is. These files and how to make them are described below.

In order to use the patch facility, you must define your system with defsystem (see chapter 28, page 660) and declare it as patchable with the :patchable option. When you load your system (with make-system, see page 666), it is added to the list of all systems present in the world. The patch facility keeps track of which version of each patchable system is present and where the data about that system reside in the file system. This information can be used to update the Lisp world automatically to the latest versions of all the systems it contains. Once a system is present, you can ask for the latest patches to be loaded, ask which patches are already loaded, and add new patches.

You can also load in patches or whole new systems and then save the entire Lisp environment away in a disk partition. This is explained on section 35.11, page 804.

When a Lisp Machine is booted, it prints out a line of information for each patchable system present in the booted Lisp world, saying which major and minor versions are loaded. This is done by print-herald (see page 674).

print-system-modifications &rest *system-names*

With no arguments, this lists all the systems present in this world and, for each system, all the patches that have been loaded into this world. For each patch it shows the major version number (which is always the same since a world can only contain one major version), the minor version number, and an explanation of what the patch does, as typed in by the person who made the patch.

If print-system-modifications is called with arguments, only the modifications to the systems named are listed.

print-herald &optional *format-dest*

Prints the names and loaded version numbers of all patchable systems loaded, and the microcode. Also printed are the number of the band you booted, the amount of physical and virtual memory you have, the host name of the machine, and its associated machine name. Example:

```
MIT System, band 7 of CADR-1.
640K physical memory, 16127K virtual memory.
 System        98.43
 CADR          3.6
 ZMail         53.10
 MIT-Specific  22.0
 Microcode     309
MIT Lisp Machine One, with associated machine OZ.
```

format-dest defaults to t; if it is nil the answer is returned as a string rather than printed out. *format-dest* can also be a stream to print on.

si:get-system-version &optional *system*

Returns two values, the major and minor version numbers of the version of *system* currently loaded into the machine, or nil if that system is not present. *system* defaults to "System".

si:system-version-info &optional (*brief-p* nil)

Returns a string giving information about which systems and what versions of the systems are loaded into the machine, and what microcode version is running. A typical string for it to produce is:

```
"System 98.48, CADR 3.6, MIT-Specific 22.0, microcode 309"
```

If *brief-p* is t, it uses short names, suppresses the microcode version, any systems which should not appear in the disk label comment, the name System, and the commas:

```
"98.48"
```

28.8.1 Defining a System

In order to use the patch facility, you must declare your system as patchable by giving the :patchable option to defsystem (see chapter 28, page 660). The major version of your system in the file system is incremented whenever make-system is used to compile it. Thus a major version is associated with a set of QFASL files. The major version of your system that is remembered as having been loaded into the Lisp environment is set to the major version in the file system whenever make-system is used to load your system and the major version in the file system is greater than what you had loaded before.

After loading your system, you can save it with the disk-save function (see page 807). disk-save asks you for any additional information you want printed as part of the greeting when the machine is booted. This is in addition to the names and versions of all the systems present in this world. If the system version does not fit in the partition comment field allocated in the disk label, disk-save asks you to type in an abbreviated form.

28.8.2 Loading Patches

load-patches &rest *options*

This function is used to bring the current world up to the latest minor version of whichever major version it is, for all systems present, or for certain specified systems. If there are any patches available, load-patches offers to read them in. With no arguments, load-patches updates all the systems present in this world. If you do not specify the systems to operate on, load-patches also reloads the site files if they have changed (section 35.12, page 810), and reloads the files defining logical host translations if they have changed (page 574).

options is a list of keywords. Some keywords are followed by an argument. The following options are accepted:

:systems *list*	*list* is a list of names of systems to be brought up to date. If this option is not specified, all patchable systems loaded are processed.
:unreleased	Loads unreleased patches with no special querying. These patches should be loaded for experimental use if you wish the benefit of the latest bug fixes, but should not be loaded if you plan to save a band.
:site	Loads the latest site files if they have been changed since last loaded. This is the default if you do not specify explicitly which systems to process.
:nosite	Prevents loading of site files. This is the default when you specify the systems to process.
:hosts	Reloads the files defining logical host translations if they have been changed since last loaded. This is the default if you do not specify explicitly which systems to process.
:nohosts	Prevents loading of logical host translation files. This is the default when you specify the systems to process.

:verbose	Prints an explanation of what is being done. This is the default.
:selective	For each patch, says what it is and then ask the user whether or not to load it. This is the default. If the user answers P (for 'Proceed'), selective mode is turned off for any remaining patches to the current system.
:noselective	Turns off :selective.
:silent	Turns off both :selective and :verbose. In :silent mode all necessary patches are loaded without printing anything and without querying the user.
:force-unfinished	Loads patches that have not been finished yet, if they have been compiled.

load-patches returns t if any patches were loaded.

When you load a patchable system with make-system, load-patches is called automatically on that system.

si:patch-loaded-p *major-version minor-version* &optional (*system-name* "SYSTEM")
Returns t if the changes in patch *major-version.minor-version* of system *system-name* are loaded. If *major-version* is the major version of that system which is currently loaded, then the changes in that patch are loaded if the current minor version is greater than or equal to *minor-version*. If the currently loaded major version is greater than *major-version* then it is assumed that the newer system version contains all the improvements patched into earlier versions, so the value is t.

28.8.3 Making Patches

There are two editor commands that are used to create patch files. During a typical maintenance session on a system you will make several edits to its source files. The patch system can be used to copy these edits into a patch file so that they can be automatically incorporated into the system to create a new minor version. Edits in a patch file can be modified function definitions, new functions, modified defvar's and defconst's, or arbitrary forms to be evaluated, even including load's of new files.

The first step in making a patch is to *start* it. At this stage you must specify which patchable system you are making a patch for. Then you *add* one or more pieces of code from other source files to the patch. Finally you *finish* the patch. This is when you fill in the description of what the patch does; this description is what load-patches prints when it offers to load the patch. If you have any doubts about whether the patch will load and work properly, you finish it *unreleased*; then you can load it to test it but no bands can be saved containing the patch until you explicitly release it later.

It is important that any change you patch should go in a patch for the patchable system to which the changed source file belongs. This makes sure that nobody loads the change into a Lisp world which does not contain the file you were changing—something that might cause trouble.

Also, it ensures that you never patch changes to the same piece of code in two different patchable systems' patches. This would lead to disaster because there is no constraint on the order in which patches to two different systems are loaded.

Starting a patch can be done with Meta-X Start Patch. It reads the name of the system to patch with the minibuffer. Meta-X Add Patch can also start a patch, so an explicit Meta-X Start Patch is needed only infrequently.

Meta-X Add Patch adds the region (if there is one) or the current "defun" to the patch file currently being constructed. If you change a function, you should recompile it, test it, then once it works use Add Patch to put it in the patch file. If no patch is being constructed, one is started for you; you must type in the name of the system to patch.

A convenient way to add all your changes to a patch file is to use Meta-X Add Patch Changed Sections or Meta-X Add Patch Buffer Changed Sections. These commands ask you, for each changed function (or each changed function in the current buffer), whether to add it to the patch being constructed. If you use these commands more than once, a function which has been added to the patch and has not been changed since is considered "unchanged".

The patch file being constructed is in an ordinary editor buffer. If you mistakenly Add Patch something that doesn't work, you can select the buffer containing the patch file and delete it. Then later you can Add Patch the corrected version.

While you are making your patch file, the minor version number that has been allocated for you is reserved so that nobody else can use it. This way if two people are patching a system at the same time, they do not both get the same minor version number.

After testing and patching all of your changes, use Meta-X Finish Patch to install the patch file so that other users can load it. This compiles the patch file if you have not done so yourself (patches are always compiled). It also asks you for a comment describing the reason for the patch; load-patches and print-system-modifications print these comments. If the patch is complex or it has a good chance of causing new problems, you should not use Meta-X Finish Patch; instead, you should make an unreleased patch.

A finished patch can be *released* or *unreleased*. If a patch is unreleased, it can be loaded in the usual manner if the user says 'yes' to a special query, but once it has been loaded the user will be strongly discouraged from saving a band. Therefore, you still have a chance to edit the patch file and recompile it if there is something wrong with it. You can be sure that the old broken patch will not remain permanently in saved bands.

To finish a patch without releasing it, use the command Meta-X Finish Patch Unreleased. Then the patch can be tested by loading it. After a sufficient period for testing, you can release the patch with Meta-X Release Patch. If you discover a bug in the patch after this point, it is not sufficient to correct it in this patch file; you must put the fix in a new patch to correct any bands already saved with the broken version of this patch.

It is a good principle not to add any new features or fix any additional bugs in a patch once that patch is released; change it only to correct problems with that patch. New fixes to other bugs should go in new patches.

You can only be constructing one patch at any time. Meta-X Add Patch automatically adds to the patch you are constructing. But you can start constructing a different patch without finishing the first. If you use the command Meta-X Start Patch while constructing a patch, you are given the option of starting a new patch. The old patch ceases to be the one you are constructing but the patch file remains in its editor buffer. Later, or in another session, you can go back to constructing the first patch with the command Meta-X Resume Patch. This commands asks for both a patchable system name and the patch version to resume constructing. You can simply save the editor buffer of a patch file and resume constructing that patch in a later session. You can even resume constructing a finished patch; though it rarely makes sense to do this unless the patch is unreleased.

If you start to make a patch and change your mind, use the command Meta-X Cancel Patch. This deletes the record that says that this patch is being worked on. It also tells the editor that you are no longer constructing any patch. You can undo a finished (but unreleased) patch by using Resume Patch and then Cancel Patch. If a patch is released, you cannot remove it from saved bands, so it is not reasonable to cancel it at that stage.

28.8.4 Private Patches

A private patch is a file of changes which is not installed to be loaded automatically in sequence by all users. It is loaded only by explicit request (using the function load). A private patch is not associated with any particular patchable system, and has no version number.

To make a private patch, use the editor command Meta-X Start Private Patch. Instead of a patchable system name, you must specify a filename to use for the patch file; since the patch is not to be installed, there is no standard naming convention for it to follow. Add text to the patch using Meta-X Add Patch and finish it using Meta-X Finish Patch. There is no concept of release for private patches so there is no point in using Meta-X Finish Patch Unreleased. There is also no data base recording all private patches, so Meta-X Start Private Patch will resume an existing patch, or even a finished patch. In fact, finishing a private patch is merely a way to write a comment into it and compile it.

Once the private patch file is made, you can load it like any other file.

The private patch facility is just an easy way to copy code from various files into one new file with Patch-File: T in its attribute list (to prevent warnings about redefining functions defined in other files) and compile that file.

28.8.5 System Status

The patch system has the concept of the *status* of a major version of a system. A status keyword is recorded in the Lisp world for each patchable system that is loaded. There is also a current status for each major version of each system, recorded in the patch directory file for that major version. Loading patches updates the status in the Lisp world to match the current status stored in the patch directory. The status in the patch directory is changed with **si:set-system-status**.

The status is displayed when the system version is displayed, in places such as the system greeting message (**print-herald**) and the disk partition comment.

The status is one of the following keywords:

:experimental	The system has been built but has not yet been fully debugged and released to users. This is the default status when a new major version is created, unless it is overridden with the **:initial-status** option to **defsystem**.
:released	The system is released for general use. This status produces no extra text in the system greeting and the disk partition comment.
:obsolete	The system is no longer supported.
:broken	This is like **:experimental**, but is used when the system was thought incorrectly to have been debugged, and hence was **:released** for a while.
:inconsistent	Unreleased patches to this system have been loaded. If any patchable system is in this status, **disk-save** demands extra confirmation, and the resulting saved band is identified as "Bad" in its disk partition comment.

si:set-system-status *system status* &optional *major-version*

Changes the current status of a system, as recorded in the patch directory file. *system* is the name of the system. *major-version* is the number of the major version to be changed; if unsupplied it defaults to the version currently loaded into the Lisp world. *status* should be one of the keywords above.

Do not set the current system status to **:inconsistent**. A status of **:inconsistent** is set up in the Lisp world when an unreleased patch is loaded, and once set that way it never changes in that Lisp world. The status recorded in the system's patch directory file should describe the situation where all currently released patches are loaded. It should never be **:inconsistent**.

28.8.6 Patch Files

The patch system maintains several different types of files in the directory associated with your system. This directory is specified to **defsystem** via either the :patchable option or the :pathname-default option. These files are maintained automatically, but they are described here so that you can know what they are and when they are obsolete and can be deleted.

If the :patchable option to **defsystem** had no argument, then the patch data files are stored on the host, device and directory specified as the system's pathname default. The names and types of the filenames are all standard and do not include the name of the system in any way.

If the :patchable option to **defsystem** is given an argument, this argument is a file namestring specifying the host, device and directory to use for storing the patch data files. In addition, the system's short name is used in constructing the names of the files. This allows you to store the patch data files for several systems in the same directory.

There are three kinds of files that record patch information:

* the system patch directory

  This file records the current major version number, so that when the system is recompiled a new number can be allocated.

  On Tops-20, this file has, by default, a name like OZ:PS:<MYDIR>PATCH.DIRECTORY, where the host, device, and directory (OZ:PS:<MYDIR>) come from the system's :pathname-default as explained above.

  If :patchable is given an argument, this file for system FOO has a name like OZ:PS:<PATDIR>FOO.PATCH-DIRECTORY, where the host, device and directory come from :patchable's argument.

* the patch directory of a major version

  There is a file of this kind for each major version of the system. It records the patches that have been made for that major version: the minor version, author, description and release status of each one.

  The data in this file are in the form of a printed representation of a Lisp list with two elements. The first is the system status of this major version (:experimental, :released, :broken or :obsolete). The second is another list with an element for each patch. The element for a patch is a list of length four: the minor version, the patch description (a string) or nil for an unfinished patch, the author's name (a string), and a flag that is t if the patch is unreleased.

  On a Tops-20, for major version 259, this file has, by default, a name like OZ:PS:<MYDIR>PATCH-259.DIRECTORY.

  If :patchable is given an argument, this file for system FOO has a name like OZ:PS:<PATDIR>FOO-259.PATCH-DIRECTORY.

* the individual patch

 For each patch made, there is a Lisp source file and a QFASL file.

 On a Tops-20, for version 259.12, these files have, by default, names like OZ:PS:<MYDIR>PATCH-259-12.LISP and OZ:PS:<MYDIR>PATCH-259-12.QFASL.

 If :patchable is given an argument, this file for system FOO has a name like OZ:PS:<PATDIR>FOO-259-12.PATCH-DIRECTORY.

On certain types of file systems, slightly different naming conventions are used to keep the names short enough to be legal.

29. Processes

The Lisp Machine supports *multi-processing*; several computations can be executed concurrently by placing each in a separate *process*. A process is like a processor, simulated by software. Each process has its own program counter, its own stack of function calls and its own special-variable binding environment in which to execute its computation. (This is implemented with stack groups; see chapter 13, page 256.)

If all the processes are simply trying to compute, the machine allows them all to run an equal share of the time. This is not a particularly efficient mode of operation since dividing the finite memory and processor power of the machine among several processes certainly cannot increase the available power and in fact wastes some of it in overhead. The typical use for processes is that at any time only one or two are trying to run. The rest are either *waiting* for some event to occur or *stopped* and not allowed to compete for resources.

A process waits for an event by means of the process-wait primitive, which is given a predicate function which defines the event being waited for. A module of the system called the process scheduler periodically calls that function. If it returns nil the process continues to wait; if it returns t the process is made runnable and its call to process-wait returns, allowing the computation to proceed.

A process may be *active* or *stopped*. Stopped processes are never allowed to run; they are not considered by the scheduler, and so can never become the current process until they are made active again. The scheduler continually tests the waiting functions of all the active processes, and those which return non-nil values are allowed to run. When you first create a process with make-process, it is inactive.

The activity of a process is controlled by two sets of Lisp objects associated with it, called its *run reasons* and its *arrest reasons*. These sets are implemented as lists. Any kind of object can be in these sets; typically keyword symbols and active objects such as windows and other processes are found. A process is considered active when it has at least one run reason and no arrest reasons.

To get a computation to happen in another process, you must first create a process, then say what computation you want to happen in that process. The computation to be executed by a process is specified as an *initial function* and a list of arguments to that function. When the process starts up it applies the function to the arguments. In some cases the initial function is written so that it never returns, while in other cases it performs a certain computation and then returns, which stops the process.

To *reset* a process means to exit its entire computation nonlocally using *unwind-stack (see page 82). Some processes are temporary and die when reset. The other, permanent functions start their computations over again when reset. Resetting a process clears its waiting condition, so that if it is active it becomes runnable. To *preset* a function is to set up its initial function (and arguments) and then reset it. This is how you start up a computation in a process.

All processes in a Lisp Machine run in the same virtual address space, sharing the same set of Lisp objects. Unlike other systems that have special restricted mechanisms for inter-process communication, the Lisp Machine allows processes to communicate in arbitrary ways through shared Lisp objects. One process can inform another of an event simply by changing the value of a global variable. Buffers containing messages from one process to another can be implemented as lists or arrays. The usual mechanisms of atomic operations, critical sections, and interlocks are provided (see store-conditional [page 688], without-interrupts [page 684], and process-lock [page 687]).

A process is a Lisp object, an instance of one of several flavors of process (see chapter 21, page 401). The remainder of this chapter describes the operations defined on processes, the functions you can apply to a process, and the functions and variables a program running in a process can use to manipulate its process.

29.1 The Scheduler

At any time there is a set of *active processes*; as described above, these are all the processes that are not stopped. Each active process is either currently running, runnable (ready to run), or waiting for some condition to become true. The active processes are managed by a special stack group called the *scheduler*, which repeatedly examines each active process to determine whether it is waiting or ready to run. The scheduler then selects one process and starts it up.

The process chosen by the scheduler becomes the *current process*, that is, the one process that is running on the machine. The scheduler sets the variable current-process to it. It remains the current process and continues to run until either it decides to wait, or a *sequence break* occurs. In either case, the scheduler stack group is resumed. It then updates the process's run time meters and chooses a new process to run next. This way, each process that is ready to run gets its share of time in which to execute.

Each process has a *priority* which is a number. Most processes have priority zero. Larger numbers give a process more priority. The scheduler only considers the highest priority runnable processes, so if there is one runnable process with priority 20 then no process with lesser priority can run.

The scheduler determines whether a process is runnable by applying the process's *wait-function* to its *wait-argument-list*. If the wait-function returns a non-nil value, then the process is ready to run; otherwise, it is waiting.

A process can wait for some condition to become true by calling process-wait (see page 685). This function sets the process's wait-function and wait-argument-list as specified by the caller, and resumes the scheduler stack group. A process can also wait for just a moment by calling process-allow-schedule (see page 686), which resumes the scheduler stack group but leaves the process runnable; it will run again as soon as all other runnable processes have had a chance.

A sequence break is a kind of interrupt that is generated by the Lisp system for any of a variety of reasons; when it occurs, the scheduler is resumed. The function si:sb-on (see page 687) can be used to control when sequence breaks occur. The default is to sequence break once a second. Thus even if a process never waits and is not stopped, it is forced to return control to the scheduler once a second so that any other runnable processes can get their turn.

The system does not generate a sequence break when a page fault occurs; thus time spent waiting for a page to come in from the disk is "charged" to a process the same as time spent computing, and cannot be used by other processes. It is done this way for the sake of simplicity; this allows the whole implementation of the process system to reside in ordinary virtual memory, so that it does not have to worry specially about paging. The performance penalty is small since Lisp Machines are personal computers, not multiplexed among a large number of processes. Usually only one process at a time is runnable.

A process's wait function is free to touch any data structure it likes and to perform any computation it likes. Of course, wait functions should be kept simple, using only a small amount of time and touching only a small number of pages, or system performance will be impacted since the wait function will consume resources even when its process is not running.

If a wait function gets an error, the error occurs inside the scheduler. If this enters the debugger, all scheduling comes to a halt until the user proceeds or aborts. Aborting in the debugger inside the scheduler "blasts" the current process by giving it a trivial wait function that always returns **nil**; this prevents recurrence of the same problem. It is best to write wait functions that cannot get errors, by keeping them simple and by arranging for any problems to be detected before the scheduler sees the wait function. **process-wait** calls the wait function once before giving it to the scheduler, and this often exposes an error before it can interfere with scheduling.

Note well that a process's wait function is executed inside the scheduler stack-group, *not* inside the process. This means that a wait function may not access special variables bound in the process. It is allowed to access global variables. It can access variables bound by a process through the closure mechanism (chapter 12, page 250). If the wait function is defined lexically within the caller of **process-wait** then it can access local variables through the lexical scoping mechanism. Most commonly any values needed by the wait function are passed to it as arguments.

current-process *Variable*

The value of **current-process** is the process that is currently executing, or **nil** while the scheduler is running. When the scheduler calls a process's wait-function, it binds **current-process** to the process so that the wait-function can access its process.

without-interrupts *body...* *Macro*

The *body* forms are evaluated with **inhibit-scheduling-flag** bound to **t**. This is the recommended way to lock out multi-processing over a small critical section of code to prevent timing errors. In other words the body is an *atomic operation*. The values of the last form in the body are ultimately returned.

In this example, **list** is presumed to be a global variable referred to from two places in the code which different processes will execute.

```
(without-interrupts
  (push item list))

(without-interrupts
  (cond ((memq item list)
         (setq list (delq item list))
         t)
        (t nil)))
```

inhibit-scheduling-flag *Variable*

The value of inhibit-scheduling-flag is normally nil. without-interrupts binds it to t, which prevents process-switching until inhibit-scheduling-flag becomes nil again. It is cleaner to use without-interrupts than to refer directly to this variable.

process-wait *whostate function* &rest *arguments*

This is the primitive for waiting. The current process waits until the application of *function* to *arguments* returns non-nil (at which time process-wait returns). Note that *function* is applied in the environment of the scheduler, not the environment of the process-wait, so special bindings in effect when process-wait was called are *not* be in effect when *function* is applied. Be careful when using any free references in *function*. *whostate* is a string containing a brief description of the reason for waiting. If the who-line at the bottom of the screen is looking at this process, it will show *whostate*.

Example:

```
(process-wait "Buffer"
              #'(lambda (b) (not (zerop (buffer-n-things b))))
              the-buffer)
```

process-sleep *interval*

Waits for *interval* sixtieths of a second, and then returns. It uses process-wait.

sleep *seconds*

Waits *seconds* seconds and then returns. *seconds* need not be an integer. This also uses process-wait.

process-wait-with-timeout *whostate interval function* &rest *arguments*

This is like process-wait except that if *interval* sixtieths of a second go by and the application of *function* to *arguments* is still returning nil, then process-wait-with-timeout returns anyway. The value returned is the value of applying *function* to *arguments*; thus, it is non-nil if the wait condition actually occurred, nil for a time-out.

If *interval* is nil, there is no timeout, and this function is then equivalent to process-wait.

with-timeout (*interval timeout-forms...*) *body...* *Macro*

body is executed with a timeout in effect for *interval* sixtieths of a second. If *body* finishes before that much time elapses, the values of the last form in *body* are returned.

If after *interval* has elapsed *body* has not completed, its execution is terminated with a throw caught by the with-timeout form. Then the *timeout-forms* are evaluated and the values of the last one of them are returned.

For example,

```
(with-timeout ((* 60. 60.) (format *query-io* " ... Yes.") t)
  (y-or-n-p "Really do it? (Yes after one minute) "))
```

is a convenient way to ask a question and assume an answer if the user does not respond promptly. This is a good thing to do for queries likely to occur when the user has walked away from the terminal and expects an operation to finish without his attention.

process-allow-schedule

Resumes the scheduler momentarily; all other processes will get a chance to run before the current process runs again.

sys:scheduler-stack-group *Constant*

This is the stack group in which the scheduler executes.

sys:clock-function-list *Variable*

This is a list of functions to be called by the scheduler 60 times a second. Each function is passed one argument, the number of 60ths of a second since the last time that the functions on this list were called. These functions implement various system overhead operations such as blinking the blinking cursor on the screen. Note that these functions are called inside the scheduler, just as are the functions of simple processes (see page 695). The scheduler calls these functions as often as possible, but never more often than 60 times a second. That is, if there are no processes ready to run, the scheduler calls the clock functions 60 times a second, assuming that, all together, they take less than 1/60 second to run. If there are processes continually ready to run, then the scheduler calls the clock functions as often as it can; usually this is once a second, since usually the scheduler gets control only once a second.

sys:active-processes *Variable*

This is the scheduler's data-structure. It is a list of lists, where the car of each element is an active process or nil and the cdr is information about that process.

sys:all-processes *Variable*

This is a list of all the processes in existence. It is mainly for debugging.

si:initial-process *Constant*

This is the process in which the system starts up when it is booted.

si:sb-on &optional *when*

Controls what events cause a sequence break, i.e. when rescheduling occurs. The following keywords are names of events which can cause a sequence break.

:clock	This event happens periodically based on a clock. The default period is one second. See sys:%tv-clock-rate, page 293.
:keyboard	Happens when a character is received from the keyboard.
:chaos	Happens when a packet is received from the Chaosnet, or transmission of a packet to the Chaosnet is completed.

Since the keyboard and Chaosnet are heavily buffered, there is no particular advantage to enabling the :keyboard and :chaos events, unless the :clock event is disabled.

With no argument, si:sb-on returns a list of keywords for the currently enabled events.

With an argument, the set of enabled events is changed. The argument can be a keyword, a list of keywords, nil (which disables sequence breaks entirely since it is the empty list), or a number, which is the internal mask, not documented here.

29.2 Locks

A *lock* is a software construct used for synchronization of two processes. A lock is either held by some process, or is free. When a process tries to seize a lock, it waits until the lock is free, and then it becomes the process holding the lock. When it is finished, it unlocks the lock, allowing some other process to seize it. A lock protects some resource or data structure so that only one process at a time can use it.

In the Lisp Machine, a lock is a locative pointer to a cell. If the lock is free, the cell contains nil; otherwise it contains the process that holds the lock. The process-lock and process-unlock functions are written in such a way as to guarantee that two processes can never both think that they hold a certain lock; only one process can ever hold a lock at one time.

process-lock *locative* &optional (*lock-value* current-process) (*whostate* "Lock") *timeout*

This is used to seize the lock that *locative* points to. If necessary, process-lock waits until the lock becomes free. When process-lock returns, the lock has been seized. *lock-value* is the object to store into the cell specified by *locative*, and *whostate* is passed on to process-wait.

If *timeout* is non-nil, it should be a fixnum representing a time interval in 60ths of a second. If it is necessary to wait more than that long, an error with condition name sys:lock-timeout is signaled.

process-unlock *locative* &optional (*lock-value* current-process)

This is used to unlock the lock that *locative* points to. If the lock is free or was locked by some other process, an error is signaled. Otherwise the lock is unlocked. *lock-value* must have the same value as the *lock-value* parameter to the matching call to process-lock, or else an error is signaled.

sys:lock-timeout (error) *Condition*

This condition is signaled when **process-lock** waits longer than the specified timeout.

It is a good idea to use **unwind-protect** to make sure that you unlock any lock that you seize. For example, if you write

```
(unwind-protect
   (progn (process-lock lock-3)
          (function-1)
          (function-2))
   (process-unlock lock-3))
```

then even if **function-1** or **function-2** does a **throw**, **lock-3** will get unlocked correctly. Particular programs that use locks often define special forms that package this **unwind-protect** up into a convenient stylistic device.

A higher level locking construct is **with-lock**:

with-lock (*lock* &key *norecursive*) *body...* *Macro*

Executes the *body* with *lock* locked. *lock* should actually be an expression whose value would be the status of the lock; it is used inside **locf** to get a locative pointer with which the locking and unlocking are done.

It is OK for one process to lock a lock multiple times, recursively, using **with-lock**, provided *norecursive* is not **nil**.

norecursive should be literally **t** or **nil**; it is not evaluated. If it is **t**, this call to **with-lock** signals an error if the lock is already locked by the running process.

A lower level construct which can be used to implement atomic operations, and is used in the implementation of **process-lock**, is **store-conditional**.

store-conditional *location oldvalue newvalue*

This stores *newvalue* into *location* iff *location* currently contains *oldvalue*. The value is **t** iff the cell was changed.

If *location* is a list, the cdr of the list is tested and stored in. This is in accord with the general principle of how to access the contents of a locative properly, and makes **(store-conditional (locf (cdr x)) ...)** work.

An even lower-level construct is the subprimitive **%store-conditional**, which is like **store-conditional** with no error checking, but is faster.

29.2.1 Process Queues

A process queue is a kind of lock which can record several processes which are waiting for the lock and grant them the lock in the order that they requested it. The queue has a fixed size. If the number of processes waiting remains less than that size then they will all get the lock in the order of requests. If too many processes are waiting then the order of requesting is not remembered for the extra ones, but proper interlocking is still maintained.

si:make-process-queue *name size*
Makes and returns a process queue object named *name*, able to record *size* processes. The count of *size* includes the process that owns the lock.

si:process-enqueue *process-queue* &optional *lock-value who-state*
Attempts to lock *process-queue* on behalf of *lock-value*. If *lock-value* is nil then the locking is done on behalf of current-process.

If the queue is locked, then *lock-value* or the current process is put on the queue. Then this function waits for that lock value to reach the front of the queue. When it does so, the lock has been granted, and the function returns. The lock is now locked in the name of *lock-value* or the current process, until si:process-dequeue is used to unlock it.

who-state appears in the who line during the wait. It defaults to "Lock".

si:process-deqeueue *process-queue* &optional *lock-value*
Unlocks process-queue. *lock-value* (which defaults to the current process) must be the value which now owns the lock on the queue, or an error occurs. The next process or other object on the queue is granted the lock and its call to si:process-enqueue will therefore return.

si:reset-process-queue *process-queue*
Unlocks the queue and clears out the list of things waiting to lock it.

si:process-queue-locker *process-queue*
Returns the object in whose name the queue is currently locked, or nil if it is not now locked.

29.3 Creating a Process

There are two ways of creating a process. One is to create a permanent process which you will hold on to and manipulate as desired. The other way is to say simply, "call this function on these arguments in another process, and don't bother waiting for the result." In the latter case you never actually use the process itself as an object.

make-process *name* &key ...
Creates and returns a process named *name*. The process is not capable of running until it has been reset or preset in order to initialize the state of its computation.

Usually you do not need to specify any of the keyword arguments. The following keyword arguments are allowed:

:simple-p	Specifying **t** here gives you a simple process (see page 695).
:flavor	Specifies the flavor of process to be created. See section 29.5, page 695, for a list of all the flavors of process supplied by the system.
:stack-group	Specifies the stack group the process is to use. If this option is not specified, a stack group is created according to the relevant options below.
:warm-boot-action	What to do with the process when the machine is booted. See page 693.
:quantum	See page 692.
:priority	See page 693.
:run-reasons	Lets you supply an initial run reason. The default is **nil**.
:arrest-reasons	Lets you supply an initial arrest reason. The default is **nil**.
:sg-area	The area in which to create the stack group. The default is the value of **default-cons-area**.
:regular-pdl-area :special-pdl-area :regular-pdl-size :special-pdl-size	These are passed on to **make-stack-group**, page 259.
:swap-sv-on-call-out :swap-sv-of-sg-that-calls-me :trap-enable	Specify those attributes of the stack group. You don't want to use these.

If you specify **:flavor**, there can be additional options provided by that flavor.

The following functions allow you to call a function and have its execution happen asynchronously in another process. This can be used either as a simple way to start up a process that will run "forever", or as a way to make something happen without having to wait for it to complete. When the function returns, the process is returned to a pool of free processes for reuse. The only difference between these three functions is in what happens if the machine is booted while the process is still active.

Normally the function to be run should not do any I/O to the terminal, as it does not have a window and terminal I/O will cause it to make a notification and wait for user attention. Refer to section 13.5, page 264 for a discussion of the issues.

process-run-function *name-or-options function* &rest *args*

Creates a process, presets it to apply *function* to *args*, and starts it running. The value returned is the new process. The process is killed if *function* returns; by default, it is also killed if it is reset. Example:

```
(defun background-print (file)
  (process-run-function "Print" 'hardcopy-file file))
```

creates a background process that prints the specified file.

name-or-options can be either a string specifying a name for the process or a list of alternating keywords and values that can specify the name and various other parameters.

:name	This keyword should be followed by a string which specifies the name of the process. The default is "Anonymous".
:restart-after-reset	This keyword says what to do to the process if it is reset. nil means the process should be killed; anything else means the process should be restarted. nil is the default.
:warm-boot-action	What to do with the process when the machine is booted. See page 693.
:restart-after-boot	This is a simpler way of saying what to do with the process when the machine is booted. If the :warm-boot-action keyword is not supplied or its value is nil, then this keyword's value is used instead. nil means the process should be killed; anything else means the process should be restarted. nil is the default.
:quantum	See page 692.
:priority	See page 693.

process-run-restartable-function *name-or-keywords function* &rest *args*

This is the same as process-run-function except that the default is that the process will be restarted if reset or after a warm boot. You can get the same effect by using process-run-function with appropriate keywords.

29.4 Process Generic Operations

These are the operations that are defined on all flavors of process. Certain process flavors may define additional operations. Not all possible operations are listed here, only those of interest to the user.

29.4.1 Process Attributes

:name *Operation on* si:process

Returns the name of the process, which was the first argument to make-process or process-run-function when the process was created. The name is a string that appears in the printed-representation of the process, stands for the process in the who-line and the peek display, etc.

:stack-group *Operation on* si:process

Returns the stack group currently executing on behalf of this process. This can be different from the initial-stack-group if the process contains several stack groups that coroutine among themselves, or if the process is in the error-handler, which runs in its own stack group.

Note that the stack-group of a *simple* process (see page 695) is not a stack group at all, but a function.

:initial-stack-group *Operation on* si:process

Returns the stack group the initial-function is called in when the process starts up or is reset.

:initial-form *Operation on* si:process

Returns the initial "form" of the process. This isn't really a Lisp form; it is a cons whose car is the initial-function and whose cdr is the list of arguments to which that function is applied when the process starts up or is reset.

In a simple process (see page 695), the initial form is a list of one element, the process's function.

To change the initial form, use the :preset operation (see page 694).

:wait-function *Operation on* si:process

Returns the process's current wait-function, which is the predicate used by the scheduler to determine if the process is runnable. This is #'true if the process is running, and #'false if the process has no current computation (for instance, if it has just been created, its initial function has returned) or if the program has decided to wait indefinitely. The wait-function of a flushed process is si:flushed-process, a function equivalent to #'false but recognizably distinct.

:wait-argument-list *Operation on* si:process

Returns the arguments to the process's current wait-function. This is frequently the &rest argument to process-wait in the process's stack. The system always uses it in a safe manner, i.e. it forgets about it before process-wait returns.

:whostate *Operation on* si:process

Returns a string that is the state of the process to go in the who-line at the bottom of the screen. This is "run" if the process is running or trying to run; otherwise, it is the reason why the process is waiting. If the process is stopped, then this whostate string is ignored and the who-line displays arrest if the process is arrested or stop if the process has no run reasons.

:quantum *Operation on* si:process
:set-quantum *60ths* *Operation on* si:process

Respectively return and change the number of 60ths of a second this process is allowed to run without waiting before the scheduler will run someone else. The quantum defaults to 1 second.

:quantum-remaining *Operation on* si:process
Returns the amount of time remaining for this process to run, in 60ths of a second.

:priority *Operation on* si:process
:set-priority *priority-number* *Operation on* si:process
Respectively return and change the priority of this process. The larger the number, the more this process gets to run. Within a priority level the scheduler runs all runnable processes in a round-robin fashion. Regardless of priority a process will not run for more than its quantum. The default priority is 0, and no normal process uses other than 0.

:warm-boot-action *Operation on* si:process
:set-warm-boot-action *action* *Operation on* si:process
Respectively return and change the process's warm-boot-action, which controls what happens if the machine is booted while this process is active. (Contrary to the name, this applies to both cold and warm booting.) This can be nil or :flush, which means to *flush* the process (see page 695), or can be a function to call. The default is si:process-warm-boot-delayed-restart, which resets the process, causing it to start over at its initial function. You can also use si:process-warm-boot-reset, which throws out of the process' computation and kills the process, or si:process-warm-boot-restart, which is like the default but restarts the process at an earlier stage of system reinitialization. This is used for processes like the keyboard process and chaos background process, which are needed for reinitialization itself.

:simple-p *Operation on* si:process
Returns nil for a normal process, t for a simple process. See page 695.

:idle-time *Operation on* si:process
Returns the time in seconds since this process last ran, or nil if it has never run.

:total-run-time *Operation on* si:process
Returns the amount of time this process has run, in 60ths of a second. This includes cpu time and disk wait time.

:disk-wait-time *Operation on* si:process
Returns the amount of time this process has spent waiting for disk I/O, in 60ths of a second.

:cpu-time *Operation on* si:process
Returns the amount of time this process has spent actually executing instructions, in 60ths of a second.

:percent-utilization *Operation on* si:process
Returns the fraction of the machine's time this process has been using recently, as a percentage (a number between 0 and 100.0). The value is a weighted average giving more weight to more recent history.

:reset-meters *Operation on* si:process
Resets the run-time counters of the process to zero.

29.4.2 Run and Arrest Reasons

:run-reasons *Operation on* si:process
Returns the list of run reasons, which are the reasons why this process should be active (allowed to run).

:run-reason *object* *Operation on* si:process
Adds *object* to the process's run reasons. This can activate the process.

:revoke-run-reason *object* *Operation on* si:process
Removes *object* from the process's run reasons. This can stop the process.

:arrest-reasons *Operation on* si:process
Returns the list of arrest reasons, which are the reasons why this process should be inactive (forbidden to run).

:arrest-reason *object* *Operation on* si:process
Adds *object* to the process's arrest reasons. This can stop the process.

:revoke-arrest-reason *object* *Operation on* si:process
Removes *object* from the process's arrest reasons. This can activate the process.

:active-p *Operation on* si:process
:runnable-p *Operation on* si:process
These two operations are the same. t is returned if the process is active, i.e. it can run if its wait-function allows. nil is returned if the process is stopped.

29.4.3 Bashing the Process

:preset *function* &rest *args* *Operation on* si:process
Sets the process's initial function to *function* and initial arguments to *args*. The process is then reset so that any computation occuring in it is terminated and it begins anew by applying *function* to *args*. A :preset operation on a stopped process returns immediately, but does not activate the process; hence the process will not really apply *function* to *args* until it is activated later.

:reset &optional *no-unwind kill* *Operation on* si:process
Forces the process to throw out of its present computation and apply its initial function to its initial arguments, when it next runs. The throwing out is skipped if the process has no present computation (e.g. it was just created), or if the *no-unwind* option so specifies. The possible values for *no-unwind* are:

:unless-current
nil — Unwind unless the stack group to be unwound is the one currently executing, or belongs to the current process.

:always	Unwind in all cases. This may cause the operation to throw through its caller instead of returning.
t	Never unwind.

If *kill* is t, the process is to be killed after unwinding it. This is for internal use by the :kill operation only.

A :reset operation on a stopped process returns immediately, but does not activate the process; hence the process will not really get reset until it is activated later.

:flush *Operation on* si:process

Forces the process to wait forever. A process may not :flush itself. Flushing a process is different from stopping it, in that it is still active and hence if it is reset or preset it will start running again.

A flushed process can be recognized because its wait-function is si:flushed-process. If a process belonging to a window is flushed, exposing or selecting the window resets the process.

:kill *Operation on* si:process

Gets rid of the process. It is reset, stopped, and removed from sys:all-processes.

:interrupt *function* &rest *args* *Operation on* si:process

Forces the process to apply *function* to *args*. When *function* returns, the process continues the interrupted computation. If the process is waiting, it wakes up, calls *function*, then waits again when *function* returns.

If the process is stopped it does not apply *function* to *args* immediately, but will later when it is activated. Normally the :interrupt operation returns immediately, but if the process's stack group is in an unusual internal state :interrupt may have to wait for it to get out of that state.

29.5 Process Flavors

These are the flavors of process provided by the system. It is possible for users to define additional flavors of their own.

si:process *Flavor*

This is the standard default kind of process.

si:simple-process *Flavor*

A simple process is not a process in the conventional sense. It has no stack group of its own; instead of having a stack group that gets resumed when it is time for the process to run, it has a function that gets called when it is time for the process to run. When the wait-function of a simple process becomes true, and the scheduler notices it, the simple process's function is called in the scheduler's own stack group. Since a simple process does not have any stack group of its own, it can't save control state in between calls; any state that it saves must be saved in data structure.

The only advantage of simple processes over normal processes is that they use up less system overhead, since they can be scheduled without the cost of resuming stack-groups. They are intended as a special, efficient mechanism for certain purposes. For example, packets received from the Chaosnet are examined and distributed to the proper receiver by a simple process that wakes up whenever there are any packets in the input buffer. However, they are harder to use, because you can't save state information across scheduling. That is, when the simple process is ready to wait again, it must return; it can't call process-wait and continue to do something else later. In fact, it is an error to call process-wait from inside a simple process. Another drawback to simple processes is that if the function signals an error, the scheduler itself is broken and multiprocessing stops; this situation can be repaired only by aborting, which blasts the process. Also, when a simple process is run, no other process is scheduled until it chooses to return; so simple processes should never run for a long time without returning.

Asking for the stack group of a simple process does not signal an error, but returns the process's function instead.

Since a simple process cannot call process-wait, it needs some other way to specify its wait-function. To set the wait-function of a simple process, use si:set-process-wait (see below). So, when a simple process wants to wait for a condition, it should call si:set-process-wait to specify the condition, then return.

si:set-process-wait *simple-process wait-function wait-argument-list*
Sets the *wait-function* and *wait-argument-list* of *simple-process*. See the description of the si:simple-process flavor (above) for more information.

29.6 Other Process Functions

process-enable *process*
Activates *process* by revoking all its run and arrest reasons, then giving it a run reason of :enable.

process-reset-and-enable *process*
Resets *process*, then enables it.

process-disable *process*
Stops *process* by revoking all its run reasons. Also revokes all its arrest reasons.

The remaining functions in this section are obsolete, since they simply duplicate what can be done by sending a message. They are documented here because their names are in the global package.

process-preset *process function* &rest *args*
Sends a :preset message.

process-reset *process*
Sends a :reset message.

process-name *process*
Gets the name of a process, like the :name operation.

process-stack-group *process*
Gets the current stack group of a process, like the :stack-group operation.

process-initial-stack-group *process*
Gets the initial stack group of a process, like the :initial-stack-group operation.

process-initial-form *process*
Gets the initial form of a process, like the :initial-form operation.

process-wait-function *process*
Gets the current wait-function of a process, like the :wait-function operation.

process-wait-argument-list *p*
Gets the arguments to the current wait-function of a process, like the :wait-argument-list operation.

process-whostate *p*
Gets the current who-line state string of a process, like the :whostate operation.

30. Errors and Debugging

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

The second explains how users can handle errors, by means of an interactive debugger; that is, it explains how to recover if you do something wrong. A new user of the Lisp Machine, or someone who just wants to know how to deal with errors and not how to cause them, should ignore the first sections and skip ahead to section 30.7, page 726.

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

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

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

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

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

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

30.1 Conditions

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

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

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

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

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

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

For example, among the condition names defined by the system are **condition**, **error**, **sys:arithmetic-error**, **sys:floating-exponent-underflow** and **sys:divide-by-zero**. **condition** is a condition name that all condition instances possess. **error** identifies the category of conditions that are considered errors. **sys:arithmetic-error** identifies the category of errors that pertain to arithmetic operations. **sys:floating-exponent-underflow** and **sys:divide-by-zero** are the most specific level of categorization. So, the condition signaled when you evaluate **(* 1s-30 1s-30 1s-30 1s-30)** possesses condition names **sys:floating-exponent-underflow**, **sys:arithmetic-error**, **error** and **condition**, while the one signaled if you evaluate **(// 1 0)** possesses condition names **sys:divide-by-zero**, **sys:arithmetic-error**, **error** and **condition**. In this example, the categories fall into a strict hierarchy, but this does not need to be the case.

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

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

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

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

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

condition-typep *condition-instance condition-name*

Returns t if *condition-instance* possesses condition name *condition-name*. *condition-name* can also be a combination of condition names using and, or and not; then the condition tested for is a boolean combination of the presence or absence of various condition names. Example:

```
(condition-typep error 'fs:file-not-found)
(condition-typep error
   '(or fs:file-not-found fs:directory-not-found))
```

errorp *object*

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

:condition-names *Operation on* condition

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

30.2 Handling Conditions

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

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

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

ignore-errors *body...* *Macro*

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

Errors whose condition instances return true for the :dangerous-condition-p operation are not handled. These include such things as running out of virtual memory.

A handler can also signal another condition. For example, signaling **sys:abort** has the effect of pretending that the user typed the **Abort** key. The following function creates a handler which signals **sys:abort**.

si:eval-abort-trivial-errors *form*

Evaluates *form* with a condition handler for many common error conditions such as :wrong-type-argument, :unbound-variable and :unclaimed-message. The handler asks the user whether to allow the debugger to be entered. If the user says 'no', the handler signals the **sys:abort** condition. If the user says 'yes', the handler does not handle the condition, allowing the debugger to do so.

In some cases the handler attempts to determine whether the incorrect variable, operation, or argument appeared in *form*; if it did not, the debugger is always allowed to run. The assumption is that *form* was typed in by the user, and the intention is to distinguish trivial mistakes from program bugs.

The handler can also ask to proceed from the condition. This is done by returning a non-**nil** value. See the section on proceeding, page 717, for more information.

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

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

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

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

condition-bind (*handlers...*) *body...* *Macro*
condition-bind-default (*handlers...*) *body...* *Macro*

A condition-bind form looks like this:

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

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

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

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

```
(condition-bind ((nil 'myhandler "it happened here" 45))
  (catch 'x
    ...))

(defun myhandler (condition string value)
  (unless (or (condition-typep condition 'fs:file-error)
              (send condition :dangerous-condition-p)
              (send condition :debugging-condition-p))
    (format error-output "~&~A:~%~A~%" string condition)
    (throw 'x value)))
```

myhandler declines to handle file errors, and all debugging conditions and dangerous errors. For all other conditions, it prints the string specified in the condition bind and throws to the tag **x** the value specified there (45).

condition-bind-default is like **condition-bind** but establishes a *default handler* instead of an ordinary handler. Default handlers work like ordinary handlers, but they are tried in a different order: first all the applicable ordinary handlers are given a chance to handle the condition, and then the default handlers get their chance. A more flexible way of doing things like this is described under **signal-condition** (page 715).

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

condition-case (*variables...*) *body-form clauses...* *Macro*

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

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

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

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

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

variable may be omitted if it is not used.

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

Here is an example:
```
(condition-case ()
    (print foo)
  (error (format t " <<Error in printing>>")))
```

condition-call (*variables...*) *body-form clauses...* *Macro*

condition-call is an extension of **condition-case** that allows you to give each clause an arbitrary conditional expression instead of just a list of condition names. It looks like this:
```
(condition-call (variables...)
    body-form
  (test forms...)
  (test forms...)
  ...)
```
The difference between this and **condition-case** is the *test* in each clause. The clauses in a **condition-call** resemble the clauses of a **cond** rather than those of a **selectq**.

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

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

The last clause can be a **:no-error** clause just as in **condition-case**. It is executed if the body returns without error. The values returned by the body are stored in the *variables*. The values of the last form in the **:no-error** clause are returned by the **condition-call**.

Only the first of *variables* is used if there is no :no-error clause. The *variables* may be omitted entirely in the unlikely event that none is used. Example:

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

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

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

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

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

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

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

```
(condition-case-if cond-form (variables...)
    body-form
  (condition-names forms...)
  (condition-names forms...)
  ...)
```

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

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

```
(condition-call-if cond-form (variables...)
    body-form
  (test forms...)
  (test forms...)
  ...)
```

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

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

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

eh:condition-handlers *Variable*

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

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

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

function is the actual handler function.

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

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

eh:condition-default-handlers *Variable*

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

30.3 Standard Condition Flavors

condition *Flavor*

The flavor condition is the base flavor of all conditions, and provides default definitions for all the operations described in this chapter.

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

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

error *Flavor*

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

sys:no-action-mixin *Flavor*

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

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

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

ferror *Flavor*

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

sys:warning *Flavor*

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

sys:bad-array-mixin *Flavor*

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

30.4 Condition Operations

Every condition instance can be asked to print an *error message* which describes the circumstances that led to the signaling of the condition. The easiest way to print one is to print the condition instance without escaping (princ, or format operation ~A). This actually uses the :report operation, which implements the printing of an error message. When a condition instance is printed with escaping, it uses the #⊂ syntax so that it can be read back in. This is done using si:print-readably-mixin, page 446.

:report *stream* *Operation on* condition

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

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

:report-string *Operation on* condition

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

Operations provided specifically for condition handlers to use:

:dangerous-condition-p *Operation on* condition

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

:debugging-condition-p *Operation on* condition

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

See also the operations :proceed-types and :proceed-type-p, which have to do with proceeding (page 717).

30.4.1 Condition Operations for the Debugger

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The fourth value, if non-nil, tells the debugger to consider the innermost frame to be "interesting". Normally, frames that are part of the interpreter (calls to si:eval1, si:apply-lambda, prog, cond, etc.) are considered uninteresting.

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

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

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

This operation uses :or method combination (see section 21.11, page 433). Some condition flavors add methods that perform some other sort of processing or enter a different command loop. For example, unbound variable errors look for look-alike symbols in other packages at this point. If the added method returns nil, the original method that enters the usual debugger command loop is called.

30.5 Signaling Conditions

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

30.5.1 Convenience Functions for Signaling

ferror &rest *make-condition-arguments*

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

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

```
        (ferror 'math:singular-matrix
                "The matrix ~S cannot be inverted." matrix)
```

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

If you prefer, you can use the formatted output functions (page 496) to generate the error message. Here is an example, though in a simple case like this using **format** is easier:

```
(ferror 'math:singular-matrix
  (format:outfmt
    "The matrix "
    (prin1 matrix)
    " cannot be inverted.")
  number)
```

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

cerror *proceed-type ignore signal-name* &rest *signal-name-arguments*

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

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

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

Common Lisp defines another calling sequence for this function:

`(cerror` *continue-format-string error-format-string args*`...)`

This signals an error of flavor **eh:common-lisp-cerror**, which prints an error message using *error-format-string* and *args*. It allows one proceed type, whose documentation is *continue-format-string*, and which proceeds silently, returning **nil** from **cerror**. **cerror** can tell which calling sequence has been used and behaves accordingly.

warn *format-string* &rest *args*

Prints a warning on ***error-output*** by passing the args to **format**, starting on a fresh line, and then returns.

If ***break-on-warnings*** is non-nil, however, **warn** signals a proceedable error, using the arguments to make an error message. If the user proceeds, **warn** simply returns.

break-on-warnings

If non-nil, **warn** signals an error rather than just printing a message.

check-type *place type-spec* [*description*] *Macro*
check-arg-type *place type-spec* [*description*] *Macro*

Signals a correctable error if the value of *place* does not fit the type *type-spec*. *place* is something that **setf** can store in. *type-spec* is a type specifier, a suitable second argument to **typep**, and is not evaluated (see section 2.3, page 14). A simple example is:

```
(check-type foo (integer 0 10))
```

This signals an error unless **(typep foo '(integer 0 10))**; that is, unless **foo**'s value is an integer between zero and ten, inclusive.

If an error is signaled, the error message contains the name of the variable or place where the erroneous value was found, and the erroneous value itself. An English description of the type of object that was wanted is computed automatically from the type specifier for use in the error message. For the commonly used type specifiers this computed description is adequate. If it is unsatisfactory in a particular case, you can specify *description*, which is used instead. In order to make the error message grammatical, *description* should start with an indefinite article.

The error signaled is of condition **sys:wrong-type-argument** (see page 61). The proceed type **:argument-value** is provided. If a handler proceeds using this proceed type, it should specify one additional argument; that value is stored into *place* with **setf**. The new value is then tested, and so on. **check-type** returns when a value passes the test.

check-arg-type is an older name for this macro.

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

check-arg is an obsolete variant of **check-type**. *predicate* is either a symbol which is predicate (a function of one argument) or a list which is a form. If it is a predicate, it is applied to the value of *var-name*, which is valid if the predicate returns non-**nil**. If it is a form, it is evaluated, and the value is valid of the form returns non-**nil**. The form ought to make use of *var-name*, but nothing checks this.

There is no way to compute an English description of valid values from *predicate*, so a *description* string must always be supplied.

type-symbol is a symbol that is used by condition handlers to determine what type of argument was expected. If *predicate* is a symbol, you may omit *type-symbol*, and *predicate* is used for that purpose as well. The use of the *type-symbol* is not really well-defined, and **check-type**, where a type specifier serves both purposes, is superior to **check-arg** for this reason.

Examples:

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

(check-arg h
           (or (stringp h) (typep h 'fs:host))
           "a host name"
           fs:host)
```

Other functions that can be used to test for invalid values include **ecase** and **ccase** (page 66), which are error-checking versions of **selectq**, and **etypecase** and **ctypecase** (page 21), error-checking versions of **typecase**.

assert *test-form* [(*places...*) [*string args...*]] *Macro*

Signals an error if *test-form* evaluates to **nil**. The rest of the **assert** form is relevant only if the error happens.

First of all, the *places* are forms that can be stored into with **setf**, and which are used (presumably) in *test-form*. The reason that the *places* are specified again in the **assert** is so that the expanded code can arrange for the user to be able to specify a new value to be stored into any one of them when he proceeds from the error. When the error is signaled, one proceed-type is provided for each *place* that is given. The condition object has flavor **eh:failed-assertion**.

If the user does proceed with a new value in that fashion, the *test-form* is evaluated again, and the error repeats until the *test-form* comes out non-**nil**.

The *string* and *args* are used to print the error message. If they are omitted, **"Failed assertion"** is used. They are evaluated only when an error is signaled, and are evaluated again each time an error is signaled. **setf**'ing the *places* may also involve evaluation, which happens each time the user proceeds and sets one.

Example:

```
(assert (neq (car a) (car b)) ((car a) (car b))
        "The CARS of A and B are EQ: ~S and ~S"
        (car a) (car b))
```

The *places* here are **(car a)** and **(car b)**. The *args* happen to be the same two forms, by not-exactly-coincidence; the current values of the *places* are often useful in the error message.

The remaining signaling functions are provided for compatibility only.

error &rest *make-condition-arguments*

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

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

which is used in Maclisp, and

(error *condition-instance*)

(error *flavor-name init-options...*)

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

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

cli:error *format-string* &rest *args*

The Common Lisp version of **error** signals an uncorrectable error whose error message is printed by passing *format-string* and *args* to **format**.

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

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

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

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

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

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

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

errset *form* [*flag*] *Macro*

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

errset is an old, Maclisp construct, implemented much like condition-case. Many uses of errset or errset-like constructs really ought to be checking for more specific conditions instead.

catch-error *form* [*flag*] *Macro*

catch-error is a variant of errset. This construct catches errors during the evaluation of *form* and returns two values. If *form* returns normally, the first value is *form*'s first value and the second value is nil. If an error occurs, the usual error message is printed unless *flag* is nil, and then control is thrown out of the catch-error form, which returns two values, first nil and second a non-nil value that indicates the occurrence of an error. *flag* is evaluated before *form* and is optional, defaulting to t.

errset *Variable*

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

err *Macro*

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

(err) is a dumb way to cause an error. If executed inside an errset, that errset returns nil, and no message is printed. Otherwise an error is signaled with error message just "ERROR>>".

(err *form*) evaluates *form* and causes the containing errset to return the result. If executed when not inside an errset, an error is signaled with *form*'s value printed as the error message.

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

30.5.2 Creating Condition Instances

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

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

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

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

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

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

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

Here is a typical **defsignal**:

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

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

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

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

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

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

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

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

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

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

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

documentation is a string which is recorded so that it can be accessed via the function documentation, as in (documentation *signal-name* 'signal).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

30.5.3 Signaling a Condition Instance

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

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

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

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

signal-condition tries various possible handlers for the condition. First eh:condition-handlers is scanned for handlers that are applicable (according to the condition names they specify) to this condition instance. After this list is exhausted, eh:condition-default-handlers is scanned the same way. Each handler that is tried can terminate the act of signaling by throwing out of signal-condition, or it can specify a way to proceed from the signal. The handler can also return nil to decline to handle the condition; then the next possible handler is offered a chance.

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

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

Otherwise, by this stage, a proceed type has been chosen from the available list. If the proceed type was among those specified by the caller of signal-condition, then proceeding consists simply of returning to that caller. The chosen proceed type is the first value, and arguments (returned by the handler along with the proceed type) may follow it. If the proceed type was implemented nonlocally with condition-resume (see page 723), then the associated proceed handler function on eh:condition-resume-handlers is called.

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

The purpose of condition-bind-default is so that you can define a handler that is allowed to handle a signal only if none of the callers' handlers handle it. A more flexible technique for doing this sort of thing is to make an ordinary handler signal the same condition instance recursively by calling signal-condition, like this:

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

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

eh:trace-conditions *Variable*

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

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

30.6 Proceeding

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

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

Each signaler may or may not implement all the proceed types which are meaningful in general for the condition names being signaled. For example, it is futile to proceed from a **sys:wrong-type-argument** error with **:argument-value** unless the signaler knows how to take the associated value and store it into the argument, or do something else that fits the conceptual specifications of **:argument-value**. For some signalers, it may not make sense to do this at all. Therefore, one of the arguments to **signal-condition** is a list of the proceed types that this particular signaler knows how to handle.

In addition to the proceed types specified by the individual signaler, other proceed types can be provided nonlocally; they are implemented by a *resume handler* which is in effect through a dynamic scope. See below, section 30.6.3, page 723.

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

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

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

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

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

:proceed-types *Operation on* condition

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

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

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

30.6.1 Proceeding and the Debugger

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

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

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

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

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

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

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

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

Prints on *stream* a description of the purpose of proceed type *proceed-type*. This operation uses :case method combination (see section 21.11, page 433), to make it convenient to define the way to document an individual proceed type. The string printed should start with an imperative verb form, capitalized, and end with a period. Example:

This example is an :or method so that it can consider any proceed type. If it returns non-nil, the system considers that it has handled the proceed type and no other methods get a chance. eh:places is an instance variable of the flavor sys:failed-assertion; its values are the proceed types this method understands.

```
(defmethod (sys:failed-assertion :or :document-proceed-type)
           (proceed-type stream ignore)
  (when (memq proceed-type eh:places)
    (format stream
            "Try again, setting ~S.~
             You type an expression for it."
            proceed-type)
    t))
```

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

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

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

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

:case method combination is used (see section 21.11, page 433), making it possible to define the handling of each proceed type individually in a separate function. The documentation string of the :case method for a proceed type is also used as the default for :document-proceed-type on that proceed type.

The argument *continuation* is an internal function of the debugger which actually proceeds from the signaled condition if the :proceed-asking-user method calls it. This is the only way to cause proceeding actually to happen. Call *continuation* with funcall, giving a proceed type and suitable arguments. The proceed type passed to *continuation* need not be the same as the one given to :proceed-asking-user; it should be one of the proceed types available for handlers to use.

Alternatively, the :prompt-and-read method can return without calling *continuation*; then the debugger continues to read commands. The options which the fs:no-more-room condition offers in the debugger, to run Dired or expunge a directory, work this way.

The argument *read-object-fn* is another internal function of the debugger whose purpose is to read arguments from the user or request confirmation. If you wish to do those things, you must funcall *read-object-function* to do it. Use the calling sequence documented for the function prompt-and-read (see page 453). (The *read-object-fn* may or may not actually use prompt-and-read.)

Here is how sys:proceed-with-value-mixin provides for the proceed type :new-value. Note the documentation string, which is automatically use by :document-proceed-type since no :case method for that operation is provided.

```
(defmethod (proceed-with-value-mixin
                      :case :proceed-asking-user :new-value)
           (continuation read-object-function)
  "Return a value; the value of an expression you type."
  (funcall continuation :new-value
           (funcall read-object-function
                    :eval-read
                    "~&Form whose value to use instead: ")))
```

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

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

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

Finally, there may be proceed types intended only for the debugger which do not actually proceed; these should be inserted into the list by the :user-proceed-types method.

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

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

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

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

:pass-on method combination is used (see section 21.11, page 433), so that if multiple mixins define methods for :user-proceed-types, each method gets a chance to add or remove proceed types. The methods should not actually modify the argument, but should cons up a new list in which certain keywords are added or removed according to the other keywords that are there.

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

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

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

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

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

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

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

30.6.2 How Signalers Provide Proceed Types

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

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

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

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

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

Signals a condition, providing proceed types and code to implement them. Each clause specifies a proceed type to provide, and also contains code to be run if a handler should proceed with that proceed type.

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

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

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

Example:

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

(signal-proceed-case
        ((newarg)
         'my-wrong-type-arg
         "The argument ~A was ~S, which is not a cons."
         'foo foo 'cons)
  (:argument-value (car newarg)))
```

The signal name my-wrong-type-arg creates errors with condition name sys:wrong-type-argument. The signal-proceed-case shown signals such an error, and handles the proceed type :argument-value. If a handler proceeds using that proceed type, the handler's value is put in newarg, and then its car is returned from the signal-proceed-case.

30.6.3 Nonlocal Proceed Types

When the caller of signal-condition specifies proceed types, these are called *local proceed types*. They are implemented at the point of signaling. There are also *nonlocal proceed types*, which are in effect for all conditions (with appropriate condition names) signaled during the execution of the body of the establishing macro. We say that the macro establishes a *resume handler* for the proceed type.

The most general construct for establishing a resume handler is condition-resume. For example, in

```
(condition-resume
   '(fs:file-error :retry-open t
                   ("Proceeds, opening the file again.")
                   (lambda (ignore) (throw 'tag nil)))
  (do-forever
    (catch 'tag (return (open pathname)))))
```

the proceed type :retry-open is available for all fs:file-error conditions signaled within the call to open.

condition-resume *handler-form* &body *body* *Macro*
condition-resume-if *cond-form handler-form* &body *body* *Macro*

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

The value of the *handler-form* should be a list with at least five elements:

(*condition-names proceed-type predicate format-string-and-args handler-function additional-args...*)

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

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

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

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

handler-function is the function called to do the work of proceeding, once this proceed type has been returned by a condition handler or the debugger.

catch-error-restart-explicit-if makes it easy to establish a particular simple kind of resume handler.

catch-error-restart-explicit-if *Macro*
cond-form (*condition-names proceed-type format-string format-args...*) *body...*

Executes *body* with (if *cond-form* produces a non-nil value) a resume handler for proceed type *proceed-type* and condition(s) *condition-names*. *condition-names* should be a symbol or a list of symbols; it is not evaluated. *proceed-type* should be a symbol.

If proceeding is done using this resume handler, control returns from the catch-error-restart-explicit-if form. The first value is nil and the second is non-nil.

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

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

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

You are allowed to use "anonymous" nonlocal proceed types, which have no conventional meaning and are not specially known to the :document-proceed-type and :proceed-asking-user operations. The anonymous proceed type may be any Lisp object. The default definition of :proceed-asking-user handles an anonymous proceed type by simply calling the continuation passed to it, reading no arguments. The default definition of :document-proceed-type handles anonymous proceed types by passing format the *format-string-and-args* list found in the resume handler (this is what that list is for).

Anonymous proceed types are often lists. Such proceed types are usually made by some variant of error-restart, and they are treated a little bit specially. For one thing, they are all put at the end of the list returned by the :proceed-types operation. For another, the debugger command Control-C or Resume never uses a proceed type which is a list. If no atomic proceed type is available, Resume or Control-C is not allowed.

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

All execute body with an anonymous resume handler for *condition-names*. The proceed type for this resume handler is a list, so the **Resume** key will not use it. *condition-names* is either a single condition name or a list of them, or **nil** meaning all conditions; it is not evaluated.

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

If the resume handler made by **error-restart** is invoked by proceeding from a signal, the automatically generated resume handler function does a throw back to the **error-restart** and the body is executed again from the beginning. If body returns, the values of the last form in it are returned from the **error-restart** form.

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

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

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

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

These forms are typically used by any sort of command loop, so that aborting within the command loop returns to it and reads another command. **error-restart-loop** is often right for simple command loops. **catch-error-restart** is useful when aborting should terminate execution rather than retry, or with an explicit conditional to test whether a throw was done.

error-restart forms often specify **(error sys:abort)** as the *condition-names*. The presence of **error** causes them to be listed (and assigned command characters) by the debugger, for all errors, and the presence of **sys:abort** causes the **Abort** key to use them. If you would like a procede type to be offered as an option by the debugger, but do not want the **Abort** key to use it, specify just **error**.

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

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

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

eh:condition-resume-handlers *Variable*

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

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

sys:abort (condition) *Condition*

This condition is signaled by the Abort key; it is how that key is implemented. Most command loops use some version of error-restart to set up a resume handler for sys:abort so that it will return to the innermost command loop if (as is usually the case) no handler handles it. These resume handlers usually apply to error as well as sys:abort, so that the debugger will offer a specific command to return to the command loop even if it is not the innermost one.

30.7 The Debugger

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

30.7.1 Entering the Debugger

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

```
>>TRAP 5543 (TRANS-TRAP)
The symbol FOOBAR is unbound.
While in the function LOSE-XCT ← LOSE-COMMAND-LOOP ← LOSE
```

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

Here is an example of an error from Lisp code:

```
>>ERROR: The argument X was 1, which is not a symbol,
While in the function FOO ← SI:EVAL1 ← SI:LISP-TOP-LEVEL1
```

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

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

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

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

The debugger normally uses the stream *debug-io* for all its input and output (see page 460). By default it is a synonym for *terminal-io*. The value of this variable in the stack group in which the error was signaled is the one that counts.

eh:*debug-io-override* *Variable*

If this is non-nil, it is used by the debugger instead of the value of *debug-io*. The value in the stack group where the error was signaled is the one that counts.

The debugger can be manually entered either by causing an error (e.g. by typing a ridiculous symbol name such as ahsdgf at the Lisp read-eval-print loop) or by typing the Break key with the Meta shift held down while the program is reading from the terminal. Typing the Break key with both Control and Meta held down forces the program into the debugger immediately, even if it is running. If the Break key is typed without Meta, it puts you into a read-eval-print loop using the break function (see page 795) rather than into the debugger.

eh &optional *process*

Causes *process* to enter the debugger, and directs the debugger to read its commands from the ambient value of *terminal-io*, current when you call eh, rather than *process* 's own value of *terminal-io* which is what would be used if *process* got an error in the usual way. The process in which you invoked eh waits while you are in the debugger, so there is no ambiguity about which process will handle your keyboard input.

If *process* had already signaled an error and was waiting for exposure of a window, then it enters the debugger to handle that error. Otherwise, the break condition is signaled in it (just like what Control-Meta-Break does) to force it into the debugger.

The Resume command makes *process* resume execution. You can also use other debugger commands such as Abort, Control-R, Control-Meta-R and Control-T to start it up again. Exiting the debugger in any way causes eh to return in its process.

process can also be a window, or any flavor instance which understands the :process operation and returns a process.

If *process* is not a process but a stack group, the current state of the stack group is examined. In this case, the debugger runs in "examine-only" mode, and executes in the process in which you invoked **eh**. You cannot resume execution of the debugged stack group, but **Resume** exits the debugger. It is your responsibility to ensure that no one tries to execute in the stack group being debugged while the debugger is looking at it.

If *process* is **nil**, **eh** finds all the processes waiting to enter the debugger and asks you which one to use.

30.7.2 How to Use the Debugger

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

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

```
→
```

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

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

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

If you type a Lisp expression, it is interpreted as a Lisp form and evaluated in the context of the current frame. That is, all dynamic bindings used for the evaluation are those in effect in the current frame, with certain exceptions explained below. The results of the evaluation are printed, and the debugger prompts again with an arrow. If, during the typing of the form, you change your mind and want to get back to the debugger's command level, type the **Abort** key or a **Control-G**; the debugger responds with an arrow prompt. In fact, at any time that input is expected from you, while you are in the debugger, you may type **Abort** or **Control-G** to cancel any debugger command that is in progress and get back to command level. **Control-G** is useful because it can never exit from the debugger as **Abort** can.

This **read-eval-print** loop maintains the values of **+**, *****, and **-** almost like the Lisp listen loop. The difference is that some single-character debugger commands such as **C-M-A** also set ***** and **+** in their own way.

If an error occurs in the evaluation of the Lisp expression you type, you may enter a second invocation of the debugger, looking at the new error. The prompt in this event is '→→' to make it clear which level of error you are examining. You can abort the computation and get back to the first debugger level by typing the **Abort** key (see below).

Various debugger commands ask for Lisp objects, such as an object to return or the name of a catch-tag. You must type a form to be evaluated; its value is the object that is actually used. This provides greater generality, since there are objects to which you might want to refer that cannot be typed in (such as arrays). If the form you type is non-trivial (not just a constant form), the debugger shows you the result of the evaluation and asks for confirmation before proceeding. If you answer negatively, or if you abort, the command is canceled and the debugger returns to command level. Once again, the special bindings in effect for evaluation of the form are those of the current frame you have selected.

The **Meta-S** and **Control-Meta-S** commands allow you to look at the bindings in effect at the current frame. A few variables are rebound by the debugger itself whenever a user-provided form is evaluated, so you you must use **Meta-S** to find the values they actually had in the erring computation.

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

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

+, ++, +++
, **, ***, *values
: **+** and ***** are rebound to the debugger's previous form and previous value. Commands for examining arguments and such, including **C-M-A**, **C-M-L** and **C-M-V**, leave ***** set to the value examined and **+** set to a locative to where the value was found. When the debugger is first entered, **+** is the last form typed, which is typically the one that caused the error, and ***** is the value of the *previous* form.

evalhook
applyhook These variables (see page 748) are rebound to **nil**, turning off the **step** facility if it was in use when the error occurred.

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

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

30.7.3 Debugger Commands

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

BAR:
   Arg 0 (ADDEND): 13

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

and so on. The backtrace commands all accept numeric arguments which say how many frames to describe, the default being to describe all the frames.

Moving around in the stack:

The Control-N command makes the "next" older frame be current. This is the frame which called the one that was current at before. The new current frame's function and arguments are printed in the format shown immediately above.

Control-P moves the current frame in the reverse direction. If you use it immediately after getting an error it selects frames that are part of the act of signaling.

Meta-< selects the frame in which the error occurred, the same frame that was selected when the debugger was entered. Meta-> selects the outermost or initial stack frame. Control-S asks you for a string and searches down the stack (toward older frames) from the current frame for a frame whose executing function's name contains that string. That frame becomes current and is printed out. These commands are easy to remember since they are analogous to editor commands.

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

Meta-L prints out the current frame in "full screen" format, which shows the arguments and their values, the local variables and their values, and the machine code with an arrow pointing to the next instruction to be executed. Refer to chapter 31, page 752 for help in reading this machine code.

Commands such as Control-N and Control-P, which are useful to issue repeatedly, take a prefix numeric argument and repeat that many types. The numeric argument is typed by using Control or Meta and the number keys, as in the editor. Some other commands such as Control-M also use the numeric argument; refer to the table at the end of the section for detailed information.

Resuming execution:

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

Control-R is used to return a value or values from the current frame; the frame that called that frame continues running as if the function of the current frame had returned. This command prompts you for each value that the caller expects; you can type either a form which evaluates to the desired value or End if you wish to return no more values.

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

Control-Meta-R *reinvokes* the current frame; it starts execution at the beginning of the function, with the arguments currently present in the stack frame. These are the same arguments the function was originally called with unless either the function itself has changed them with setq or you have set them in the debugger. If the function has been redefined in the meantime (perhaps you edited it and fixed its bug) the new definition is used. Control-Meta-R asks for confirmation before resuming execution.

Meta-R is similar to Control-Meta-R but allows you to change the arguments if you wish. You are prompted for the new arguments one by one; you can type a form which evaluates to the desired argument, or Space to leave that argument unchanged, or End if you do not want any more arguments. Space is allowed only if this argument was previously passed, and End is not allowed for a required argument. Once you have finished specifying the arguments, you must confirm before execution resumes.

Stepping through function calls and returns:

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

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

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

Transfering to other systems:

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

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

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

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

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

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

The names eh:arg, eh:val, etc. are for compatibility with the Symbolics system.

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

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

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

Like eh-arg but accesses the current frame's local variables instead of its arguments.

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

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

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

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

eh-fun
eh:fun

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

30.7.4 Summary of Commands

Control-A	Prints argument list of function in current frame.
Control-Meta-A	Sets * to the *n*th argument of the current frame.
Control-B	Prints brief backtrace.
Meta-B	Prints longer backtrace.
Control-Meta-B	Prints longer backtrace with no censoring of "uninteresting" functions.
Control-C or Resume	Attempts to continue, using the first proceed type on the list of available ones for this error.
Meta-C	Attempts to continue, setq'ing the unbound variable or otherwise "permanently" fixing the error. This uses the proceed type :store-new-value, and is available only if that proceed type is.

Control-D	Attempts to continue like Control-C, but trap on the next function call.
Meta-D	Toggles the flag that causes a trap on the next function call after you continue or otherwise exit the debugger.
Control-E	Switches to Zmacs to edit the source code for the function in the current frame.
Control-Meta-F	Sets * to the function in the current frame.
Control-G or Abort	Quits to command level. This is not a command, but something you can type to escape from typing in an argument of a command.
Control-Meta-H	Describes the condition handlers and resume handlers established by the current frame.
Control-L or Clear-Screen	Redisplays error message and current frame.
Meta-L	Displays the current frame, including local variables and compiled code.
Control-Meta-L	Sets * to the value of local variable *n* of the current frame.
Control-M	Sends a bug report containing the error message and a backtrace of *n* frames (default is 3).
Control-N or Line	Moves to the next (older) frame. With argument, moves down *n* frames.
Meta-N	Moves to next frame and displays it like Meta-L. With argument, move down *n* frames.
Control-Meta-N	Moves to next frame even if it is "uninteresting" or still accumulating arguments. With argument, moves down *n* frames.
Control-P or Return	Moves up to previous (newer) frame. With argument, moves up *n* frames.
Meta-P	Moves to previous frame and displays it like Meta-L. With argument, moves up *n* frames.
Control-Meta-P	Moves to previous frame even if it is "uninteresting" or still accumulating arguments. With argument, moves up *n* frames.
Control-R	Returns a value or values from the current frame.
Meta-R	Reinvokes the function in the current frame (restart its execution at the beginning), optionally changing the arguments.
Control-Meta-R	Reinvokes the function in the current frame with the same arguments.
Control-S	Searches for a frame containing a specified function.
Meta-S	Reads the name of a special variable and returns that variable's value in the current frame. Instance variables of **self** may also be specified even if not special.
Control-Meta-S	Prints a list of special variables bound by the current frame and the values they are bound to by the frame. If the frame binds **self**, all the instance variables of **self** are listed even if they are not special.
Control-T	Throws a value to a tag.

Control-Meta-U	Moves down the stack to the previous "interesting" frame.
Control-X	Toggles the flag in the current frame that causes a trap on exit or throw through that frame.
Meta-X	Sets the flag causing a trap on exit or throw through the frame for the current frame and all the frames outside of it.
Control-Meta-X	Clears the flag causing a trap on exit or throw through the frame for the current frame and all the frames outside of it.
Control-Meta-V	Sets * to the *n*th value being returned from the current frame. This is non-nil only in a trap on exit from the frame.
Control-Meta-W	Switches to the window-oriented debugger.
Control-Z or Abort	Aborts the computation and throw back to the most recent break or debugger, to the program's command level, or to Lisp top level.
? or Help	Prints debugger command self-documentation.
Meta-<	Moves to the frame in which the error was signaled, and makes it current once again.
Meta->	Moves to the outermost (oldest) stack frame.
Control-0 through Control-Meta-9	Numeric arguments to the following command are specified by typing a decimal number with Control and/or Meta held down.
Super-A, etc.	The commands Super-A, Super-B, etc. are assigned to all the available proceed types for this error. The assignments are different each time the debugger is entered, so it prints a list of them when it starts up. Help P prints the list again.

30.7.5 Deexposed Windows and Background Processes

If the debugger is entered in a window that is not exposed, a notification is used to inform you that it has happened.

In general, a notification appears as a brief message printed inside square brackets if the selected window can print it. Otherwise, blinking text appears in the mouse documentation line telling you that a notification is waiting; to see the notification, type Terminal N or select a window that can print it. In either case, an audible beep is made.

In the case of a notification that the debugger is waiting to use a deexposed window, you can select and expose the window in which the error happened by typing Terminal 0 S. You can do this even if the notification has not been printed yet because the selected window cannot print it. If you select the waiting window, in this way or in any other way, the notification is discarded since you already know what it was intended to tell you.

If the debugger is entered in a process that has no window or other suitable stream to type out on, the window system assigns it a "background window". Since this window is initially not exposed, a notification is printed as above and you must use Terminal 0 S to see the window.

If an error happens in the scheduler stack group or the first level error handler stack group which are needed for processes to function, or in the keyboard or mouse process (both needed for the window system to function), the debugger uses the *cold load stream*, a primitive facility for terminal I/O which bypasses the window system.

If an error happens in another process but the window system is locked so that the notification mechanism cannot function, the cold load stream is used to ask what to do. You can tell the debugger to use the cold load stream immediately, to forcibly clear the window system locks and notify immediately as above, or to wait for the locks to become unlocked and then notify as above. If you tell it to wait, you can resume operation of the machine. Meanwhile, you can use the command Terminal Control-Clear-Input to forcibly unlock the locks, or Terminal Call to tell the debugger to use the cold load stream. The latter command normally enters a break-loop that uses the cold-load stream, but if there are any background errors, it offers to enter the debugger to handle them. You can also handle the errors in a Lisp listen loop of your choice by means of the function eh (page 727), assuming you can select a functioning Lisp listen loop.

30.7.6 Debugging after a Warm Boot

After a warm boot, the process that was running at the time of booting (or at the time the machine crashed prior to booting) may be debugged if you answer 'no' when the system asks whether to reset that process.

si:debug-warm-booted-process
Invoke the debugger, like the function eh (page 727), on the process that was running as of the last warm boot (assuming there was such a process).

On the CADR, the state you see in the debugger is not correct; some of the information dates from some period of time in advance of the boot or the crash.

On the Lambda, the state you see in the debugger will, in some system version, be accurate.

30.8 Tracing Function Execution

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

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

trace *Special form*

A trace form looks like:

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

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

a symbol	This is a function name, with no options. The function is traced in the default way, printing a message each time it is called and each time it returns.
a list (*function-name option-1 option-2* ...)	*function-name* is a symbol and the *options* control how it is to be traced. The various options are listed below. Some options take arguments, which should be given immediately following the option name.
a list (:function *function-spec option-1 option-2* ...)	This is like the previous form except that *function-spec* need not be a symbol (see section 11.2, page 223). It exists because if *function-name* was a list in the previous form, it would instead be interpreted as the following form:
a list ((*function-1 function-2*...) *option-1 option-2* ...)	All of the functions are traced with the same options. Each *function* can be either a symbol or a general function-spec.

The following trace options exist:

:break *pred*	Causes a breakpoint to be entered after printing the entry trace information but before applying the traced function to its arguments, if and only if *pred* evaluates to non-nil. During the breakpoint, the symbol arglist is bound to a list of the arguments of the function.
:exitbreak *pred*	This is just like break except that the breakpoint is entered after the function has been executed and the exit trace information has been printed, but before control returns. During the breakpoint, the symbol arglist is bound to a list of the arguments of the function, and the symbol values is bound to a list of the values that the function is returning.
:error	Causes the error handler to be called when the function is entered. Use Resume (or Control-C) to continue execution of the function. If this option is specified, there is no printed trace output other than the error message printed by the error handler. This is semi-obsolete, as breakon is more convenient and does more exactly the right thing.
:step	Causes the function to be single-stepped whenever it is called. See the documentation on the step facility, section 30.11, page 746.
:stepcond *pred*	Causes the function to be single-stepped only if *pred* evaluates to non-nil.
:entrycond *pred*	Causes trace information to be printed on function entry only if *pred* evaluates to non-nil.
:exitcond *pred*	Causes trace information to be printed on function exit only if *pred* evaluates to non-nil.

:cond *pred*	This specifies both :exitcond and :entrycond together.
:wherein *function*	Causes the function to be traced only when called, directly or indirectly, from the specified function *function*. One can give several trace specs to trace, all specifying the same function but with different wherein options, so that the function is traced in different ways when called from different functions. This is different from advise-within, which only affects the function being advised when it is called directly from the other function. The trace :wherein option means that when the traced function is called, the special tracing actions occur if the other function is the caller of this function, or its caller's caller, or its caller's caller's caller, etc.
:argpdl *pdl*	Specifies a symbol *pdl*, whose value is initially set to nil by trace. When the function is traced, a list of the current recursion level for the function, the function's name, and a list of arguments is consed onto the *pdl* when the function is entered, and cdr'ed back off when the function is exited. The *pdl* can be inspected from within a breakpoint, for example, and used to determine the very recent history of the function. This option can be used with or without printed trace output. Each function can be given its own pdl, or one pdl may serve several functions.
:entryprint *form*	The *form* is evaluated and the value is included in the trace message for calls to the function. You can give this option multiple times, and all the *form*'s thus specified are evaluated and printed. \\ precedes the values to separate them from the arguments.
:exitprint *form*	The *form* is evaluated and the value is included in the trace message for returns from the function. You can give this option multiple times, and all the *form*'s thus specified are evaluated and printed. \\ precedes the values to separate them from the returned values.
:print *form*	The *form* is evaluated and the value is included in the trace messages for both calls to and returns from the function. Equivalent to :exitprint and :entryprint at once.
:entry *list*	This specifies a list of arbitrary forms whose values are to be printed along with the usual entry-trace. The list of resultant values, when printed, is preceded by \\ to separate it from the other information.
:exit *list*	This is similar to entry, but specifies expressions whose values are printed with the exit-trace. Again, the list of values printed is preceded by \\.
:arg :value :both nil	These specify which of the usual trace printouts should be enabled. If :arg is specified, then on function entry the name of the function and the values of its arguments will be printed. If :value is specified, then on function exit the returned value(s) of the function will be printed. If :both is specified, both of these will be printed. If nil is specified, neither will be printed. If none of these four options are specified the default is to :both. If any further *options* appear after one of these, they are not treated as options! Rather, they are considered to be arbitrary forms whose values are to be printed on entry and/or exit to the function,

along with the normal trace information. The values printed will be preceded by a //, and follow any values specified by :entry or :exit.

Note that since these options "swallow" all following options, if one is given it should be the last option specified.

In the evaluation of the expression arguments to various **trace** options such as :cond and :break, the value of **arglist** is a list of the arguments given to the traced function. Thus
```
(trace (foo :break (null (car arglist))))
```
would cause a break in **foo** if and only if the first argument to **foo** is **nil**. If the :break option is used, the variable **arglist** is valid inside the break-loop. If you **setq arglist** before actual function execution, the arguments seen by the function will change.

In the evaluation of the expression arguments to various **trace** options such as :cond and :break on exit from the traced function, the variable **values** is bound to a list of the resulting values of the traced function. If the :exitbreak option is used, the variables **values** and **arglist** are valid inside the break-loop. If you **setq values**, the values returned by the function will change.

The trace specifications may be "factored", as explained above. For example,
```
(trace ((foo bar) :break (bad-p arglist) :value))
```
is equivalent to
```
(trace (foo :break (bad-p arglist) :value)
       (bar :break (bad-p arglist) :value))
```
Since a list as a function name is interpreted as a list of functions, non-atomic function names (see section 11.2, page 223) are specified as follows:
```
(trace (:function (:method flavor :message) :break t))
```

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

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

Tracing is implemented with encapsulation (see section 11.9, page 244), so if the function is redefined (e.g. with **defun** or by loading it from a QFASL file) the tracing will be transferred from the old definition to the new definition.

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

untrace *Special form*

Undoes the effects of **trace** and restores functions to their normal, untraced state. **untrace** accepts multiple specifications, e.g. (**untrace foo quux fuphoo**). Calling **untrace** with no arguments will untrace all functions currently being traced.

trace-compile-flag *Variable*

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

See also the function compile-encapsulations, page 302.

30.9 Breakon

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

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

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

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

breakon with no arguments returns a list of the functions that are broken on.

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

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

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

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

rather than a conditional breakon.

To break only the *n*'th time foo is called, do

```
(defvar i n)
(breakon 'foo '(zerop (decf i)))
```

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

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

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

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

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

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

To cause the encapsulation which implements the breakon to be compiled, call **compile-encapsulations** or set **compile-encapsulations-flag** non-nil. See page 302. This may eliminate some of the problems that occur if you breakon a function such as **prog** that is used by the evaluator. (A conditional to trap only in one stack group will help here also.)

30.10 Advising a Function

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

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

Advice can be used for testing out a change to a function in a way that is easy to retract. In this case, you would call **advise** from the terminal. It can also be used for customizing a function that is part of a program written by someone else. In this case you would be likely to put a call to **advise** in one of your source files or your login init file (see page 801), rather than modifying the other person's source code.

Advising is implemented with encapsulation (see section 11.9, page 244), so if the function is redefined (e.g. with **defun** or by loading it from a QFASL file) the advice will be transferred from the old definition to the new definition.

advise *Macro*

A function is advised by the special form

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

None of this is evaluated. *function* is the function to put the advice on. It is usually a symbol, but any function spec is allowed (see section 11.2, page 223). The *forms* are the advice; they get evaluated when the function is called. *class* should be either **:before**, **:after**, or **:around**, and says when to execute the advice (before, after, or around the execution of the definition of the function). The meaning of **:around** advice is explained a couple of sections below.

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

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

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

For example,

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

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

advise with no arguments returns a list of advised functions.

unadvise *Macro*

(unadvise *function class position*)

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

If some of the arguments are missing or **nil**, all pieces of advice which match the non-**nil** arguments are removed. Thus, if *function* is missing or **nil**, all advice on all functions which match the specified *class* and *position* are removed. If *position* is missing or **nil**, then all advice of the specified class on the specified function is removed. If only *function* is non-**nil**, all advice on that function is removed.

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

si:advise-1 *function class name position forms*
Adds advice. The arguments have the same meaning as in **advise**. Note that the *forms* argument is *not* a **&rest** argument.

si:unadvise-1 &optional *function class position*
Removes advice. If *function* or *class* or *position* is **nil** or unspecified, advice is removed from all functions or all classes of advice or advice at all positions are removed.

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

To cause the advice to be compiled, call **compile-encapsulations** or set **compile-encapsulations-flag** non-nil. See page 302.

30.10.1 Designing the Advice

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

The list of the arguments to the function can be found in the variable **arglist**. **:before** advice can replace this list, or an element of it, to change the arguments passed to the definition itself. If you replace an element, it is wise to copy the whole list first with
```
    (setq arglist (copylist arglist))
```
After the function's definition has been executed, the list of the values it returned can be found in the variable **values**. **:after** advice can set this variable or replace its elements to cause different values to be returned.

All the advice is executed within a **block nil** so any piece of advice can exit the entire function with **return**. The arguments of the **return** are returned as the values of the function and no further advice is executed. If a piece of **:before** advice does this then the function's definition is not even called.

30.10.2 :around Advice

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

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

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

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

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

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

will work reasonably.

30.10.3 Advising One Function Within Another

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

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

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

advise-within *Macro*

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

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

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

unadvise-within *Macro*

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

removes advice that has been placed on (:within *within-function function-to-advise*). Any of the four arguments may be missing or **nil**; then that argument is unconstrained. All advice matching whichever arguments are non-**nil** is removed. For example, **(unadvise-within foo nil :before)** removes all :before-advice from anything within **foo**. **(unadvise-within)** removes all advice placed on anything within anything. By contrast, **(unadvise)** removes all advice, including advice placed on a function for all callers. Advice placed on a function not within another specific function is never removed by **unadvise-within**.

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

30.11 Stepping Through an Evaluation

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

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

step *form*

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

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

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

to evaluate the form **(foo t 3)** with single stepping.

The other way to get into the stepper is to use the **:step** option of **trace** (see page 738). If a function is traced with the **:step** option, then whenever that function is called it will be single stepped.

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

When evaluation is proceeding with single stepping, before any form is evaluated, it is (partially) printed out, preceded by a forward arrow (→) character When a macro is expanded, the expansion is printed out preceded by a double arrow (↔) character. When a form returns a value, the form and the values are printed out preceded by a backwards arrow (←) character; if there is more than one value being returned, an and-sign (∧) character is printed between the values. When the stepper has evaluated the args to a form and is about to apply the function, it prints a lambda (λ) because entering the lambda is the next thing to be done.

Since the forms may be very long, the stepper does not print all of a form; it truncates the printed representation after a certain number of characters. Also, to show the recursion pattern of who calls whom in a graphic fashion, it indents each form proportionally to its level of recursion.

After the stepper prints any of these things, it waits for a command from the user. There are several commands to tell the stepper how to proceed, or to look at what is happening. The commands are:

Control-N (Next)
: Steps to the Next event, then asks for another command. Events include beginning to evaluate a form at any level or finishing the evaluation of a form at any level.

Space
: Steps to the next event at this level. In other words, continue to evaluate at this level, but don't step anything at lower levels. This is a good way to skip over parts of the evaluation that don't interest you.

Control-A (Args)
: Skips over the evaluation of the arguments of this form, but pauses in the stepper before calling the function that is the car of the form.

Control-U (Up)
: Continues evaluating until we go up one level. This is like the space command, only more so; it skips over anything on the current level as well as lower levels.

Control-X (eXit)
: Exits; finishes evaluation without any more stepping.

Control-T (Type)
: Retypes the current form in full (without truncation).

Control-G (Grind)
: Grinds (i.e. prettyprints) the current form.

Control-E (Editor)
: Switches windows, to the editor.

Control-B (Breakpoint)
: Enters a break loop from which you can examine the values of variables and other aspects of the current environment. From within this loop, the following variables are available:

step-form	the current form.
step-values	the list of returned values.
step-value	the first returned value.

If you change the values of these variables, you will affect execution.

Control-L
: Clears the screen and redisplays the last 10. pending forms (forms that are being evaluated).

Meta-L
: Like Control-L, but doesn't clear the screen.

Control-Meta-L
: Like Control-L, but redisplays all pending forms.

? or Help
: Prints documentation on these commands.

It is strongly suggested that you write some little function and try the stepper on it. If you get a feel for what the stepper does and how it works, you will be able to tell when it is the right thing to use to find bugs.

30.12 Evalhook

The evalhook facility provides a "hook" into the evaluator; it is a way you can get a Lisp form of your choice to be executed whenever the evaluator is called. The stepper uses evalhook, and usually it is the only thing that ever needs to. However, if you want to write your own stepper or something similar, this is the primitive facility that you can use to do so. The way this works is a bit hairy, but unless you need to write your own stepper you don't have to worry about it.

evalhook *Variable*
evalhook *Variable*

If the value of evalhook is non-nil, then special things happen in the evaluator. Its value is called the *hook function*. When a form (any form, even a number or a symbol) is to be evaluated, the hook function is called instead. Whatever values the hook function returns are taken to be the results of the evaluation. Both evalhook and applyhook are bound to nil before the hook function is actually called.

The hook function receives two arguments: the form that was to be evaluated, and the lexical environment of evaluation. These two arguments allow the hook function to perform later, if it wishes, the very same evaluation that the hook was called instead of.

applyhook *Variable*
applyhook *Variable*

If the value of applyhook is non-nil, it is called the next time the interpreter is about to apply a function to its evaluated arguments. Whatever values the apply hook function returns are taken to be the results of calling the other function. Both evalhook and applyhook are bound to nil before the hook function is actually called.

The hook function receives three arguments: the function that was going to be called, the list of arguments it was going to receive, and the lexical environment of evaluation. These arguments allow the hook function to perform later, if it wishes, the very same evaluation that the hook was called instead of.

When either the evalhook or the applyhook is called, both variables are bound to nil. They are also rebound to nil by break and by the debugger, and setq'ed to nil when errors are dismissed by throwing to the Lisp top level loop. This provides the ability to escape from this mode if something bad happens.

In order not to impair the efficiency of the Lisp interpreter, several restrictions are imposed on the evalhook and applyhook. They apply only to evaluation—whether in a read-eval-print loop, internally in evaluating arguments in forms, or by explicit use of the function eval. They *do not* have any effect on compiled function references, on use of the function apply, or on the mapping functions.

evalhook *form evalhook applyhook* &optional *environment*

Evaluates *form* in the specified *environment*, with *evalhook* and *applyhook* in effect for all recursive evaluations of subforms of *form*. However, the *evalhook* is not called for the evaluation of *form* itself.

environment is a list which represents the lexical environment to be in effect for the evaluation of *form*. **nil** means an empty lexical environment, in which no lexical bindings exist. This is the environment used when **eval** itself is called. Aside from **nil**, the only reasonable way to get a value to pass for *environment* is to use the last argument passed to a hook function. You must take care not to use it after the context in which it was made is exited, because environments normally contain stack lists which become garbage after their stack frames are popped.

environment has no effect on the evaluation of a variable which is regarded as special. This is always done by examining the value cell. However, environment contains the record of the local special declarations currently in effect, so it does enter in the decision of whether a variable is special.

Here is an example of the use of **evalhook**:

```
;; This function evaluates a form while printing debugging information.
(defun hook (x)
   (terpri)
   (evalhook x 'hook-function nil))

;; Notice how this function calls evalhook to evaluate the form f,
;; so as to hook the sub-forms.
(defun hook-function (f env)
   (let ((v (multiple-value-list
                (evalhook f 'hook-function nil env))))
     (format t "form: ~S~%values: ~S~%" f v)
     (values-list v)))
```

The following output might be seen from **(hook '(cons (car '(a . b)) 'c))**:

```
form: (quote (a . b))
values: ((a . b))
form: (car (quote (a . b)))
values: (a)
form: (quote c)
values: (c)
(a . c)
```

applyhook *function list-of-args evalhook applyhook* &optional *environment*

Applies *function* to *list-of-args* in the specified *environment*, with *evalhook* and *applyhook* in effect for all recursive evaluations of subforms of *function*'s body. However, *applyhook* is not called for this application of function itself. For more information, refer to the definition of **evalhook**, immediately above.

30.13 The MAR

The MAR facility allows any word or contiguous set of words to be monitored constantly, and can cause an error if the words are referenced in a specified manner. The name MAR is from the similar device on the ITS PDP-10's; it is an acronym for 'Memory Address Register'. The MAR checking is done by the Lisp Machine's memory management hardware, so the speed of general execution is not significantly slowed down when the MAR is enabled. However, the speed of accessing pages of memory containing the locations being checked is slowed down somewhat, since every reference involves a microcode trap.

These are the functions that control the MAR:

set-mar *location cycle-type* &optional *n-words*

Sets the MAR on *n-words* words, starting at *location*. *location* may be any object. Often it will be a locative pointer to a cell, probably created with the locf special form. *n-words* currently defaults to 1, but eventually it may default to the size of the object. *cycle-type* says under what conditions to trap. :read means that only reading the location should cause an error, :write means that only writing the location should, t means that both should. To set the MAR to detect setq (and binding) of the variable foo, use

```
(set-mar (variable-location foo) :write)
```

clear-mar

Turns off the MAR. Warm-booting the machine disables the MAR but does not turn it off, i.e. references to the MARed pages are still slowed down. clear-mar does not currently speed things back up until the next time the pages are swapped out; this may be fixed some day.

mar-mode

(mar-mode) returns a symbol indicating the current state of the MAR. It returns one of:

nil	The MAR is not set.
:read	The MAR will cause an error if there is a read.
:write	The MAR will cause an error if there is a write.
t	The MAR will cause an error if there is any reference.

Note that using the MAR makes the pages on which it is set somewhat slower to access, until the next time they are swapped out and back in again after the MAR is shut off. Also, use of the MAR currently breaks the read-only feature if those pages were read-only.

Proceeding from a MAR break allows the memory reference that got an error to take place, and continues the program with the MAR still effective. When proceeding from a write, you have the choice of whether to allow the write to take place or to inhibit it, leaving the location with its old contents.

sys:mar-break (condition) *Condition*

This is the condition, not an error, signaled by a MAR break.

The condition instance supports these operations:

:object	The object one of whose words was being referenced.
:offset	The offset within the object of the word being referenced.
:value	The value read, or to be written.
:direction	Either :read or :write.

The proceed type :no-action simply proceeds, continuing with the interrupted program as if the MAR had not been set. If the trap was due to writing, the proceed type :proceed-no-write is also provided, and causes the program to proceed but does not store the value in the memory location.

Most—but not all—write operations first do a read. setq and rplaca both do. This means that if the MAR is in :read mode it catches writes as well as reads; however, they trap during the reading phase, and consequently the data to be written are not yet known. This also means that setting the MAR to t mode causes most writes to trap twice, first for a read and then again for a write. So when the MAR says that it trapped because of a read, this means a read at the hardware level, which may not look like a read in your program.

31. How to Read Assembly Language

Sometimes it is useful to study the machine language code produced by the Lisp Machine's compiler, usually in order to analyze an error, or sometimes to check for a suspected compiler problem. This chapter explains how the Lisp Machine's instruction set works and how to understand what code written in that instruction set is doing. Fortunately, the translation between Lisp and this instruction set is very simple: after you get the hang of it, you can move back and forth between the two representations without much trouble. The following text does not assume any special knowledge about the Lisp Machine, although it sometimes assumes some general computer science background knowledge.

31.1 Introduction

Nobody looks at machine language code by trying to interpret octal numbers by hand. Instead, there is a program called the Disassembler which converts the numeric representation of the instruction set into a more readable textual representation. It is called the Disassembler because it does the opposite of what an Assembler would do; however, there isn't actually any assembler that accepts this input format, since there is never any need to manually write assembly language for the Lisp Machine.

The simplest way to invoke the Disassembler is with the Lisp function **disassemble**. Here is a simple example. Suppose we type:

```
(defun foo (x)
  (assq 'key (get x 'propname)))

(compile 'foo)

(disassemble 'foo)
```

This defines the function **foo**, compiles it, and invokes the Disassembler to print out the textual representation of the result of the compilation. Here is what it looks like:

```
22 MOVE D-PDL FEF|6             ;'KEY
23 MOVE D-PDL ARG|0             ;X
24 MOVE D-PDL FEF|7             ;'PROPNAME
25 (MISC) GET D-PDL
26 (MISC) ASSQ D-RETURN
```

The Disassembler is also used by the Error Handler and the Inspector. When you see stuff like the above while using one of these programs, it is disassembled code, in the same format as the **disassemble** function uses. Inspecting a compiled code object shows the disassembled code.

Now, what does this mean? Before we get started, there is just a little bit of jargon to learn.

The acronym PDL stands for Push Down List, and means the same thing as Stack: a last-in first-out memory. The terms PDL and stack will be used interchangeably. The Lisp Machine's architecture is rather typical of "stack machines"; there is a stack that most instructions deal with, and it is used to hold values being computed, arguments, and local variables, as well as flow-of-control information. An important use of the stack is to pass arguments to instructions, though not all instructions take their arguments from the stack.

The acronym 'FEF' stands for Function Entry Frame. A FEF is a compiled code object produced by the compiler. After the defun form above was evaluated, the function cell of the symbol foo contained a lambda expression. Then we compiled the function foo, and the contents of the function cell were replaced by a FEF. The printed representation of the FEF for foo looks like this:

```
    #<DTP-FEF-POINTER 11464337 FOO>
```

The FEF has three parts (this is a simplified explanation): a header with various fixed-format fields; a part holding constants and invisible pointers, and the main body, holding the machine language instructions. The first part of the FEF, the header, is not very interesting and is not documented here (you can look at it with describe but it won't be easy to understand). The second part of the FEF holds various constants referred to by the function; for example, our function foo references two constants (the symbols key and propname), and so (pointers to) those symbols are stored in the FEF. This part of the FEF also holds invisible pointers to the value cells of all symbols that the function uses as variables, and invisible pointers to the function cells of all symbols that the function calls as functions. The third part of the FEF holds the machine language code itself.

Now we can read the disassembled code. The first instruction looked like this:

```
    22 MOVE D-PDL FEF|6              ;'KEY
```

This instruction has several parts. The 22 is the address of this instruction. The Disassembler prints out the address of each instruction before it prints out the instruction, so that you can interpret branching instructions when you see them (we haven't seen one of these yet, but we will later). The MOVE is an opcode: this is a MOVE instruction, which moves a datum from one place to another. The D-PDL is a destination specification. The D stands for 'Destination', and so D-PDL means 'Destination-PDL': the destination of the datum being moved is the PDL. This means that the result will be pushed onto the PDL, rather than just moved to the top; this instruction is pushing a datum onto the stack. The next field of the instruction is FEF|6. This is an *address*, and it specifies where the datum is coming from. The vertical bar serves to separate the two parts of the address. The part before the vertical bar can be thought of as a *base register*, and the part after the bar can be thought of as being an offset from that register. FEF as a base register means the address of the FEF that we are disassembling, and so this address means the location six words into the FEF. So what this instruction does is to take the datum located six words into the FEF, and push it onto the PDL. The instruction is followed by a comment field, which looks like ;'KEY. This is not a comment that any person wrote; the disassembler produces these to explain what is going on. The semicolon just serves to start the comment, the way semicolons in Lisp code do. In this case, the body of the comment, 'KEY, is telling us that the address field (FEF|6) is addressing a constant (that is what the single-quote in 'KEY means), and that the printed representation of that constant is KEY. With the help of this comment we finally get the real story about what this instruction is doing: it is pushing (a pointer to) the symbol key onto the stack.

The next instruction looks like this:

```
    23 MOVE D-PDL ARG|0              ;X
```

This is a lot like the previous instruction: the only difference is that a different "base register" is being used in the address. The ARG base register is used for addressing your arguments: ARG|0 means that the datum being addressed is the zeroth argument. Again, the comment field explains what that means: the value of X (which was the zeroth argument) is being pushed onto the stack.

The third instruction is just like the first and second ones; it pushes the symbol propname onto the stack.

The fourth instruction is something new:

```
    25 (MISC) GET D-PDL
```

The first thing we see here is (MISC). This means that this is one of the so-called *miscellaneous* instructions. There are quite a few of these instructions. With some exceptions, each miscellaneous instruction corresponds to a Lisp function and has the same name as that Lisp function. If a Lisp function has a corresponding miscellaneous instruction, then that function is hand-coded in Lisp Machine microcode.

Miscellaneous instructions only have a destination field; they don't have any address field. The inputs to the instruction come from the stack: the top *n* elements on the stack are used as inputs to the instruction and popped off the stack, where *n* is the number of arguments taken by the function. The result of the function is stored wherever the destination field says. In our case, the function being executed is get, a Lisp function of two arguments. The top two values will be popped off the stack and used as the arguments to get (the value pushed first is the first argument, the value pushed second is the second argument, and so on). The result of the call to get will be sent to the destination D-PDL; that is, it will be pushed onto the stack. (In case you were wondering about how we handle optional arguments and multiple-value returns, the answer is very simple: functions that use either of those features cannot be miscellaneous instructions! If you are curious as to what functions are hand-microcoded and thus available as miscellaneous instructions, you can look at the defmic forms in the file SYS: SYS; DEFMIC LISP.)

The fifth and last instruction is similar to the fourth:

```
    26 (MISC) ASSQ D-RETURN
```

What is new here is the new value of the destination field. This one is called D-RETURN, and it can be used anywhere destination fields in general can be used (like in MOVE instructions). Sending something to "Destination-Return" means that this value should be the returned value of the function, and that we should return from this function. This is a bit unusual in instruction sets; rather than having a "return" instruction, we have a destination that, when stored into, returns from the function. What this instruction does, then, is to invoke the Lisp function assq on the top two elements of the stack and return the result of assq as the result of this function.

Now, let's look at the program as a whole and see what it did:

```
22 MOVE D-PDL FEF|6              ;'KEY
23 MOVE D-PDL ARG|0              ;X
24 MOVE D-PDL FEF|7              ;'PROPNAME
25 (MISC) GET D-PDL
26 (MISC) ASSQ D-RETURN
```

First it pushes the symbol key. Then it pushes the value of x. Then it pushes the symbol propname. Then it invokes get, which pops the value of x and the symbol propname off the stack and uses them as arguments, thus doing the equivalent of evaluating the form (get x 'propname). The result is left on the stack; the stack now contains the result of the get on top, and the symbol key underneath that. Next, it invokes assq on these two values, thus doing the equivalent of evaluating (assq 'key (get x 'propname)). Finally, it returns the value produced by assq. Now, the original Lisp program we compiled was:

```
(defun foo (x)
  (assq 'key (get x 'propname)))
```

We can see that the code produced by the compiler is correct: it will do the same thing as the function we defined will do.

In summary, we have seen two kinds of instructions so far: the MOVE instruction, which takes a destination and an address, and two of the large set of miscellaneous instructions, which take only a destination, and implicitly get their inputs from the stack. We have seen two destinations (D-PDL and D-RETURN), and two forms of address (FEF addressing and ARG addressing).

31.2 A More Advanced Example

Here is a more complex Lisp function, demonstrating local variables, function calling, conditional branching, and some other new instructions.

```
(defun bar (y)
  (let ((z (car y)))
    (cond ((atom z)
           (setq z (cdr y))
           (foo y))
          (t
           nil))))
```

The disassembled code looks like this:

```
20 CAR D-PDL ARG|0            ;Y
21 POP LOCAL|0                ;Z
22 BR-NOT-ATOM 27
23 CDR D-PDL ARG|0            ;Y
24 POP LOCAL|0                ;Z
25 CALL D-RETURN FEF|6        ;#'FOO
26 MOVE D-LAST ARG|0          ;Y
27 MOVE D-RETURN 'NIL
```

The first instruction here is a CAR instruction. It has the same format as MOVE: there is a destination and an address. The CAR instruction reads the datum addressed by the address, takes the car of it, and stores the result into the destination. In our example, the first instruction addresses the zeroth argument, and so it computes (car y); then it pushes the result onto the stack.

The next instruction is something new: the POP instruction. It has an address field, but it uses it as a destination rather than as a source. The POP instruction pops the top value off the stack, and stores that value into the address specified by the address field. In our example, the value on the top of the stack is popped off and stored into address LOCAL|0. This is a new form of address; it means the zeroth local variable. The ordering of the local variables is chosen by the compiler, and so it is not fully predictable, although it tends to be by order of appearance in the code; fortunately you never have to look at these numbers, because the comment field explains what is going on. In this case, the variable being addressed is z. So this instruction pops the top value on the stack into the variable z. The first two instructions work together to take the car of y and store it into z, which is indeed the first thing the function bar ought to do. (If you have two local variables with the same name, then the comment field won't tell you which of the two you're talking about; you'll have to figure that out yourself. You can tell two local variables with the same name apart by looking at the number in the address.)

The next instruction is a familiar MOVE instruction, but it uses a new destination: D-IGNORE. This means that the datum being addressed isn't moved anywhere. If so, then why bother doing this instruction? The reason is that there is conceptually a set of *indicator* bits, as are found in most modern computers such as the 68000, the Vax, as well as in obsolete computers such as the 370. Every instruction that moves or produces a datum sets the indicator bits from that datum so that following instructions can test them. So the reason that the MOVE instruction is being done is so that someone can test the indicators set up by the value that was moved, namely the value of **z**.

All instructions except the branch instructions set the indicator bits from the result produced and/or stored by that instruction.

The next instruction is a conditional branch; it changes the flow of control, based on the values in the indicator bits, which in this case reflect the value popped by the POP instruction 21. The branch instruction is BR-NOT-ATOM 27, which means "Branch, if the quantity was not an atom, to location 27; otherwise proceed with execution". If z was an atom, the Lisp Machine branches to location 27, and execution proceeds there. (As you can see by skipping ahead, location 27 just contains a MOVE instruction, which will cause the function to return **nil**.)

If z is not an atom, the program keeps going, and the CDR instruction is next. This is just like the CAR instruction except that it takes the cdr; this instruction pushes the value of (cdr y) onto the stack. The next one pops that value off into the variable z.

There are just two more instructions left. These two instructions are our first example of how function calling is compiled. It is the only really tricky thing in the instruction set. Here is how it works in our example:

```
25 CALL D-RETURN FEF|6          ;#'FOO
26 MOVE D-LAST ARG|0            ;Y
```

The form being compiled here is (foo y). This means we are applying the function which is in the function cell of the symbol foo, and passing it one argument, the value of y. The way function calling works is in the following three steps. First of all, there is a CALL instruction that specifies the function object being applied to arguments. This creates a new stack frame on the stack, and stores the function object there. Secondly, all the arguments being passed except the last one are pushed onto the stack. Thirdly and lastly, the last argument is sent to a special destination, called D-LAST, meaning "this is the last argument". Storing to this destination is what actually calls the function, *not* the CALL instruction itself.

There are two things you might wonder about this. First of all, when the function returns, what happens to the returned value? Well, this is what we use the destination field of the CALL instruction for. The destination of the CALL is not stored into at the time the CALL instruction is executed; instead, it is saved on the stack along with the function operation (in the stack frame created by the CALL instruction). Then, when the function actually returns, its result is stored into that destination.

The other question is what happens when there isn't any last argument; that is, when there is a call with no arguments at all? This is handled by a special instruction called CALL0. The address of CALL0 addresses the function object to be called; the call takes place immediately and the result is stored into the destination specified by the destination field of the CALL0 instruction.

So, let's look at the two-instruction sequence above. The first instruction is a CALL; the function object it specifies is at FEF|6, which the comment tells us is the contents of the function cell of foo (the FEF contains an invisible pointer to that function cell). The destination field of the CALL is D-RETURN, but we aren't going to store into it yet; we will save it away in the stack frame and use it later. So the function doesn't return at this point, even though it says D-RETURN in the instruction; this is the tricky part.

Next we have to push all the arguments except the last one. Well, there's only one argument, so nothing needs to be done here. Finally, we move the last argument (that is, the only argument: the value of y) to D-LAST, using the MOVE instruction. Moving to D-LAST is what actually invokes the function, so at this point the function foo is invoked. When it returns, its result is sent to the destination stored in the stack frame: D-RETURN. Therefore, the value returned by the call to foo will be returned as the value of the function bar. Sure enough, this is what the original Lisp code says to do.

When the compiler pushes arguments to a function call, it sometimes does it by sending the values to a destination called D-NEXT (meaning the "next" argument). This is exactly the same as D-PDL when producing a compiled function. The distinction is important when the compiler output is passed to the microcompiler to generate microcode.

Here is another example to illustrate function calling. This Lisp function calls one function on the results of another function.

```
(defun a (x y)
  (b (c x y) y))
```

The disassembled code looks like this:

```
22 CALL D-RETURN FEF|6          ;#'B
23 CALL D-PDL FEF|7             ;#'C
24 MOVE D-PDL ARG|0             ;X
25 MOVE D-LAST ARG|1            ;Y
26 MOVE D-LAST ARG|1            ;Y
```

The first instruction starts off the call to the function **b**. The destination field is saved for later: when this function returns, we will return its result as **a**'s result. Next, the call to **c** is started. Its destination field, too, is saved for later; when **c** returns, its result should be pushed onto the stack, so that it will be the next argument to **b**. Next, the first and second arguments to **c** are passed; the second one is sent to D-LAST and so the function **c** is called. Its result, as we said, will be pushed onto the stack, and thus become the first argument to **b**. Finally, the second argument to **b** is passed, by storing in D-LAST; **b** gets called, and its result is sent to D-RETURN and is returned from **a**.

31.3 The Rest of the Instructions

Now that we've gotten some of the feel for what is going on, I will start enumerating the instructions in the instruction set. The instructions fall into four classes. Class I instructions have both a destination and an address. Class II instructions have an address, but no destination. Class III instructions are the branch instructions, which contain a branch address rather than a general base-and-offset address. Class IV instructions have a destination, but no address; these are the miscellaneous instructions.

We have already seen just about all the Class I instructions. There are nine of them in all: MOVE, CALL, CALL0, CAR, CDR, CAAR, CADR, CDAR, and CDDR. MOVE just moves a datum from an address to a destination; the CxR and CxxR instructions are the same but perform the function on the value before sending it to the destination; CALL starts off a call to a function with some arguments; CALL0 performs a call to a function with no arguments.

We've seen most of the possible forms of address. So far we have seen the FEF, ARG, and LOCAL base registers. There are two other kinds of addresses. One uses a "constant" base register, which addresses a set of standard constants: NIL, T, 0, 1, and 2. The disassembler doesn't even bother to print out CONSTANT|*n*, since the number *n* would not be even slightly interesting; it just prints out 'NIL or '1 or whatever. The other kind of address is a special one printed as PDL-POP, which means that to read the value at this address, an object should be popped off the top of the stack.

There are more Class II instructions. The only one we've seen so far is POP, which pops a value off the stack and stores it into the specified address. Another, called MOVEM (from the PDP-10 opcode name, meaning MOVE to Memory), stores the top element of the stack into the specified address, but doesn't pop it off the stack.

Seven Class II instructions implement heavily-used two-argument functions: +, -, *, /, LOGAND, LOGXOR, and LOGIOR. These instructions take the first argument from the top of the stack (popping it off) and their second argument from the specified address, and they push the result on the stack. Thus the stack level does not change due to these instructions. Here is a small function that shows some of these new things:

```
(defun foo (x y)
  (setq x (logxor y (- x 2))))
```

The disassembled code looks like this:

```
16 MOVE D-PDL ARG|1             ;Y
17 MOVE D-PDL ARG|0             ;X
20 - '2
21 LOGXOR PDL-POP
22 MOVEM ARG|0                  ;X
23 MOVE D-RETURN PDL-POP
```

Instructions 20 and 21 use two of the new Class II instructions: the - and LOGXOR instructions. Instructions 21 and 23 use the PDL-POP address type, and instruction 20 uses the "constant" base register to get to a fixnum 2. Finally, instruction 22 uses the MOVEM instruction; the compiler wants to use the top value of the stack to store it into the value of x, but it doesn't want to pop it off the stack because it has another use for it: to return it from the function.

Another four Class II instructions implement some commonly used predicates: =, >, <, and EQ. The two arguments come from the top of the stack and the specified address; the stack is popped, the predicate is applied to the two objects, and the result is left in the indicators so that a branch instruction can test it, and branch based on the result of the comparison. These instructions remove the top item on the stack and don't put anything back, unlike the previous set, which put their results back on the stack.

Next, there are four Class II instructions to read, modify, and write a quantity in ways that are common in Lisp code. These instructions are called SETE-CDR, SETE-CDDR, SETE-1+, and SETE-1-. The SETE- means to set the addressed value to the result of applying the specified one-argument function to the present value. For example, SETE-CDR means to read the value addressed, apply cdr to it, and store the result back in the specified address. This is used when compiling (setq x (cdr x)), which commonly occurs in loops; the other functions are used frequently in loops, too.

There are two instructions used to bind special variables. The first is BIND-NIL, which binds the cell addressed by the address field to nil; the second is BIND-POP, which binds the cell to an object popped off the stack rather than nil. The latter instruction pops a value off the stack; the former does not use the stack at all.

There are two instructions to store common values into addressed cells. SET-NIL stores nil into the cell specified by the address field; SET-ZERO stores 0. Neither instruction uses the stack at all.

Finally, the PUSH-E instruction creates a locative pointer to the cell addressed by the specified address, and pushes it onto the stack. This is used in compiling (value-cell-location 'z) where z is an argument or a local variable, rather than a symbol (special variable).

Those are all of the Class II instructions. Here is a contrived example that uses some of the ones we haven't seen, just to show you what they look like:

```
(defun weird (x y)
  (cond ((= x y)
         (let ((*foo* nil) (*bar* 5))
           (declare (special *foo* *bar*))
           (setq x (cdr x)))
         nil)
        (t
         (setq x nil)
         (caar (variable-location y)))))
```

The disassembled code looks like this:

```
24 MOVE D-PDL ARG|0          ;X
25 = ARG|1                   ;Y
26 BR-NIL 35
27 BIND-NIL FEF|6            ;*FOO*
30 PUSH-NUMBER 5
31 BIND-POP FEF|7            ;*BAR*
32 SETE-CDR ARG|0            ;X
33 (MISC) UNBIND 2 bindings
34 MOVE D-RETURN 'NIL
35 SET-NIL ARG|0             ;X
36 PUSH-E ARG|1              ;Y
37 CAAR D-RETURN PDL-POP
```

Instruction 25 is an = instruction; it numerically compares the top of the stack, x, with the addressed quantity, y. The x is popped off the stack, and the indicators are set to the result of the equality test. Instruction 26 checks the indicators, branching to 35 if the result of the call to = was nil; that is, the machine will branch to 35 if the two values were not equal. Instruction 27 binds *foo* to nil; instructions 30 and 31 bind *bar* to 5. Instruction 30 is a peculiar class IV instruction called PUSH-NUMBER which pushes a constant integer on the stack. The integer must be in the range of zero to 511 in order for PUSH-NUMBER to be used. Instruction 32 demonstrates the use of SETE-CDR to compile (setq x (cdr x)), and instruction 35 demonstrates the use of SET-NIL to compile (setq x nil). Instruction 36 demonstrates the use of PUSH-E to compile (variable-location y).

The Class III instructions are for branching. These have neither addresses nor destinations of the usual sort. Instead, they have branch-addresses; they say where to branch, if the branch is going to happen. There are several instructions, differing in the conditions under which they branch and whether they pop the stack. Branch-addresses are stored internally as self-relative addresses, to make Lisp Machine code relocatable, but the disassembler does the addition for you and prints out FEF-relative addresses so that you can easily see where the branch is going to.

The branch instructions we have seen so far decide whether to branch on the basis of the nil-indicator, that is, whether the last value dealt with was nil or non-nil. BR-NIL branches if it was nil, and BR-NOT-NIL branches if it was not nil. There are two more instructions that test the result of the atom predicate on the last value dealt with. BR-ATOM branches if the value was an atom (that is, if it was anything besides a cons), and BR-NOT-ATOM branches if the value was not an atom (that is, if it was a cons). The BR instruction is an unconditional branch (it always branches).

None of the above branching instructions deal with the stack. There are two more instructions called BR-NIL-POP and BR-NOT-NIL-POP, which are the same as BR-NIL and BR-NOT-NIL except that if the branch is not done, the top value on the stack is popped off the stack. These are used for compiling and and or special forms.

Finally, there are the Class IV instructions, most of which are miscellaneous hand-microcoded Lisp functions. The file SYS: SYS; DEFMIC LISP has a list of all the miscellaneous instructions. Most correspond to Lisp functions, including the subprimitives, although some of these functions are very low level internals that may not be documented anywhere (don't be disappointed if you don't understand all of them). Please do not look at this file in hopes of finding obscure functions that you think you can use to speed up your programs; in fact, the compiler automatically uses these things when it can, and directly calling weird internal functions will only serve to make your code hard to read, without making it any faster. In fact, we don't guarantee that calling undocumented functions will continue to work in the future.

The DEFMIC file can be useful for determining if a given function is in microcode, although the only definitive way to tell is to compile some code that uses it and look at the results, since sometimes the compiler converts a documented function with one name into an undocumented one with another name.

31.4 Function Entry

When a function is first entered in the Lisp Machine, interesting things can happen because of the features that are invoked by use of the various lambda-list keywords. The microcode performs various services when a function is entered, even before the first instruction of the function is executed. These services are called for by various fields of the header portion of the FEF, including a list called the *Argument Descriptor List*, or *ADL*. We won't go into the detailed format of any of this, as it is complex and the details are not too interesting. Disassembling a function that makes use of the ADL prints a summary of what the ADL says to do, before the beginning of the code.

The function-entry services include the initialization of unsupplied optional arguments and of &AUX variables. The ADL has a little instruction set of its own, and if the form that computes the initial value is something simple, such as a constant or a variable, then the ADL can handle things itself. However, if things get too complicated, instructions are needed, and the compiler generates some instructions at the front of the function to initialize the unsupplied variables. In this case, the ADL specifies several different starting addresses for the function, depending on which optional arguments have been supplied and which have been omitted. If all the optional arguments are supplied, then the ADL starts the function off after all the instructions that would have initialized the optional arguments: since the arguments were supplied, their values should not be set, and so all these instructions are skipped over. Here's an example:

```
(defvar *y*)

(defun foo (&optional (x (car *y*)) (z (* x 3)))
  (cons x z))
```

The disassembled code looks like this:

```
Arg 0 (X) is optional, local,
  initialized by the code up to pc 32.
Arg 1 (Z) is optional, local,
  initialized by the code up to pc 35.

30 CAR D-PDL FEF|6              ;*Y*
31 POP ARG|0                    ;X
32 MOVE D-PDL ARG|0             ;X
33 * '3
34 POP ARG|1                    ;Z
35 MOVE D-PDL ARG|0             ;X
36 MOVE D-PDL ARG|1             ;Z
37 (MISC) CONS D-RETURN
```

If no arguments are supplied, the function will be started at instruction 30; if only one argument is supplied, it will be started at instruction 32; if both arguments are supplied, it will be started at instruction 35.

The thing to keep in mind here is that when there is initialization of variables, you may see it as code at the beginning of the function, or you may not, depending upon whether it is too complex for the ADL to handle. This is true of **&aux** variables as well as unsupplied **&optional** arguments.

When there is a **&rest** argument, it is passed to the function as the zeroth local variable, rather than as any of the arguments. This is not really so confusing as it might seem, since a **&rest** argument is not an argument passed by the caller; rather it is a list of some of the arguments, created by the function-entry microcode services. In any case the comment tells you what is going on. In fact, one hardly ever looks much at the address fields in disassembled code, since the comment tells you the right thing anyway. Here is a silly example of the use of a **&rest** argument:

```
(defun prod (&rest values)
  (apply #'* values))
```

The disassembled code looks like this:

```
20 MOVE D-PDL FEF|6          ;#'*
21 MOVE D-PDL LOCAL|0        ;VALUES
22 (MISC) APPLY D-RETURN
```

As can be seen, **values** is referred to as **LOCAL|0**.

Another thing the microcode does at function entry is to bind the values of any arguments or **&aux** variables that are special. Thus, you won't see any **BIND** instructions for binding them.

31.5 Special Class IV Instructions

We said earlier that most of the Class IV instructions are miscellaneous hand-microcoded Lisp functions. However, a few of them are not Lisp functions at all. There are two instructions that are printed as **UNBIND 3 bindings** or **POP 7 values**; the number can be anything up to 16 (these numbers are printed in decimal). These instructions just do what they say, unbinding the last *n* values that were bound or popping the top *n* values off the stack.

Another Class IV instruction is **PUSH-NUMBER**. It pushes a constant integer, in the range zero to 511. An example of it appeared on page 760.

The array referencing functions—**aref**, **aset**, and **aloc**—take a variable number of arguments, but they are handled differently depending on how many there are. For one-, two-, and three-dimensional arrays, these functions are turned into internal functions with names **ar-1**, **as-1**, and **ap-1** (with the number of dimensions substituted for **1**). Again, there is no point in using these functions yourself; it would only make your code harder to understand but not any faster at all. When there are more than three dimensions, the functions **aref**, **aset** and **aloc** are called in the ordinary manner.

```
(defun foo (y x i j &aux v)
  (setq v (aref x i j))
  (setf (aref y i) v))

16 MOVE D-PDL ARG|1          ;X
17 MOVE D-PDL ARG|2          ;I
20 MOVE D-PDL ARG|3          ;J
21 (MISC) AR-2 D-PDL
22 POP LOCAL|0               ;V
23 MOVE D-PDL ARG|0          ;Y
24 MOVE D-PDL ARG|2          ;I
25 MOVE D-PDL LOCAL|0        ;V
26 (MISC) SET-AR-1 D-RETURN
```

Reference to one-dimensional arrays with constant subscripts use special instructions which have the array index encoded instead of an address.

```
(defun foo (x)
  (+ (aref x 3) (array-leader x 2))
  (setf (aref x 5) t))

FOO:
16 MOVE D-PDL ARG|0            ;X
17 AR-1 (3) D-IGNORE
20 MOVE D-PDL ARG|0            ;X
21 ARRAY-LEADER (2) D-IGNORE
22 MOVE D-PDL ARG|0            ;X
23 MOVE D-PDL 'T
24 SET-AR-1 (5) D-RETURN
```

The AR-1 instruction is to be distinguished from the MISC AR-1 instruction. AR-1 pops an array off the stack and encodes the subscript itself. The 3 in (3) is the subscript. ARRAY-LEADER is similar but refers to an array leader slot. SET-AR-1 pops an array and then pops a value to store into it at the specified slot. SET-AR-1 is analogous. There also exist %INSTANCE-REF and SET-%INSTANCE-REF instructions.

When you call a function and expect to get more than one value back, a slightly different kind of function calling is used. Here is an example that uses multiple-value to get two values back from a function call:

```
(defun foo (x)
  (let (y z)
    (multiple-value (y z)
      (bar 3))
    (+ x y z)))
```

The disassembled code looks like this:

```
20 MOVE D-PDL FEF|6            ;#'BAR
21 MOVE D-PDL '2
22 (MISC) %CALL-MULT-VALUE D-IGNORE
23 MOVE D-LAST '3
24 POP LOCAL|1                 ;Z
25 POP LOCAL|0                 ;Y
26 MOVE D-PDL ARG|0            ;X
27 + LOCAL|0                   ;Y
30 + LOCAL|1                   ;Z
31 MOVE D-RETURN PDL-POP
```

A %CALL-MULT-VALUE instruction is used instead of a CALL instruction. The destination field of %CALL-MULT-VALUE is unused and will always be D-IGNORE. %CALL-MULT-VALUE takes two "arguments", which it finds on the stack; it pops both of them. The first one is the function object to be applied; the second is the number of return values that are expected.

The rest of the call proceeds as usual, but when the call returns, the returned values are left on the stack. The number of objects left on the stack is always the same as the second "argument" to %CALL-MULT-VALUE. In our example, the two values returned are left on the stack, and they are immediately popped off into z and y. There is also a %CALLO-MULT-VALUE instruction, for the same reason CALLO exists.

The multiple-value-bind form works similarly; here is an example:

```
(defun foo (x)
  (multiple-value-bind (y *foo* z)
      (bar 3)
    (declare (special *foo*))
    (+ x y z)))
```

The disassembled code looks like this:

```
22 MOVE D-PDL FEF|7              ;#'BAR
23 MOVE D-PDL '3
24 (MISC) %CALL-MULT-VALUE D-IGNORE
25 MOVE D-LAST '3
26 POP LOCAL|1                   ;Z
27 BIND-POP FEF|6                ;*FOO*
30 POP LOCAL|0                   ;Y
31 MOVE D-PDL ARG|0              ;X
32 + LOCAL|0                     ;Y
33 + LOCAL|1                     ;Z
34 MOVE D-RETURN PDL-POP
```

The %CALL-MULT-VALUE instruction is still used, leaving the results on the stack; these results are used to bind the variables.

Calls done with multiple-value-list work with the %CALL-MULT-VALUE-LIST instruction. It takes one "argument" on the stack: the function object to apply. When the function returns, the list of values is left on the top of the stack. Here is an example:

```
(defun foo (x y)
  (multiple-value-list (bar -7 y x)))
```

The disassembled code looks like this:

```
22 MOVE D-PDL FEF|6              ;#'BAR
23 (MISC) %CALL-MULT-VALUE-LIST D-IGNORE
24 MOVE D-PDL FEF|7              ;'-7
25 MOVE D-PDL ARG|1              ;Y
26 MOVE D-LAST ARG|0             ;X
27 MOVE D-RETURN PDL-POP
```

Returning of more than one value from a function is handled by special miscellaneous instructions. %RETURN-2 and %RETURN-3 are used to return two or three values; these instructions take two and three arguments, respectively, on the stack and return from the current function just as storing to D-RETURN would. If there are more than three return values, they are all pushed, then the number that there were is pushed, and then the %RETURN-N instruction is executed. None of these instructions use their destination field. Note: the **return-list** function is just an ordinary miscellaneous instruction; it takes the list of values to return as an argument on the stack and returns those values from the current function.

The function **apply** is compiled using a special instruction called %SPREAD to iterate over the elements of its last argument, which should be a list. %SPREAD takes one argument (on the stack), which is a list of values to be passed as arguments (pushed on the stack). If the destination of %SPREAD is D-PDL (or D-NEXT), then the values are just pushed; if it is D-LAST, then after the values are pushed, the function is invoked. **apply** with more than two arguments will always compile using a %SPREAD whose destination is D-LAST. Here is an example:

```
(defun foo (a b &rest c)
  (apply #'format t a c)
  b)
```

The disassembled code looks like this:

```
FOO:
20 CALL D-IGNORE FEF|6          ;#'FORMAT
21 MOVE D-PDL 'T
22 MOVE D-PDL ARG|0             ;A
23 MOVE D-PDL LOCAL|0           ;C
24 (MISC) %SPREAD D-LAST
25 MOVE D-RETURN ARG|1          ;B
```

Note that in instruction 23, the address LOCAL|0 is used to access the **&rest** argument.

The **catch** special form is also handled specially by the compiler. Here is a simple example of **catch**:

```
(defun a ()
  (catch 'foo (bar)))
```

The disassembled code looks like this:

```
24 MOVE D-PDL FEF|6             ;'30
25 (MISC) %CATCH-OPEN D-RETURN
26 MOVE D-PDL FEF|7             ;'FOO
27 CALL0 D-RETURN FEF|8         ;#'BAR
```

The %CATCH-OPEN instruction is like the CALL instruction; it starts a call to the **catch** function. It takes one "argument" on the stack, which is the location in the program that should be branched to if this **catch** is **thrown** to. In addition to saving that program location, the instruction saves the state of the stack and of special-variable binding so that they can be restored in the event of a **throw**. So instructions 24 and 25 start a **catch** block, and the rest of the function computes the two arguments of the **catch**. Note, however, that **catch** is not actually called. The last form inside the **catch**, in this case **(bar)**, is compiled so as to return its values directly out of the function **a**. The only way that the inactive stack frame for **catch** matters is if a **throw** is done during the execution of **bar**. This searches for a pending call to **catch** and returns from that frame. In this case, since the **%CATCH-OPEN** instruction specifies **D-RETURN**, the values thrown are returned from **a**.

You may have wondered why instruction 24 is there at all. If the destination of a **catch** is not **D-RETURN**, it is necessary for **throw** to resume execution of the function containing the **catch**. Then it is necessary to specify what instruction to resume at. For example:

```
(defun a ()
  (catch 'foo (bar))
  (print t))
```

The disassembled code looks like this:

```
26 MOVE D-PDL FEF|6            ;'32
27 (MISC) %CATCH-OPEN D-IGNORE
30 MOVE D-PDL FEF|7            ;'(BAR)
31 MOVE D-LAST FEF|8           ;'FOO
32 CALL D-RETURN FEF|9         ;#'PRINT
33 MOVE D-LAST 'T
```

The instruction 26 pushes 32, which is the number of instruction at which execution should resume if there is a **throw**.

To allow compilation of **(multiple-value (...) (catch ...))**, there is a special instruction called **%CATCH-OPEN-MULT-VALUE**, which is a cross between **%CATCH-OPEN** and **%CALL-MULT-VALUE**.

32. Querying the User

The following functions provide a convenient and consistent interface for asking questions of the user. Questions are printed and the answers are read on the stream ***query-io***, which normally is synonymous with ***terminal-io*** but can be rebound to another stream for special applications.

The macro **with-timeout** (see page 686) can be used with the functions in this chapter to assume an answer if the user does not respond in a fixed period of time.

We first describe two simple functions for yes-or-no questions, then the more general function on which all querying is built.

y-or-n-p &optional *format-string* &rest *format-args*

This is used for asking the user a question whose answer is either 'y' for 'yes' or 'n' for 'no'. It prints a message by passing *format-string* and *format-args* to **format**, reads a one-character answer, echoes it as 'Yes' or 'No', and returns **t** if the answer is 'yes' or **nil** if the answer is 'no'. The characters which mean 'yes' are 'Y', 'T', **Space**, and **Hand-up**. The characters which mean "no" are 'N', **Rubout**, and **Hand-down**. If any other character is typed, the function beeps and demands a 'Y or N' answer.

You should include a question mark and a space at the end of the message. **y-or-n-p** does type '(Y or N)' for you.

query-io is used for all input and output.

y-or-n-p should be used only for questions that the user knows are coming. If the user is not going to be anticipating the question (e.g. if the question is "Do you really want to delete all of your files?" out of the blue) then **y-or-n-p** should not be used, because the user might type ahead a 'T', 'Y', 'N', **Space**, or **Rubout**, and therefore accidentally answer the question. In such cases, use **yes-or-no-p**.

yes-or-no-p &optional *format-string* &rest *format-args*

This is used for asking the user a question whose answer is either 'yes' or 'no'. It prints a message by passing *format-string* and *format-args* to **format**, beeps, and reads in a line from ***query-io***. If the line is 'yes', it returns **t**. If the line is 'no', it returns **nil**. (Case is ignored, as are leading and trailing spaces and tabs.) If the input line is anything else, **yes-or-no-p** beeps and demands a 'yes or no' answer.

You should include a question mark and a space at the end of the message. **yes-or-no-p** does type '(Yes or No)' for you.

query-io is used for all input and output.

To allow the user to answer a yes-or-no question with a single character, use **y-or-n-p**. **yes-or-no-p** should be used for unanticipated or momentous questions; this is why it beeps and why it requires several keystrokes to answer it.

fquery *options format-string* &rest *format-args*

Asks a question, printed by (format *query-io* *format-string format-args...*), and returns the answer. fquery takes care of checking for valid answers, reprinting the question when the user clears the screen, giving help, and so forth.

options is a list of alternating keywords and values, used to select among a variety of features. Most callers pass a constant list as the *options* (rather than consing up a list whose contents varies). The keywords allowed are:

:type	What type of answer is expected. The currently-defined types are :tyi (a single character), :readline or :mini-buffer-or-readline (a line terminated by a carriage return). :tyi is the default. :mini-buffer-or-readline is nearly the same as :readline, the only difference being that the former uses a minibuffer if used inside the editor.
:choices	Defines the allowed answers. The allowed forms of choices are complicated and explained below. The default is the same set of choices as the y-or-n-p function (see above). Note that the :type and :choices options should be consistent with each other.
:list-choices	If t, the allowed choices are listed (in parentheses) after the question. The default is t: supplying nil causes the choices not to be listed unless the user tries to give an answer which is not one of the allowed choices.
:help-function	Specifies a function to be called if the user hits the Help key. The default help-function simply lists the available choices. Specifying nil disables special treatment of Help. Specifying a function of three arguments—the stream, the list of choices, and the type-function—allows smarter help processing. The type-function is the internal form of the :type option and can usually be ignored.
:condition	If non-nil, a signal name (see page 713) to be signaled before asking the question. A condition handler may handle the condition, specifying an answer for fquery to return, in which case the user is not asked. The details are given below. The default signal name is :fquery, which signals condition name :fquery.
:fresh-line	If t, *query-io* is advanced to a fresh line before asking the question. If nil, the question is printed wherever the cursor was left by previous typeout. The default is t.
:beep	If t, fquery beeps to attract the user's attention to the question. The default is nil, which means not to beep unless the user tries to give an answer which is not one of the allowed choices.
:stream	The value should be either an I/O stream or a symbol or expression that will evaluate to one. fquery uses the specified stream instead of *query-io* for all its input and output.
:clear-input	If t, fquery throws away type-ahead before reading the user's response to the question. Use this for unexpected questions. The default is nil, which means not to throw away type-ahead unless the user tries to give an answer which is not one of the allowed choices. In that case, type-ahead

is discarded since the user probably wasn't expecting the question.

:make-complete
: If t and *query-io* is a typeout-window, the window is "made complete" after the question has been answered. This tells the system that the contents of the window are no longer useful. Refer to the window system documentation for further explanation. The default is t.

The argument to the :choices option is a list each of whose elements is a *choice*. The cdr of a choice is a list of the user inputs which correspond to that choice. These should be characters for :type :tyi or strings for :type :readline. The car of a choice is either a symbol which fquery should return if the user answers with that choice, or a list whose first element is such a symbol and whose second element is the string to be echoed when the user selects the choice. In the former case nothing is echoed. In most cases :type :readline would use the first format, since the user's input has already been echoed, and :type :tyi would use the second format, since the input has not been echoed and furthermore is a single character, which would not be mnemonic to see on the display.

A choice can also be the symbol :any. If used, it must be the last choice. It means that any input is allowed, and should simply be returned as a string or character if it does not match any of the other choices.

Perhaps this can be clarified by example. The yes-or-no-p function uses this list of choices:

```
        ((t "Yes") (nil "No"))
```

and the y-or-n-p function uses this list:

```
        (((t "Yes.") #\y #\t #\space #\hand-up)
         ((nil "No.") #\n #\rubout #\hand-down))
```

If a signal name is specified (or allowed to default to :fquery), before asking the question fquery will signal it. (See section 30.1, page 698 for information about conditions.) make-condition will receive, in addition to the signal name, all the arguments given to fquery, including the list of options, the format string, and all the format arguments. fquery provides one proceed type, :new-value, and if a condition handler proceeds, the argument it proceeds with is returned by **fquery**.

If you want to use the formatted output functions instead of **format** to produce the promting message, write

(fquery *options* (format:outfmt *exp-or-string exp-or-string* ...))

format:outfmt puts the output into a list of a string, which makes **format** print it exactly as is. There is no need to supply additional arguments to the **fquery** unless it signals a condition. In that case the arguments might be passed so that the condition handler can see them. The condition handler will receive a list containing one string, the message, as its third argument instead of just a string. If this argument is passed along to **format**, all the right things happen.

fquery (condition) *Condition*

This condition is signaled, by default, by **fquery**. The condition instance supports these operations:

:options	Returns the list of options given to **fquery**.
:format-string	Returns the format string given to **fquery**.
:format-args	Returns the list of additional args for format, given to **fquery**.

One proceed type is provided, :new-value. It should be used with a single argument, which will be returned by **fquery** in lieue of asking the user.

format:y-or-n-p-options *Constant*

A suitable list to pass as the first argument to **fquery** to make it behave like **y-or-n-p**.

format:yes-or-no-p-options *Constant*

A suitable list to pass as the first argument to **fquery** to make it behave like **yes-or-no-p**.

format:y-or-n-p-choices *Constant*

A list which **y-or-n-p** uses as the value of the :choices option.

33. Initializations

There are a number of programs and facilities in the Lisp Machine that require that "initialization routines" be run either when the facility is first loaded, or when the system is booted, or both. These initialization routines may set up data structures, start processes running, open network connections, and so on.

It is easy to perform an action when a file is loaded: simply place an expression to perform the action in the file. But this causes the action to be repeated if the file is loaded a second time, and often that should not be done. Also, this does not provide a way to cause actions to be taken at other times, such as when the system is booted or when a garbage collection is started.

The *initialization list* facility serves these needs. An initialization list is a symbol whose value is a list of *initializations*, put on by various programs, all to be performed when a certain event (such as a cold boot) happens. When the event occurs, the system function in charge of handling the event (si:lisp-reinitialize, for cold boot) executes all the initializations on the appropriate list, in the order they are present on the list.

Each initialization has a name, a form to be evaluated, a flag saying whether the form has yet been evaluated, and the source file of the initialization, if any. The name is a string or a symbol and lies in the car of an initialization; thus assoc may be used on initialization lists to find particular initializations.

System and user files place initializations on initialization lists using the function add-initialization. The name of the initialization is specified so that the system can distinguish between adding a new initialization and repeating or changing the definition of an initialization already known: if there is already an initialization with the specified name, this is a new definition of the same initialization. One can specify that the initialization be executed immediately if it is new but not if it is repeated.

User programs are free to create their own initialization lists to be run at their own times.

33.1 System Initialization Lists

There are several initialization lists built into the system. Each one is invoked by the system at a specific time, such as immediately after a cold boot, or during disk-save. A user program can put initializations on these lists to cause actions to be taken at those times as the program needs. This avoids the need to modify system functions such as lisp-reinitialize or disk-save in order to make them interact properly with the user program.

The system initialization lists are generally identified by keywords rather than by their actual names. We name them here by their keywords. In each case, the actual initialization list symbol is in the si package, and its name is the conventional keyword followed by '-initialization-list'. Thus, for :cold, there is si:cold-initialization-list. This is just a convention.

Unless otherwise specified, an initialization added to a system list is not run when it is added, only when the appropriate event happens. A few system lists are exceptions and also run each initialization when it is added. Such exceptions are noted explicitly.

The :once initialization list is used for initializations that need to be done only once when the subsystem is loaded and must never be done again. For example, there are some databases that need to be initialized the first time the subsystem is loaded, but should not be reinitialized every time a new version of the software is loaded into a currently running system. This list is for that purpose. When a new initialization is added to this list, it is executed immediately; but when an initialization is redefined, it is not executed again.

The :cold initialization list is used for things that must be run once at cold-boot time. The initializations on this list are run after the ones on :system but before the ones on the :warm list.

The :warm initialization list is used for things which must be run every time the machine is booted, including warm boots. The function that prints the greeting, for example, is on this list. For cold boots, the :cold initializations are done before the :warm ones.

The :system initialization list is like the :warm list but its initializations are run *before* those of the :cold list. These are generally very fundamental system initializations that must be done before the :cold or :warm initializations can work. Initializing the process and window systems, the file system, and the Chaosnet NCP falls in this category. By default, a new initialization added to this list is run immediately also. In general, the system list should not be touched by user subsystems, though there may be cases when it is necessary to do so.

The :before-cold initialization list is used for things to be run by disk-save. Thus they happen essentially at cold boot time, but only once when the world is saved, not each time it is started up.

The :site initialization list is run every time a new site table and host table are loaded by update-site-configuration-info. By default, adding an initialization to this list runs the initialization immediately, even if the initialization is not new.

The :site-option initialization list is run every time the site options may have changed; that is, when a new site tables are loaded or after a cold boot (to see the per-machine options of the machine being booted on). By default, adding an initialization to this list runs the initialization immediately, even if the initialization is not new.

The :full-gc initialization list is run by the function si:full-gc just before garbage collecting. Initializations might be put on this list to discard pointers to bulky objects, or to turn copy lists into cdr-coded form so that they will remain permanently localized.

The :after-flip initialization list is run after every garbage collection flip, at the beginning of scavenging. These initializations can force various objects to be copied into new space near each other simply by referencing them all consecutively.

The :after-full-gc initialization list is run by the function si:full-gc just after a flip is done, but before scavenging.

The :login and :logout lists are run by the **login** and **logout** functions (see page 801) respectively. Note that **disk-save** calls **logout**. Also note that often people don't call **logout**; they just cold-boot the machine.

33.2 Programming Initializations

add-initialization *name form* &optional *list-of-keywords initialization-list-name*

Adds an initialization called *name* with the form *form* to the initialization list specified either by *initialization-list-name* or by keyword. If the initialization list already contains an initialization called *name*, it is redefined to execute *form*.

initialization-list-name, if specified, is a symbol that has as its value the initialization list. If it is void, it is initialized (!) to nil, and is given a si:initialization-list property of t. If a keyword specifies an initialization list, *initialization-list-name* is ignored and should not be specified.

The keywords allowed in *list-of-keywords* are of two kinds. Most specify the initialization list to use; a list of such keywords makes up most of the previous section. Aside from them, four other keywords are allowed, which specify when to evaluate *form*. They are called the *when-keywords*. Here is what they mean:

:normal	Only place the form on the list. Do not evaluate it until the time comes to do this kind of initialization. This is the default unless :system, :once, :site or :site-option is specified.
:first	Evaluate the form now if it is not flagged as having been evaluated before. This is the default if :system or :once is specified.
:now	Evaluate the form now unconditionally as well as adding it to the list.
:redo	Do not evaluate the form now, but set the flag to **nil** even if the initialization is already in the list and flagged **t**.

Actually, the keywords are compared with **string-equal** and may be in any package. If both kinds of keywords are used, the list keyword should come *before* the when-keyword in *list-of-keywords*; otherwise the list keyword may override the when-keyword.

The **add-initialization** function keeps each list ordered so that initializations added first are at the front of the list. Therefore, by controlling the order of execution of the additions, you can control explicit dependencies on order of initialization. Typically, the order of additions is controlled by the loading order of files. The system list is the most critically ordered of the predefined lists.

The **add-initialization** keywords that specify an initialization list are defined by a variable; you can add new keywords to it.

si:initialization-keywords *Variable*

Each element on this list defines the keyword for one initialization list. Each element is a list of two or three elements. The first is the keyword symbol that names the initialization list. The second is a special variable, whose value is the initialization list itself. The third, if present, is a symbol defining the default "time" at which initializations added to this list should be evaluated; it should be si:normal, si:now, si:first, or si:redo. This third element just acts as a default; if the list of keywords passed to add-initialization contains one of the keywords normal, now, first, or redo, it overrides this default. If the third element is not present, it is as if the third element were si:normal.

delete-initialization *name* &optional *keywords initialization-list-name*

Removes the specified initialization from the specified initialization list. Keywords may be any of the list options allowed by add-initialization.

initializations *initialization-list-name* &optional *redo-flag flag-value*

Performs the initializations in the specified list. *redo-flag* controls whether initializations that have already been performed are re-performed; nil means no, non-nil is yes, and the default is nil. *flag-value* is the value to be bashed into the flag slot of an entry. If it is unspecified, it defaults to t, meaning that the system should remember that the initialization has been done. The reason that there is no convenient way for you to specify one of the specially-known-about lists is that you shouldn't be calling initializations on them. This is done by the system when it is appropriate.

reset-initializations *initialization-list-name*

Bashes the flag of all entries in the specified list to nil, thereby causing them to get rerun the next time the function initializations is called on the initialization list.

34. Dates and Times

The time package contains a set of functions for manipulating dates and times: finding the current time, reading and printing dates and times, converting between formats, and other miscellany regarding peculiarities of the calendar system. It also includes functions for accessing the Lisp Machine's microsecond timer.

Times are represented in two different formats by the functions in the time package. One way is to represent a time by many numbers, indicating a year, a month, a date, an hour, a minute, and a second (plus, sometimes, a day of the week and timezone). If a year less than 100 is specified, a multiple of 100 is added to it to bring it within 50 years of the present. Year numbers returned by the time functions are greater than 1900. The month is 1 for January, 2 for February, etc. The date is 1 for the first day of a month. The hour is a number from 0 to 23. The minute and second are numbers from 0 to 59. Days of the week are fixnums, where 0 means Monday, 1 means Tuesday, and so on. A timezone is specified as the number of hours west of GMT; thus in Massachusetts the timezone is 5. Any adjustment for daylight savings time is separate from this.

This "decoded" format is convenient for printing out times into a readable notation, but it is inconvenient for programs to make sense of these numbers and pass them around as arguments (since there are so many of them). So there is a second representation, called Universal Time, which measures a time as the number of seconds since January 1, 1900, at midnight GMT. This "encoded" format is easy to deal with inside programs, although it doesn't make much sense to look at (it looks like a huge integer). So both formats are provided; there are functions to convert between the two formats; and many functions exist in two versions, one for each format.

The Lisp Machine hardware includes a timer that counts once every microsecond. It is controlled by a crystal and so is fairly accurate. The absolute value of this timer doesn't mean anything useful, since it is initialized randomly; what you do with the timer is to read it at the beginning and end of an interval, and subtract the two values to get the length of the interval in microseconds. These relative times allow you to time intervals of up to an hour (32 bits) with microsecond accuracy.

The Lisp Machine keeps track of the time of day by maintaining a *timebase*, using the microsecond clock to count off the seconds. On the CADR, when the machine first comes up, the timebase is initialized by querying hosts on the Chaosnet to find out the current time. The Lambda has a calendar clock which never stops, so it normally does not need to do this. You can also set the time base using time:set-local-time, described below.

There is a similar timer that counts in 60ths of a second rather than microseconds; it is useful for measuring intervals of a few seconds or minutes with less accuracy. Periodic housekeeping functions of the system are scheduled based on this timer.

34.1 Getting and Setting the Time

get-decoded-time
time:get-time

Gets the current time, in decoded form. Return seconds, minutes, hours, date, month, year, day-of-the-week, and daylight-savings-time-p, with the same meanings as decode-universal-time (see page 782). If the current time is not known, nil is returned.

The name time:get-time is obsolete.

get-universal-time

Returns the current time in Universal Time form.

time:set-local-time &optional *new-time*

Sets the local time to *new-time*. If *new-time* is supplied, it must be either a universal time or a suitable argument to time:parse-universal-time (see page 781). If it is not supplied, or if there is an error parsing the argument, you are prompted for the new time. Note that you will not normally need to call this function; it is useful mainly when the timebase gets screwed up for one reason or another.

34.1.1 Elapsed Time in 60ths of a Second

The following functions deal with a different kind of time. These are not calendrical date/times, but simply elapsed time in 60ths of a second. These times are used for many internal purposes where the idea is to measure a small interval accurately, not to depend on the time of day or day of month.

time

Returns a number that increases by 1 every 60th of a second. The value wraps around roughly once a day. Use the time-lessp and time-difference functions to avoid getting in trouble due to the wrap-around. time is completely incompatible with the Maclisp function of the same name.

Note that time with an argument measures the length of time required to evaluate a form. See page 794.

get-internal-run-time
get-internal-real-time

Returns the total time in 60ths of a second since the last boot. This value does not wrap around. Eventually it becomes a bignum. The Lisp Machine does not distinguish between run time and real time.

internal-time-units-per-second *Constant*

According to Common Lisp, this is the ratio between a second and the time unit used by values of get-internal-real-time. On the Lisp Machine, the value is 60. The value may be different in other Common Lisp implementations.

time-lessp *time1 time2*

t if *time1* is earlier than *time2*, compensating for wrap-around, otherwise nil.

time-difference *time1 time2*

Assuming *time1* is later than *time2*, returns the number of 60ths of a second difference between them, compensating for wrap-around.

time-increment *time interval*

Increments *time* by *interval*, wrapping around if appropriate.

34.1.2 Elapsed Time in Microseconds

time:microsecond-time

Returns the value of the microsecond timer, as a bignum. The values returned by this function wrap around back to zero about once per hour.

time:fixnum-microsecond-time

Returns as a fixnum the value of the low 23 bits of the microsecond timer. This is like time:microsecond-time, with the advantage that it returns a value in the same format as the time function, except in microseconds rather than 60ths of a second. This means that you can compare fixnum-microsecond-times with time-lessp and time-difference. time:fixnum-microsecond-time is also a bit faster, but has the disadvantage that since you only see the low bits of the clock, the value can wrap around more quickly (about every eight seconds). Note that the Lisp Machine garbage collector is so designed that the bignums produced by time:microsecond-time are garbage-collected quickly and efficiently, so the overhead for creating the bignums is really not high.

34.2 Printing Dates and Times

The functions in this section create printed representations of times and dates in various formats and send the characters to a stream. To any of these functions, you may pass nil as the *stream* parameter and the function will return a string containing the printed representation of the time, instead of printing the characters to any stream.

The three functions time:print-time, time:print-universal-time, time:print-brief-universal-time and time:print-current-time accept an argument called *date-print-mode*, whose purpose is to control how the date is printed. It always defaults to the value of time:*default-date-print-mode*. Possible values include:

:dd//mm//yy	Print the date as in '3/16/53'.
:mm//dd//yy	Print as in '16/3/53'.
:dd-mm-yy	Print as in '16-3-53'.
:dd-mmm-yy	Print as in '16-Mar-53'.
:|dd mmm yy|	Print as in '16 Mar 53'.
:ddmmmyy	Print as in '16Mar53'.

:yymmdd	Print as in '530316'.
:yymmmdd	Print as in '53Mar16'.

time:print-current-time &optional (*stream* *standard-output*)
Prints the current time, formatted as in 11/25/80 14:50:02, to the specified stream. The date portion may be printed differently according to the argument *date-print-mode*.

time:print-time *seconds minutes hours date month year* &optional (*stream* *standard-output*) *date-print-mode*
Prints the specified time, formatted as in 11/25/80 14:50:02, to the specified stream. The date portion may be printed differently according to the argument *date-print-mode*.

time:print-universal-time *universal-time* &optional (*stream* *standard-output*) (*timezone* time:*timezone*) *date-print-mode*
Prints the specified time, formatted as in 11/25/80 14:50:02, to the specified stream. The date portion may be printed differently according to the argument *date-print-mode*.

time:print-brief-universal-time *universal-time* &optional (*stream* *standard-output*) *reference-time date-print-mode*
This is like time:print-universal-time except that it omits seconds and only prints those parts of *universal-time* that differ from *reference-time*, a universal time that defaults to the current time. Thus the output is in one of the following three forms:

```
02:59              ;the same day
3/4 14:01          ;a different day in the same year
8/17/74 15:30      ;a different year
```

The date portion may be printed differently according to the argument *date-print-mode*.

time:*default-date-print-mode* *Variable*
Holds the default for the *date-print-mode* argument to each of the functions above. Initially the value here is :mm//dd/yy.

time:print-current-date &optional (*stream* *standard-output*)
Prints the current time, formatted as in Tuesday the twenty-fifth of November, 1980; 3:50:41 pm, to the specified stream.

time:print-date *seconds minutes hours date month year day-of-the-week* &optional (*stream* *standard-output*)
Prints the specified time, formatted as in Tuesday the twenty-fifth of November, 1980; 3:50:41 pm, to the specified stream.

time:print-universal-date *universal-time* &optional (*stream* *standard-output*) (*timezone* time:*timezone*)
Prints the specified time, formatted as in Tuesday the twenty-fifth of November, 1980; 3:50:41 pm, to the specified stream.

34.3 Reading Dates and Times

These functions accept most reasonable printed representations of date and time and convert them to the standard internal forms. The following are representative formats that are accepted by the parser. Note that slashes are escaped with additional slashes, as is necessary if these strings are input in traditional syntax.

```
"March 15, 1960"        "3//15//60"    "3//15//1960"
"15 March 1960"         "15//3//60"    "15//3//1960"
"March-15-60"           "3-15-60"      "3-15-1960"
"15-March-60"           "15-3-60"      "15-3-1960"
"15-Mar-60"             "3-15"         "15 March 60"
"Fifteen March 60"      "The Fifteenth of March, 1960;"
"Friday, March 15, 1980"

"1130."      "11:30"      "11:30:17"   "11:30 pm"
"11:30 AM"   "1130"       "113000"
"11.30"      "11.30.00"   "11.3"       "11 pm"

"12 noon"    "midnight"   "m"     "6:00 gmt"   "3:00 pdt"
```

any date format may be used with any time format

```
"One minute after March 3, 1960"
```
meaning one minute after midnight
```
"Two days after March 3, 1960"
"Three minutes after 23:59:59 Dec 31, 1959"

"Now"      "Today"      "Yesterday"    "five days ago"
"two days after tomorrow"    "the day after tomorrow"
"one day before yesterday"   "BOB@OZ's birthday"
```

time:parse *string* &optional (*start* 0) (*end* nil) (*futurep* t) *base-time must-have-time date-must-have-year time-must-have-second* (*day-must-be-valid* t)

Interpret *string* as a date and/or time, and return seconds, minutes, hours, date, month, year, day-of-the-week, daylight-savings-time-p, and relative-p. *start* and *end* delimit a substring of the string; if *end* is nil, the end of the string is used. *must-have-time* means that *string* must not be empty. *date-must-have-year* means that a year must be explicitly specified. *time-must-have-second* means that the second must be specified. *day-must-be-valid* means that if a day of the week is given, then it must actually be the day that corresponds to the date. *base-time* provides the defaults for unspecified components; if it is nil, the current time is used. *futurep* means that the time should be interpreted as being in the future; for example, if the base time is 5:00 and the string refers to the time 3:00, that means the next day if *futurep* is non-nil, but it means two hours ago if *futurep* is nil. The *relative-p* returned value is t if the string included a relative part, such as 'one minute after' or 'two days before' or 'tomorrow' or 'now'; otherwise, it is nil.

If the input is not valid, the error condition sys:parse-error is signaled (see page 505).

time:parse-universal-time *string* &optional (*start* 0) (*end* nil) (*futurep* t) *base-time must-have-time date-must-have-year time-must-have-second* (*day-must-be-valid* t)

This is the same as time:parse except that it returns two values: an integer, representing the time in Universal Time, and the *relative-p* value.

34.4 Reading and Printing Time Intervals

In addition to the functions for reading and printing instants of time, there are other functions specifically for printing time intervals. A time interval is either a number (measured in seconds) or nil, meaning 'never'. The printed representations used look like '3 minutes 23 seconds' for actual intervals, or 'Never' for nil (some other synonyms and abbreviations for 'never' are accepted as input).

time:print-interval-or-never *interval* &optional (*stream* *standard-output*)

interval should be a non-negative fixnum or nil. Its printed representation as a time interval is written onto *stream*.

time:parse-interval-or-never *string* &optional *start end*

Converts *string*, a printed representation for a time interval, into a number or *nil*. *start* and *end* may be used to specify a portion of *string* to be used; the default is to use all of *string*. It is an error if the contents of string do not look like a reasonable time interval. Here are some examples of acceptable strings:

```
"4 seconds"        "4 secs"          "4 s"
"5 mins 23 secs"   "5 m 23 s"        "23 SECONDS 5 M"
          "3 yrs 1 week 1 hr 2 mins 1 sec"
"never"            "not ever"        "no"              ""
```

Note that several abbreviations are understood, the components may be in any order, and case (upper versus lower) is ignored. Also, "months" are not recognized, since various months have different lengths and there is no way to know which month is being spoken of. This function always accepts anything that was produced by time:print-interval-or-never; furthermore, it returns exactly the same fixnum (or nil) that was printed.

time:read-interval-or-never &optional (*stream* *standard-input*)

Reads a line of input from *stream* (using readline) and then calls time:parse-interval-or-never on the resulting string.

34.5 Time Conversions

decode-universal-time *universal-time* &optional (*timezone* time:*timezone*)
Converts *universal-time* into its decoded representation. The following values are returned: seconds, minutes, hours, date, month, year, day-of-the-week, daylight-savings-time-p, and the timezone used. *daylight-savings-time-p* tells you whether or not daylight savings time is in effect; if so, the value of *hour* has been adjusted accordingly. You can specify *timezone* explicitly if you want to know the equivalent representation for this time in other parts of the world.

encode-universal-time *seconds minutes hours date month year* &optional *timezone*
Converts the decoded time into Universal Time format, and return the Universal Time as an integer. If you don't specify *timezone*, it defaults to the current timezone adjusted for daylight savings time; if you provide it explicitly, it is not adjusted for daylight savings time. If *year* is less than 100, it is shifted by centuries until it is within 50 years of the present.

time:*timezone* *Variable*
The value of **time:*timezone*** is the time zone in which this Lisp Machine resides, expressed in terms of the number of hours west of GMT this time zone is. This value does not change to reflect daylight savings time; it tells you about standard time in your part of the world.

34.6 Internal Functions

These functions provide support for those listed above. Some user programs may need to call them directly, so they are documented here.

time:initialize-timebase
Initializes the timebase by querying Chaosnet hosts to find out the current time. This is called automatically during system initialization. You may want to call it yourself to correct the time if it appears to be inaccurate or downright wrong. See also **time:set-local-time**, page 777.

time:daylight-savings-time-p *hours date month year*
Returns **t** if daylight savings time is in effect for the specified hour; otherwise, return **nil**. If *year* is less than 100, it is shifted by centuries until it is within 50 years of the present.

time:daylight-savings-p
Returns **t** if daylight savings time is currently in effect; otherwise, returns **nil**.

time:month-length *month year*
Returns the number of days in the specified *month*; you must supply a *year* in case the month is February (which has a different length during leap years). If *year* is less than 100, it is shifted by centuries until it is within 50 years of the present.

time:leap-year-p *year*

Returns t if *year* is a leap year; otherwise return nil. If *year* is less than 100, it is shifted by centuries until it is within 50 years of the present.

time:verify-date *date month year day-of-the-week*

If the day of the week of the date specified by *date*, *month*, and *year* is the same as *day-of-the-week*, returns nil; otherwise, returns a string that contains a suitable error message. If *year* is less than 100, it is shifted by centuries until it is within 50 years of the present.

time:day-of-the-week-string *day-of-the-week* &optional (*mode* :long)

Returns a string representing the day of the week. As usual, 0 means Monday, 1 means Tuesday, and so on. Possible values of *mode* are:

:long	Returns the full English name, such as "Monday", "Tuesday", etc. This is the default.
:short	Returns a three-letter abbreviation, such as "Mon", "Tue", etc.
:medium	Returns a longer abbreviation, such as "Tues" and "Thurs".
:french	Returns the French name, such as "Lundi", "Mardi", etc.
:german	Returns the German name, such as "Montag", "Dienstag", etc.
:italian	Returns the Italian name, such as "Lunedi", "Martedi", etc.

time:month-string *month* &optional (*mode* :long)

Returns a string representing the month of the year. As usual, 1 means January, 2 means February, etc. Possible values of *mode* are:

:long	Returns the full English name, such as "January", "February", etc. This is the default.
:short	Returns a three-letter abbreviation, such as "Jan", "Feb", etc.
:medium	Returns a longer abbreviation, such as "Sept", "Novem", and "Decem".
:roman	Returns the Roman numeral for *month* (this convention is used in Europe).
:french	Returns the French name, such as "Janvier", "Fevrier", etc.
:german	Returns the German name, such as "Januar", "Februar", etc.
:italian	Returns the Italian name, such as "Gennaio", "Febbraio", etc.

time:timezone-string &optional (*timezone* time:*timezone*)
(*daylight-savings-p* (time:daylight-savings-p))

Return the three-letter abbreviation for this time zone. For example, if *timezone* is 5, then either "EST" (Eastern Standard Time) or "CDT" (Central Daylight Time) is used, depending on *daylight-savings-p*.

35. Miscellaneous Useful Functions

This chapter describes a number of functions that don't logically fit in anywhere else. Most of these functions are not normally used in programs, but are "commands", i.e. things that you type directly at Lisp.

35.1 Documentation

documentation *name doc-type*

Returns the documentation string of *name* in the role *doc-type*. *doc-type* should be a symbol, but only its print-name matters. **function** as *doc-type* requests the documentation of *name* as a function, **variable** as *doc-type* requests the documentation of *name* as a variable, and so on.

When *doc-type* is **function**, *name* can be any function spec, and the documentation string of its function definition is returned. Otherwise, *name* must be a symbol, and *doc-type* may be anything. However, only these values of *doc-type* are standardly used:

variable	Documentation of *name* as a special variable. Such documentation is recorded automatically by **defvar**, **defconst**, **defconstant**, **defparameter** (page 33).
type	Documentation of *name* as a type for **typep**. Recorded automatically when a documentation string is given in a **deftype** form (page 19).
structure	Documentation of *name* as a **defstruct** type. Recorded automatically by a **defstruct** for *name* (chapter 20, page 372).
setf	Documentation on what it means to **setf** a form that starts with *name*. Recorded when there is a documentation string in a **defsetf** of *name* (page 345).
flavor	Documentation of the flavor named *name*. Put on by the **:documentation** option in a **defflavor** for *name* (page 414).
resource	Documentation of the resource named *name*. Put on when there is a documentation string in a **defresource** of *name* (page 124).
signal	Documentation for *name* as a signal name. Put on when there is a documentation string in a **defsignal** or **defsignal-explicit** for *name* (page 714).

Documentation strings for any *doc-type* can be added to *name* by doing **(setf (documentation** *name doc-type***)** *string***)**.

The command **Control-Shift-D** in Zmacs and the rubout handler, used within a call to a function, prints the documentation of that function. **Control-Shift-V**, within a symbol, prints the documentation of that symbol as a variable.

35.2 Hardcopy

The hardcopy functions allow you to specify the printer to use on each call. The default is set up by the site files for your site, but can be overridden for a particular machine in the LMLOCS file or by a user in his INIT file. Any kind of printer can be used, no matter how it is actually driven, if it is hooked into the software properly as described below.

A *printer-type* is a keyword that has appropriate properties; a *printer* is either a printer-type or a list starting with one. The rest of the list can specify *which* printer of that type you want to use (perhaps with a host name or filename).

The printer types defined by the system are:

:dover	This printer type is used by itself as a printer, and refers to the Dover at MIT.
:xgp	This printer type indicates a printer that is accessed by writing spool files in MIT XGP format. A printer would be specified as a list, (:xgp *filename*), specifying where to write the spool file.
:press-file	This printer type is used in a list together with a file name, as in (:press-file "OZ:<RMS>FOO.PRESS"). Something is "printed" on such a printer by being converted to a press file and written under that name.

hardcopy-file *filename* &rest *options*

Print the file *filename* in hard copy on the specified printer or the default printer. *options* is a list of keyword argument names and values. There are only two keywords that are always meaningful: :format and :printer. Everything else is up to the individual printer to interpret. The list here is only a paradigm or suggestion.

:printer	The value is the printer to use. The default is the value of si:*default-printer*.
:format	The value is a keyword that specifies the format of file to be parsed. The standard possibilities are :text (an ordinary file of text), :xgp (a file of the sort once used by the XGP at MIT), :press (a Xerox-style press file) and :suds-plot (a file produced by the Stanford drawing program). However, each kind of printer may define its own format keywords.
:font	
:font-list	The value of *font* is the name of a font to print the file in (a string). Alternatively, you can give :font-list and specify a list of such font names, for use if the file contains font-change commands. The interpretation of a font name is dependent on the printer being used. There is no necessary relation to Lisp machine display fonts. However, printers are encouraged to use, by default, fonts that are similar in appearance to the Lisp machine fonts listed in the file's attribute list, if it is a text file.
:heading-font	The value is the name of the font for use in page headers, if there are any.
:vsp	The value is the extra spacing to use between lines, in over and beyond the height of the fonts.

:page-headings
If the value is non-nil, a heading is added to each page.

:copies The value is the number of copies to print.

:spool If the printer provides optional spooling, this argument says whether to spool (default is nil). Some printers may intrinsically always spool; others may have no way to spool.

set-printer-default-option *printer-type option value*
Sets a default for *option* for printers of type *printer-type*. Any use of the hardcopy functions with a printer of that type and no value specified for *option* will use the value *value*. For example,

```
    (set-printer-default-option :dover :spool t)
```

causes output to Dover printers to be spooled unless the :spool option is explicitly specified with value nil.

Currently defaultable options are :font, :font-list, :heading-font, :page-headings, :vsp, :copies, and :spool.

hardcopy-stream *stream* &rest *options*
Like hardcopy-file but uses the text read from *stream* rather than opening a file. The :format option is not allowed (since implementing it requires the ability to open the file with unusual open options).

hardcopy-bit-array *array left top right bottom* &rest *options*
Print all or part of the bit-array *array* on the specified or default printer. *options* is a list of keyword argument names and values; the only standard option is :printer, which specifies the printer to use. The default printer is si:*default-bit-array-printer*, or, if that is nil, si:*default-printer.

left, *top*, *right* and *bottom* specify the subrectangle of the array to be printed. All four numbers measure from the top left corner (which is element 0, 0).

hardcopy-status &optional *printer* (*stream* *standard-output*)
Prints the status of *printer*, or the default printer. This should include if possible such things as whether the printer has paper and what is in the queue.

si:*default-printer* *Variable*
This is the default printer. It is set from the :default-printer site option.

si:*default-bit-array-printer* *Variable*
If non-nil, this is the default printer for printing bit arrays, overriding si:*default-printer*. A separate default is provided for bit arrays since some printers that can print files cannot print bit arrays. This variable is set initially from the :default-bit-array-printer site option.

Defining a printer type:

A printer type is any keyword that has suitable functions on the appropriate properties.

To be used with the function **hardcopy-file**, the printer type must have a **si:print-file** property. To be used with **hardcopy-stream**, the printer type must have a **si:print-stream** property. **hardcopy-bit-array** uses the **si:print-bit-array** property. **hardcopy-status** uses the **si:print-status** property. (The hardcopy functions' names are not themselves used simply to avoid using a symbol in the global package as a property name of a symbol that might be in the global package as well).

Each property, to be used, should be a function whose first argument will be the printer and whose remaining arguments will fit the same pattern as those of the hardcopy function the user called. (They will not necessarily be the same arguments, as some additional keywords may be added to the list of keyword arguments; but they will fit the same description.)

For example,

```
    (hardcopy-file "foo" :printer '(:press-file "bar.press"))
```

results in the execution of

```
    (funcall (get :press-file 'si:print-file)
             '(:press-file "bar.press")
             "foo" :printer '(:press-file "bar.press"))
```

A printer type need not support operations that make no sense on it. For example, there is no **si:print-status** property on **:press-file**.

35.3 Metering

The metering system is a way of finding out what parts of your program use up the most time. When you run your program with metering, every function call and return is recorded, together with the time at which it took place. Page faults are also recorded. Afterward, the metering system analyzes the records and tells you how much time was spent executing withain each function. Because the records are stored in the disk partition called **METR**, there is room for a lot of data.

Before you meter a program, you must enable metering in some or all stack groups. **meter:enable** is used for this. Then you evaluate one or more forms with metering, perhaps by using **meter:test** or **meter:run**. Finally, you use **meter:analyze** to summarize and print the metering data.

There are two parameters that control whether metering data are recorded. First of all, the variable **sys:%meter-microcode-enables** contains bits that enable recording of various kinds of events. Secondly, each stack group has a flag that controls whether events are recorded while running in that stack group.

sys:%meter-microcode-enables *Variable*

Enables recording of metering data. Each bit controls recording of one kind of event.

1	This bit enables recording of page faults.
2	This bit enables recording of consing.
4	This bit enables recording of function entry and exit.
8	This bit enables recording of stack group switching.

The value is normally zero, which turns off all recording.

These are the functions used to control which stack groups do metering:

meter:enable &rest *things*

Enables metering in the stack groups specified by *things*. Each thing in *things* may be a stack group, a process (which specifies the process's stack group), or a window (which specifies the window's process's stack group). t is also allowed. It enables metering in all stack groups.

meter:disable &rest *things*

Disables metering in the stack groups specified by *things*. The arguments allowed are the same as for meter:enable. (meter:disable t) turns off (meter:enable t), but does not disable stack groups enabled individually. (meter:disable) disables all stack groups no matter how you specified to enable them.

meter:metered-objects *Variable*

This is a list of all the *things* you have enabled with meter:enable and not disabled.

These are the functions to evaluate forms with metering:

meter:run *forms*

Clears out the metering data and evaluates the *forms* with sys:%meter-microcode-enables bound to 14 octal (record function entry and exit, and stack group switching). Any of the evaluation that takes place in enabled stack groups will record metering data.

meter:test *form* (*enables* #o14)

Clears out the metering data, enables metering for the current stack group only, and evaluates *form* with sys:%meter-microcode-enables bound to *enables*.

This is how you print the results:

meter:analyze &key *analyzer stream file buffer return info* &allow-other-keys

Analyzes the data recorded by metering. *analyzer* is a keyword specifies a kind of analysis. :tree is the default. Another useful alternative is :list-events. Particular analyzers handle other keyword arguments in addition to those listed above.

The output is printed on *stream*, written to a file named *file*, or put in an editor buffer named *buffer* (at most one of these three arguments should be specified). The default is to print on *standard-output*.

Analyzing the metering data involves creating a large intermediate data base. Normally this is created afresh each time **meter:analyze** is called. If you specify a non-nil value for *return*, the intermediate data structure is returned by **meter:analyze**, and can be passed in on another call as the *info* argument. This can save time. But you can only do this if you use the same *analyzer* each time, as different analyzers use different termporary data structures.

The default analyzer **:tree** prints out the amount of run time and real time spent executing each function that was called. The real time includes time spend waiting and time spent writing metering data to disk; for computational tasks, the latter makes the real time less useful than the run time. **:tree** handles these additional keyword arguments to **meter:analyze:**

:find-callers	The argument for this keyword is a function spec or a list of function specs. A list of who called the specified functions, and how often, is printed instead of the usual output.
:stack-group	The argument is a stack group or a list of them; only the activities in those stack groups are printed.
:sort-function	The argument is the name of a suitable sorting function that is used to sort the items for the various functions that were called. Sorting functions provided include **meter:max-page-faults**, **meter:max-calls**, **meter:max-run-time** (the default), **meter:max-real-time**, and **meter:max-run-time-per-call**.
:summarize	The argument is a function spec or a list of function specs; only those functions' statistics are printed.
:inclusive	If this is non-nil, the times for each function include the time spent in executing subroutines called from the function. Note: if a function is called recursively, the time spent in the inner call(s) is counted twice (or more).

The analyzer **:list-events** prints out one line about each event recorded. The line contains the run time and real time (in microseconds), the running count of page faults, the stack group name, the function that was running, the stack depth, the type of event, and a piece of data. For example:

```
      0    0     0 ZMACS-WINDOWS  METER:TEST   202 CALL SI:EVAL
    115   43     0 ZMACS-WINDOWS  METER:TEST   202 RET  SI:EVAL
    180   87     0 ZMACS-WINDOWS  METER:TEST   202 RET  CATCH

   real run    pf  stack-group    function  stack event data
   time time                                level type
```

:list-events is often useful with recording of page faults (**sys:%meter-microcode-enables** set to 1).

meter:reset

Clears out all metering data.

Because metering records pointers to Lisp objects in a disk partition which is not part of the Lisp address space, garbage collection is inhibited (by arresting the gc process) when you turn on metering.

meter:resume-gc-process

Allows garbage collection to continue (if it is already turned on) by unarresting it.

35.4 Poking Around in the Lisp World

who-calls *x* &optional *package* (*inheritors* t) (*inherited* t)

x must be a symbol or a list of symbols. who-calls tries to find all of the functions in the Lisp world that call *x* as a function, use *x* as a variable, or use *x* as a constant. (Constants which are lists containing *x* are not found.) It tries to find all of the functions by searching all of the function cells of all of the symbols in *package* and packages that inherit from *package* (unless *inheritors* is nil) and packages *package* inherits from (unless *inherited* is nil). *package* defaults to the global package, which means that all normal packages are checked.

If who-calls encounters an interpreted function definition, it simply tells you if *x* appears anywhere in the interpreted code. who-calls is smarter about compiled code, since it has been nicely predigested by the compiler. Macros expanded in the compilation of the code can be found because they are recorded in the caller's debugging info alist, even though they are not actually referred to by the compiled code.

If *x* is a list of symbols, who-calls does them all simultaneously, which is faster than doing them one at a time.

who-uses is an obsolete name for who-calls.

The editor has a command, Meta-X List Callers, which is similar to who-calls.

The symbol :unbound-function is treated specially by who-calls. (who-calls :unbound-function) searches all the compiled code for any calls through a symbol that is not currently defined as a function. This is useful for finding errors such as functions you misspelled the names of or forgot to write.

who-calls prints one line of information for each caller it finds. It also returns a list of the names of all the callers.

what-files-call *x* &optional *package* (*inheritors* t) (*inherited* t)

Similar to who-calls but returns a list of the pathnames of all the files that contain functions that who-calls would have printed out. This is useful if you need to recompile and/or edit all of those files.

apropos *substring* &optional *package* &key (*inheritors* t) *inherited dont-print predicate boundp fboundp*

(apropos *substring*) tries to find all symbols whose print-names contain *substring* as a substring. Whenever it finds a symbol, it prints out the symbol's name; if the symbol is defined as a function and/or bound to a value, it tells you so and prints the names of the arguments (if any) to the function.

If *predicate* is non-nil, it should be a function; only symbols on which the function returns non-nil are counted. In addition, *fboundp* non-nil means only symbols with function definitions are considered, and *boundp* non-nil means that only symbols with values are considered.

apropos looks for symbols on *package*, and all packages that use *package* (unless *inheritors* is nil). If *inherited* is non-nil, all packages used by *package* are searched as well. *package* can be a package or a symbol or string naming a package. It can also be a list of packages, symbols and strings; all of the packages thus specified are searched. *package* defaults to a list of all packages except invisible ones.

apropos returns a list of all the symbols it finds. If *dont-print* is non-nil, that is all it does.

sub-apropos *substring starting-list* &key *predicate dont-print*

Finds all symbols in *starting-list* whose names contain *substring*, and that satisfy *predicate*. If *predicate* is nil, the substring is the only condition. The symbols are printed if *dont-print* is nil. A list of the symbols found is returned, in any case.

This function is most useful when applied to the value of *, after apropos has returned a long list.

where-is *pname* &optional *package*

Prints the names of all packages that contain a symbol with the print-name *pname*. If *pname* is a string it gets upper-cased. The package *package* and all packages that inherit from it are searched. *package* can be a package or the name of a package, or a list of packages and names. It defaults to a list of all packages except invisible ones. where-is returns a list of all the symbols it finds.

describe *x*

describe tries to tell you all of the interesting information about any object *x* (except for array contents). describe knows about arrays, symbols, floats, packages, stack groups, closures, and FEFs, and prints out the attributes of each in human-readable form. Sometimes objects found inside *x* are described also; such recursive descriptions are indented appropriately. For instance, describe of a symbol also describes the symbol's value, its definition, and each of its properties. describe of a float (full-size or short) shows you its internal representation in a way that is useful for tracking down roundoff errors and the like.

If *x* is a named-structure, describe invokes the :describe operation to print the description, if that is supported. To understand this, you should read the section on named structures (see page 390). If the :describe operation is not supported, describe looks on the named-structure symbol for information that might have been left by **defstruct**; this information would tell it what the symbolic names for the entries in the structure are, and **describe** knows how to use the names to print out what each field's name and contents is.

describe of an instance always invokes the **:describe** operation. All flavors support it, since **si:vanilla-flavor** defines a method for it.

describe always returns its argument, in case you want to do something else to it.

inspect *x*

A window-oriented version of **describe**. See the window system documentation for details, or try it and type **Help**.

disassemble *function*

Prints out a human-readable version of the macro-instructions in *function*. *function* should be a FEF, or a function spec whose definition is a FEF. The macro-code instruction set is explained in chapter 31, page 752.

The **grindef** function (see page 528) may be used to display the definition of a non-compiled function.

room &rest *areas*

Prints a summary of memory usage.

The first line of output tells you the amount of physical memory on the machine, the amount of available virtual memory not yet filled with data (that is, the portion of the available virtual memory that has not yet been allocated to any region of any area), and the amount of wired physical memory (i.e. memory not available for paging).

Following lines tell you how much room is left in some areas. For each area it tells you about, it prints out the name of the area, the number of regions that currently make up the area, the current size of the area in kilowords, and the amount of the area that has been allocated, also in kilowords. If the area cannot grow, the percentage that is free is displayed.

(room) tells you about those areas that are in the list that is the value of the variable **room**. These are the most interesting ones.

(room *area1 area2*...**)** tells you about those areas, which can be either the names or the numbers.

(room t) tells you about all the areas.

(room nil) does not tell you about any areas; it only prints the first line of output.

room *Variable*

The value of **room** is a list of area names and/or area numbers, denoting the areas that the function **room** should describe if given no arguments. Its initial value is:

```
        (working-storage-area macro-compiled-program)
```

35.5 Utility Programs

ed &optional *x*

ed is the main function for getting into the editor, Zmacs. The commands of Zmacs are very similar to those of Emacs.

(ed) or (ed nil) simply enters the editor, leaving you in the same buffer as the last time you were in the editor. It has the same effect as typing System E.

(ed t) puts you in a fresh buffer with a generated name (like BUFFER-4).

(ed *pathname*) edits that file. *pathname* may be an actual pathname or a string.

(ed 'foo) tries hard to edit the definition of the **foo** function. It can find a buffer or file containing the source code for **foo** and position the cursor at the beginning of the code. In general, **foo** can be any function-spec (see section 11.2, page 223).

(ed 'zwei:reload) reinitializes the editor. It forgets about all existing buffers, so use this only as a last resort.

zwei:save-all-files

This function is useful in emergencies in which you have modified material in Zmacs buffers that needs to be saved, but the editor is partially broken. This function does what the editor's Save All Files command does, but it stays away from redisplay and other advanced facilities so that it might work if other things are broken.

dired &optional *pathname*

Puts up a window and edits the directory named by *pathname*, which defaults to the last file opened. While editing a directory you may view, edit, compare, hardcopy, and delete the files and subdirectories it contains. While in the directory editor type the **Help** key for further information.

mail &optional *user text call-editor-anyway*

Sends the string *text* as mail to *user*. *user* should also be a string, of the form "*username*@*hostname*". Multiple recipients separated by commas are also allowed.

If you do not provide two arguments, **mail** puts up an editor window in which you may compose the mail. Type the **End** key to send the mail and return from the **mail** function.

The window is also used if *call-editor-anyway* is non-**nil**.

bug &optional *topic text call-editor-anyway*

Reports a bug. *topic* is the name of the faulty program (a symbol or a string). It defaults to lispm (the Lisp Machine system itself). *text* is a string which contains the information to report. If you do not provide two arguments, or if *call-editor-anyway* is non-nil, a window is put up for you to compose the mail.

bug is like mail but includes information about the system version and what machine you are on in the text of the message. This information is important to the maintainers of the faulty program; it aids them in reproducing the bug and in determining whether it is one that is already being worked on or has already been fixed.

print-notifications &optional (*from* 0) *to*

Reprints any notifications that have been received. from and to are used to restrict which notifications are printed: both count from the most recent notification as number 0. Thus, (print-notifications 2 8) prints six notifications after skipping the two most recent.

The difference between notifications and sends is that sends come from other users, while notifications are usually asynchronous messages from the Lisp Machine system itself. However, the default way for the system to inform you about a send is to make a notification! So print-notifications *normally* includes all sends as well.

Typing Terminal 1 N pops up a window and calls print-notifications to print on it.

si:print-disk-error-log

Prints information about the half dozen most recent disk errors (since the last cold boot).

peek &optional *character*

Selects the PEEK utility, which displays various information about the system, periodically updating it. PEEK has several modes, which are entered by typing a single key which is the name of the mode or by clicking on the menu at the top. The initial mode is selected by the argument, *character*. If no argument is given, PEEK starts out by explaining what its modes are.

time *form*

Evaluates *form* and prints the length of time that the evaluation took. The values of *form* are returned.

Note that time with no argument is a function to return a time value counting in 60ths of a second; see page 777. This unfortunate collision is a consequence of Common Lisp.

35.6 The Lisp Listen Loop

These functions constitute the Lisp top level read-eval-print loop or *listen loop* and its associated functions.

si:lisp-top-level
This is the first function called in the initial Lisp environment. It calls **lisp-reinitialize**, clears the screen, and calls **si:lisp-top-level1**.

lisp-reinitialize
This function does a wide variety of things, such as resetting the values of various global constants and initializing the error system.

si:lisp-top-level1 **terminal-io**
This is the actual listen loop. Within it, ***terminal-io*** is bound to the argument supplied. This is the stream used for reading and printing if ***standard-input*** and ***standard-output*** are synonyms for ***terminal-io***, as they normally are.

The listen loop reads a form from ***standard-input***, evaluates it, prints the result (with escaping) to ***standard-output***, and repeats indefinitely. If several values are returned by the form all of them are printed. Also the values of *****, **+**, **-**, **//**, **++**, ******, **+++**, ******* and ***values*** are maintained (see below).

break *format-string* &rest *format-args*
Enters a breakpoint loop, which is similar to a Lisp top level loop. *format-string* and the *format-args* are passed to **format** to print a message.

```
;Breakpoint message; Resume to continue, Abort to quit.
```

and then enters a loop reading, evaluating, and printing forms. A difference between a break loop and the top level loop is that when reading a form, **break** checks for the following special cases: If the **Abort** key is typed, control is returned to the previous break or error-handler, or to top-level if there is none. If the **Resume** key is typed, **break** returns **nil**. If the list (**return** *form*) is typed, **break** evaluates *form* and returns the result, without ever calling the function **return**.

Inside the **break** loop, the streams ***standard-output***, ***standard-input***, and **query-io** are bound to be synonymous to ***terminal-io***; ***terminal-io*** itself is not rebound. Several other internal system variables are bound, and you can add your own symbols to be bound by pushing elements onto the value of the variable **sys:*break-bindings*** (see page 797).

break used to be a special form whose first argument was a string or symbol which was simply printed *without evaluating it*. In order to facilitate conversion, **break** really still is a special form. If the call appears to use the old conventions, it behaves in the old way, but the compiler issues a warning if it sees such code.

prin1 *Variable*

The value of this variable is normally **nil**. If it is non-nil, then the read-eval-print loop uses its value instead of the definition of **prin1** to print the values returned by functions. This hook lets you control how things are printed by all read-eval-print loops—the Lisp top level, the **break** function, and any utility programs that include a read-eval-print loop. It does not affect output from programs that call the **prin1** function or any of its relatives such as **print** and **format**; if you want to do that, read about customizing the printer, on section 23.1, page 513. If you set **prin1** to a new function, remember that the read-eval-print loop expects the function to print the value but not to output a **return** character or any other delimiters.

- *Variable*

While a form is being evaluated by a read-eval-print loop, - is bound to the form itself.

+ *Variable*

While a form is being evaluated by a read-eval-print loop, + is bound to the previous form that was read by the loop.

***** *Variable*
// *Variable*

While a form is being evaluated by a read-eval-print loop, * is set to the result printed the last time through the loop. If there were several values printed (because of a multiple-value return), * is bound to the first value. // is bound to a list of all the values of the previous form.

If evaluation of a form is aborted, * and // remain set to the results of the last successfully completed form. If evaluation is successful but printing is aborted, * and // are already set for the following form.

Note that when using Common Lisp syntax you would type just /.

++ *Variable*

++ holds the previous value of +, that is, the form evaluated two interactions ago.

+++ *Variable*

+++ holds the previous value of ++.

****** *Variable*
//// *Variable*

Hold the previous values of * and //, that is, the results of the form evaluated two interactions ago. Only forms whose evaluation is successful cause the values of * and // to move into ** and ////.

Note that when using Common Lisp syntax you would type just //.

******* *Variable*
////// *Variable*

Hold the previous values of ** and ////, that is, the results of the form evaluated three interactions ago. Note that when using Common Lisp syntax you would type just ///.

values *Variable*

values holds a list of all lists of values produced by evaluation in this Lisp listener. (car *values*) is nearly equivalent to //, (cadr *values*) to ////, and so on. The difference is that an element is pushed on *values* for each form whose evaluation is started. If evaluation is aborted, the element of *values* is nil.

sys:*break-bindings* *Variable*

When break is called, it binds some special variables under control of the list which is the value of sys:*break-bindings*. Each element of the list is a list of two elements: a variable and a form that is evaluated to produce the value to bind it to. The bindings happen sequentially. Users may push things on this list (adding to the front of it), but should not replace the list wholesale since several of the variable bindings on this list are essential to the operation of break.

lisp-crash-list *Variable*

The value of lisp-crash-list is a list of forms. lisp-reinitialize sequentially evaluates these forms, and then sets lisp-crash-list to nil.

In most cases, the *initialization* facility should be used rather than lisp-crash-list. Refer to chapter 33, page 772.

35.7 The Garbage Collector

gc-on

Turns automatic garbage collection on. Garbage collection will happen when and as needed. Automatic garbage collection is off by default.

Since garbage collection works by copying, you are asked for confirmation if there may not be enough space to complete a garbage collection even if it is started immediately.

gc-off

Turns automatic garbage collection off.

gc-on *Variable*

t when garbage collection is on, nil when it is not. You cannot control garbage collection by setting this variable; it exists so you can examine it. In particular, you can tell if the system found it necessary to turn off garbage collection because it was close to running out of virtual memory.

Normally, automatic garbage collection happens in incremental mode; that is, scavenging happens in parallel with computation. Each consing operation scavenges or copies four words per word consed. In addition, scavenging goes on whenever the machine appears idle.

si:inhibit-idle-scavenging-flag *Variable*

If this is non-nil, scavenging is not done during idle time.

If you are running a noninteractive crunching program, the incremental nature of garbage collection may not be helpful. Then you can make garbage collection more efficient by making it a batch process.

si:gc-reclaim-immediately *Variable*

If this variable is non-nil, automatic garbage collection is done as a batch operation: when the garbage collection process decides that the time has come, it copies all the useful data and discards the old address space, running full blast. (It is still possible to use the machine while this is going on, but it is slow.) More specifically, the garbage collection process scavenges and reclaims oldspace immediately right after a flip happens, using all of the machine's physical memory. This variable is only relevant if you have turned on automatic garbage collection with **(gc-on)**.

A batch garbage collection requires less free space than an incremental one. If there is not enough space to complete an incremental garbage collection, you may be able to win by selecting batch garbage collection instead.

si:gc-reclaim-immediately-if-necessary *Variable*

If this variable is non-nil, then automatic garbage collection is done in batch mode if, when the flip is done, there does not seem to be enough space left to do it incrementally. This variable's value is relevant only if **si:gc-reclaim-immediately** is **nil**.

si:gc-flip-ratio *Variable*

This variable tells the garbage collector what fraction of the data it should expect to have to copy, after each flip. It should be a positive number no larger than one. By default, it is one. But if your program is consing considerable amounts of garbage, a value less than one may be safe. The garbage collector uses this variable to figure how much space it will need to copy all the living data, and therefore indirectly how often garbage collection must be done.

si:gc-flip-minimum-ratio *Variable*

This value is used, when non-nil, to control warnings about having too little space to garbage collect. Its value is a positive number no greater than one, just like that of **si:gc-flip-ratio**. The difference between the two is that **si:gc-flip-ratio** controls when garbage collection is *recommended*, whereas **si:gc-flip-minimum-ratio** controls when the system considers the last possible time to do so. If **si:gc-flip-minimum-ratio** is **nil**, **si:gc-flip-ratio** serves both purposes.

Garbage collection is turned off if it appears to be about to run out of memory. You get a notification if this happens. You also get a notification when you are nearly at the point of not having enough space to guarantee garbage collecting successfully.

In addition to turning on automatic garbage collection, you can also manually request one immediate complete collection with the function **si:full-gc**. The usual reason for doing this is to make a band smaller before saving it. **si:full-gc** also resets all temporary areas (see **si:reset-temporary-area**, page 299).

si:full-gc

Performs a complete garbage collection immediately. This does not turn automatic garbage collection on or off; it performs the garbage collection in the process you call it in. A full gc of the standard system takes about 7 minutes, currently.

si:clean-up-static-area *area-number*

This is a more selective way of causing static areas to be garbage collected once. The argument is the area number of a static area; that particular area will be garbage collected the next time a garbage collection is done (more precisely, it will be copied and discarded after the next flip). If you then call **si:full-gc**, it will happen then.

gc-status

The function **gc-status** prints information related to garbage collection. When scavenging is in progress, it tells you how the task is progressing. While scavenging is not in progress and oldspace does not exist, it prints information about how soon a new flip will be required.

While a garbage collection is not in progress, the output from **gc-status** looks like this:

```
Dynamic (new+copy) space 557,417, Old space 0, Static 3,707,242,
Free space 10,453,032, with 10,055,355 needed for garbage collection
 assuming 100% live data (SI:GC-FLIP-RATIO = 1).
If GC is turned on, a flip will happen in 397,677 words.
Scavenging during cons Off, Idle scavenging On,
Automatic garbage collection Off.
GC Flip Ratio 1, GC Reclaim Immediately Off
```

or

```
Dynamic (new+copy) space 561,395, Old space 0, Static 3,707,242,
Free space 10,453,032, with 10,058,670 needed for garbage collection
 assuming 100% live data (SI:GC-FLIP-RATIO = 1).
A flip will happen in 394,362 words.
Scavenging during cons On, Idle scavenging On,
Automatic garbage collection On.
GC Flip Ratio 1, GC Reclaim Immediately Off
```

The "dynamic space" figure is the amount of garbage collectable space and the "static" figure is the amount of static space used. There is no old space since an old space only exists during garbage collection.

The amount of space needed for garbage collection represents an estimate of how much space user programs will use up while scavenging is in progress. It includes a certain amount of padding. The difference between the free space and that amount is how much consing you can do before a garbage collection will begin (if automatic garbage collection is on).

The amount needed for a garbage collection depends on the value of **si:*gc-reclaim-immediately***; more if it is **nil**.

While a garbage collection is in progress, the output looks like this:

```
Incremental garbage collection now in progress.
Dynamic (new+copy) space 45,137, Old space 972,514, Static 3,707,498,
Between 3,701,440 and 4,629,998 words of scavenging left to do.
Free space 9,289,795 (of which 928,558 might be needed for copying).
Ratio scavenging work/free space = 0.55.
Scavenging during cons On, Idle scavenging On,
Automatic garbage collection On.
GC Flip Ratio 1, GC Reclaim Immediately Off
```

Notice that most of the dynamic space has become old space and new space is small. Not much has been copied since the flip took place. The maximum and minimum estimates for the amount of scavenging are based on different limits for how much of old space may need to be copied; as scavenging progresses, the maximum decreases steadily, but the minimum may increase. The free space is smaller now, but it will get larger when scavenging is finished and old space is freed up. (The total amounts are not the same now because unused parts of regions may not be included in any of the figures.)

si:set-scavenger-ws *number-of-pages*

Incremental scavenging is restricted to a fixed amount of physical memory to reduce its interference with your other activities.

This function specifies the number of pages of memory that incremental garbage collection can use. 256 is a good value for a 256k machine. If the garbage collector gets very poor paging performance, use of this function may fix it.

35.8 Logging In

Logging in tells the Lisp Machine who you are, so that other users can see who is logged in, you can receive messages, and your INIT file can be run. An INIT file is a Lisp program which gets loaded when you log in; it can be used to set up a personalized environment.

When you log out, it should be possible to undo any personalizations you have made so that they do not affect the next user of the machine. Therefore, anything done by an INIT file should be undoable. In order to do this, for every form in the INIT file, a Lisp form to undo its effects should be added to the list that is the value of logout-list. The login-forms construct helps make this easy; see below.

user-id *Variable*

The value of user-id is either the name of the logged in user, as a string, or else an empty string if there is no user logged in. It appears in the who-line.

logout-list *Variable*

The value of logout-list is a list of forms to be evaluated when the user logs out.

login *name* &optional *host inhibit-init-file*

Sets your name (the variable **user-id**) to *name* and logs in a file server on *host*. *host* also becomes your default file host. The default value of *host* depends on which Lisp Machine you use using; it is called the associated machine (see page 815). **login** also runs the **:login** initialization list (see page 773).

If *host* requires passwords for logging in you are asked for a password. Adding an asterisk at the front of your password enables any special capabilities you may be authorized to use, by calling **fs:enable-capabilities** (page 609).

Unless *inhibit-init-file* is specified as non-**nil**, **login** loads your init file if it exists. On ITS, your init file is *name* **LISPM** on your home directory. On TOPS-20 your init file is **LISPM.INIT** on your directory. On VMS, it is **LISPM.INI**. On Unix, it is **lispm.init**.

If anyone is logged into the machine already, **login** logs him out before logging in *name*. (See **logout**.) Init files should be written using the **login-forms** construct so that **logout** can undo them. Usually, however, you cold-boot the machine before logging in, to remove any traces of the previous user. **login** returns **t**.

log1 &rest *options*

Like **login** but the arguments are specified differently. *options* is a list of keywords and values; the keywords **:host** and **:init** specify the host to log in on and whether to load the init file if any. Any other keywords are also allowed. **log1** itself ignores them, but the init file can act on them. The purpose of **log1**, as opposed to **login**, is to enable you to specify other keywords for your init file's sake.

si:user-init-options *Variable*

During the execution of the user's init file, inside **log1**, this variable contains the arguments given to **log1**. Options not meaningful to **log1** itself can be specified, so that the init file can find them here and act on them.

logout

First, **logout** evaluates the forms on **logout-list**. Then it sets **user-id** to an empty string and **logout-list** to **nil**. Then it runs the **:logout** initialization list (see page 773), and returns **t**.

login-forms *undoable-forms...* *Macro*

The body of a **login-forms** is composed of forms to be evaluated, whose effects are to be undone if you log out. For example,

```
(login-forms
  (setq fs:*defaults-are-per-host* t))
```

would set the variable immediately but arrange for its previous value to be restored if you log out.

login-forms is not an AI program; it must be told how to undo each function that will be used immediately inside it. This is done by giving the function name (such as **setq**) a **:undo-function** property which is a function that takes a form as an argument and returns a form to undo the original form. For **setq**, this is done as follows:

```
(defun (setq :undo-function) (form &aux results)
  (do ((l (cdr form) (cddr l)))
      ((null l))
    (cond ((boundp (car l))
           (push '(setq ,(car l) ',(symeval (car l))) results))
          (t (push '(makunbound ',(car l)) results))))
  '(progn . ,results))
```

Undo functions are standardly provided for the functions setq, pkg-goto-globally, setq-globally, add-initialization, deff, defun, defsubst, macro, advise and zwei:set-comtab. Constructs which macroexpand into uses of those functions are also supported.

Note that setting *read-base* and *print-base* should be done with setq-globally rather than setq, since those variables are likely to be bound by the load function while the init file is executed.

login-setq {*variable value*}... *Macro*

login-setq is like setq except that it puts a setq form on logout-list to set the variables to their previous values. login-setq is obsolete; use login-forms around a setq instead.

login-eval *x*

login-eval is used for functions that are "meant to be called" from INIT files, such as zwei:set-comtab-return-undo, which conveniently return a form to undo what they did. login-eval pushes the result of the form *x* onto logout-list. It is obsolete now because login-forms is a cleaner interface.

si:undoable-forms-1 *undo-list-name forms* &optional *complaint-string*

This is what login-forms uses. *forms* is a list of forms; they are evaluated and forms for undoing their effects are pushed onto the value of the symbol *undo-list-name*. If an element of *forms* has no known way to be undone, a message is printed using the string *complaint-string*. For login-forms, the string supplied is "at logout".

35.9 Dribble Files

dribble &optional *filename*

With an argument, dribble opens *filename* as a 'dribble file' (also known as a 'wallpaper file') and then enters a Lisp listen loop in which *standard-input* and *standard-output* are rebound to direct all the output and echoing they do to the file as well as to the terminal.

Dribble output can be sent to an editor buffer by using a suitable pathname; see section 24.7.6, page 575.

Calling dribble with no arguments terminates dribbling; it throws to the original call to dribble, which closes the file and returns.

dribble-all &optional *filename*

Like dribble except that all input and output goes to the dribble file, including break loops, queries, warnings and sessions in the debugger. This works by binding *terminal-io* instead of *standard-output* and *standard-input*.

35.10 Version Information

Common Lisp defines several standard ways of inquiring about the identity and capabilities of the Lisp system you are using.

features *Variable*

A list of atoms which describe the software and hardware features of the Lisp implementation. By default, this is

```
(:loop :defstruct :lispm :cadr :mit :chaos :sort :fasload :string
 :newio :roman :trace :grindef :grind :common)
```

Most important is the symbol :lispm; this indicates that the program is executing on the Lisp Machine. :cadr indicates the type of hardware, :mit which version of the Lisp Machine operating system, and :chaos that the Chaosnet protocol is available. :common indicates that Common Lisp is supported.

Most of the other elements are for Maclisp compatibility. Common Lisp defines the variable *features* but does not define what should appear in the list. The order of elements in the list has no significance. Membership checks should use string-equal so that packages are not significant

The #+ and #- read constructs (page 525) check for the presence of an element in this list. Thus, #+lispm when read by a Lisp Machine causes the following expression to be significant, because :lispm is present in the features list.

The remaining standard means of inquiry are specified by Common Lisp to be functions rather than variables, for reasons that seem poorly thought out.

lisp-implementation-type

Returns a string saying what kind of Lisp implementation you are using. On the Lisp Machine it is always "Zetalisp".

lisp-implementation-version

Returns a string saying the version numbers of the Lisp implementation. On the Lisp Machine it looks something like

"System 98.3, CADR 3.0, ZMAIL 52.2".

machine-type

Returns a string describing the kind of hardware in use. It is "CADR" or "LAMBDA".

machine-version
Returns a string describing the kind of hardware and microcode version. It starts with the value of machine-type. It might be "CADR Microcode 309".

machine-instance
Returns a string giving the name of this machine. Do not be confused; the value is a string, not an instance. Example: "CADR-18".

software-type
Returns a string describing the type of operating system software that Lisp is working with. On the Lisp Machine, it is always "Zetalisp", since the Lisp Machine Lisp software is the operating system.

software-version
Returns a string describing the version numbers of the operating system software in use. This is the same as lisp-implementation-version on the Lisp Machine since the same software is being described.

short-site-name
Returns a string giving briefly the name of the site you are at. A site is an institution which has a group of Lisp Machines. The string you get is the value of the :short-site-name site option as given in SYS: SITE; SITE LISP. See section 35.12, page 810 for more information. Example: "MIT AI Lab".

long-site-name
Returns a string giving a verbose name for the site you are at. This string is specified by the site option :long-site-name. Example: "Massachusetts Institute of Technology, Artificial Intelligence Laboratory".

35.11 Booting and Disk Partitions

A Lisp Machine disk is divided into several named *partitions* (also called *bands* sometimes). Partitions can be used for many things. Every disk has a partition named PAGE, which is used to implement the virtual memory of the Lisp Machine. When you run Lisp, this is where the Lisp world actually resides. There are also partitions that hold saved images of the Lisp Machine microcode, conventionally named MCR*n* (where *n* is a digit), and partitions that hold saved images of Lisp worlds, conventionally named LOD*n*. A saved image of a Lisp world is also called a *virtual memory load* or *system load*. The microcode and system load are stored separately so that the microcode can be changed without going through the time-consuming process of generating a new system load.

The directory of partitions is in a special block on the disk called the label. The label names one of the partitions as the current microcode and one as the current system load. When you cold-boot, the contents of the current microcode band are loaded into the microcode memory, and then the contents of the current saved image of the Lisp world is copied into the PAGE partition. Then Lisp starts running. When you warm-boot, the contents of the current microcode band are loaded, but Lisp starts running using the data already in the PAGE partition.

For each partition, the directory of partitions contains a brief textual description of the contents of the partition. For microcode partitions, a typical description might be "UCADR 310"; this means that version 310 of the microcode is in the partition. For saved Lisp images, it is a little more complicated. Ideally, the description would say which versions of which systems are loaded into the band. Unfortunately, there isn't enough room for that in most cases. A typical description is "99.4 Daed 5.1", meaning that this band contains version 99.4 of System and version 5.1 of Daedalus. The description is created when a Lisp world is saved away by disk-save (see below).

35.11.1 Manipulating the Label

print-disk-label &optional (*unit* 0) (*stream* *standard-output*)
Prints a description of the label of the disk specified by *unit* onto *stream*. The description starts with the name of the disk pack, various information about the disk that is generally uninteresting, and the names of the two current load partitions (microcode and saved Lisp image). This is followed by one line of description for each partition. Each one has a name, disk address, size, and textual comment. The current microcode partition and the current system load partition are marked with asterisks, each at the beginning of the line.

unit may be the unit number of the disk (most Lisp machines just have one unit, number 0), or the host name of another Lisp Machine on the Chaosnet, as a string (in which case the label of unit 0 on that machine is printed, and the user of that machine is notified that you are looking at his label), or, for CADRs only, the string "CC" (which prints the label of unit 0 of the machine connected to this machine's debugging hardware).

Use of "CC" as the *unit* is the way to examine or fix up the label of a machine which cannot work because of problems with the label. On a Lambda, this must be done through the SDU.

set-current-band *partition-name* &optional (*unit* 0)
Sets the current saved Lisp image partition to be *partition-name*. If *partition-name* is a number, the name LOD*n* is used.

unit can be a disk drive number, the host name of another Lisp Machine, or the string "CC". See the comments under print-disk-label, above.

If the partition you specify goes with a version of microcode different from the one that is current, this function offers to select the an appropriate microcode partition as well. Normally you should answer Y.

set-current-microload *partition-name* &optional (*unit* 0)
Sets the current microcode partition to be *partition-name*. If *partition-name* is a number, the name MCR*n* is used.

unit can be a disk drive number, the host name of another Lisp Machine, or the string "CC". See the comments under print-disk-label, above.

si:current-band &optional (*unit* 0)
si:current-microload &optional (*unit* 0)

Return, respectively, the name of the current band and the current microload on the specified unit.

When using the functions to set the current load partitions, be extra sure that you are specifying the correct partition. Having done it, cold-booting the machine will reload from those partitions. Some versions of the microcode will not work with some versions of the Lisp system, and if you set the two current partitions incompatibly, cold-booting the machine will fail. To fix this, on a CADR, use another CADR's debugging hardware, running print-disk-label and set-current-band on the other CADR and giving "CC" as the *unit* argument. On a Lambda, this is done via the SDU.

si:edit-disk-label *unit* &optional *init-p*

Runs an interactive label editor on the specified unit. This editor allows you to change any field in the label. The Help key documents the commands. You have to be an expert to need this and to understand what it does, so the commands are not documented here. Ask someone if you need help. You can screw yourself very badly with this function.

disk-restore &optional *partition*

Allows booting from a band other than the current one. *partition* may be the name or the number of a disk partition containing a virtual-memory load, or nil or omitted, meaning to use the current partition. The specified partition is copied into the paging area of the disk and then started.

Although you can use this to boot a different Lisp image than the installed one, this does not provide a way to boot a different microcode image. disk-restore brings up the new band with the currently running microcode.

disk-restore asks the user for confirmation before doing it.

describe-partition *partition* &optional *unit*

Tells you various useful things about a partition; including where on disk the partition begins, and how long it is.

If you specify a saved Lisp system partition, such as LOD3, it also tells you important information about the contents of the partition: the microcode version which the partition goes with, the size of the data in the partition and the highest virtual address used. The size of the partition tells how large a partition you need to make a copy of this one, and the highest virtual address used (which is measured in units of disk blocks) tells you how large a PAGE partition you need in order to run this partition.

35.11.2 Updating Software

Of all the procedures described in this section, the most common one is to take a partition containing a Lisp image, update it to have all the latest patches (see section 28.8, page 672), and save it away in a disk partition. The function load-and-save-patches does it all conveniently for you.

load-and-save-patches

Loads patches and saves a band, with a simple user interface. Run this function immediately after cold booting, without logging in first; it logs in automatically as LISPM (or whatever is specified in the site files). The first thing it does is print the list of disk partitions and ask you which one to save in. Answer LOD*n*, using the name of a partition from the printed list. You must then confirm. Then the patches are loaded and the resulting world is saved with no further user interaction, as long as no problem arises.

It is convenient to use this function just before you depart, allowing it to finish unattended.

If you wish to do something other than loading all and only the latest patches, you must perform the steps by hand. Start by cold-booting the machine, to get a fresh, empty system. Next, you must log in as something whose INIT file does not affect the Lisp world noticably (so that when you save away the Lisp image, the side-effects of the INIT file won't get saved too); on MIT-OZ, for example, you can log in as LISPM with password LISPM. Now you can load in any new software you want; usually you should also do (load-patches) for good measure. You may also want to call si:set-system-status to change the release status of the system.

When you're done loading everything, do (print-disk-label) to find a band in which to save your new Lisp world. It is best not to reuse the current band, since if something goes wrong during the saving of the partition, while you have written, say, half of the band that is current, it may be impossible to cold-boot the machine. Once you have found the partition, you use the disk-save function to save everything into that partition.

disk-save *partition-name* &optional *no-query incremental*

Saves the current Lisp world in the designated partition. *partition-name* may be a partition name (a string), or it may be a number in which case the name LOD*n* is used.

The user is first asked for yes-or-no confirmation that he really wants to reuse the named partition. A non-nil value for *no-query* prevents this question. This is only for callers that have already asked.

Next it is necessary to figure out what to put into the textual description of the band, for the disk label. This starts with the brief version of si:system-version-info (see page 674). Then comes a string of additional information; if *no-query* is nil, the user is offered the chance to provide a new string. The current value of this string is returned by si:system-version-info and printed by booting. The version info and the string both go in the comment field of the disk label for this band. If they don't together fit into the fixed size available, the user is asked to retype the whole thing (the version info as well as your comment) in a compressed form that does fit.

The Lisp environment is then saved away into the designated partition, and then the equivalent of a cold-boot from that partition is done.

Once the patched system has been successfully saved and the system comes back up, you can make it current with set-current-band.

When you do a disk-save, it may tell you that the band you wish to save in is not big enough to hold all the data in your current world. It may be possible for you to reduce the size of the data so that it will fit in that band, by garbage collecting. Simply do (si:full-gc).

Try to avoid saving patched systems after running the editor or the compiler. This works, but it makes the saved system a lot bigger. In order to produce a clean saved environment, you should try to do as little as possible between the time you cold-boot and the time you save the partition.

si:login-history *Variable*

The value of si:login-history is a list of entries, one for each person who has logged into this world since it was created. This makes it possible to tell who disk-saved a band with something broken in it. Each entry is a list of the user ID, the host logged into, the Lisp Machine on which the world was being executed, and the date and time.

35.11.3 Saving Personal Software

If you have a large application system which takes a while to load, you may wish to save a band containing it.

To do this, boot a fresh band, log in without running your init file, do make-system to load the application system, and then invoke disk-save. When disk-save asks for an additional comment, give your name or the name of the application system you loaded, and a date. This will tell other people who to ask whether the band is still in use if they would like to save other things.

You can greatly reduce the amount of disk space needed for the saved band by making it an *incremental* band; that is, a band which contains the differences between the Lisp world you want to save and the system band you originally loaded. Since all the pages of the system which your application program did not change do not have to be saved, an incremental band is generally much smaller—perhaps by a factor of ten.

To make an incremental band, give a non-nil third argument to disk-save, as in

```
(disk-save "lod4" nil t)
```

Figuring out which pages need to be saved in the incremental band takes a couple of extra minutes.

You can restore the incremental band with disk-restore or boot it like any other band. This works by first booting the original band and then copying in the differences that the incremental band records. It takes only a little longer than booting the original system band.

The original band to which an incremental band refers must be a complete load. When you update a standard system band (loading patches, for instance) you should always make a complete load, so that the previous system band is not needed for the new one to function.

The incremental band records the partition name of the original system band. That original band must still exist, with the same contents,. in order for the incremental band to work properly. The incremental band contains some error check data which is used to verify this. The error checking is done by the microcode when the incremental band is booted, but it is also done by set-current-band, so that you are not permitted to select an incremental band if it is not going to work.

When using incremental bands, it is important to preserve the system bands that they depend on. Therefore, system bands should not be updated too frequently. describe-partition on an incremental band says which full band it depends on; you can use this to determine which bands should be kept for the sake of incremental bands that depend on them.

In order to realize the maximum savings in disk space possible because of incremental bands, you must make the partition you saved in smaller once the save is finished and you know how much space was actually used. This is done with si:edit-disk-label. The excess space at the end of the partition can be used to make another partition which is used for the next incremental band saved. Eventually when some of the incremental bands are no longer needed the rest must be shuffled so that the free space can be put together into larger partitions. This can be done with si:copy-disk-partition.

An easier technique is to divide a couple of the initial partitions into several equal-sized partitions of about 4000 pages, and use these for all incremental saving. You can easily provide room for 12 incremental bands this way in addition to a few system bands and file system.

You must not do a garbage collection to reduce the size of the world before you make an an incremental band. This is because garbage collection alters so many pages that an incremental band would be as big as a complete band.

35.11.4 Copying Bands

The normal way to install new software on a machine is to copy the microcode and world load bands from another machine.

The first step is to find a machine that is not in use and has the desired system. Let us call this the source machine. The machine where the new system is to be installed is the target machine. You can use finger to see which machines are free, and use print-disk-label with an argument to examine the label of that machine's disk and see if it has the system you want.

Then you should do a (print-disk-label) to find suitable partitions to copy them into. It is advisable not to copy them into the selected partitions; if you did that, and the machine crashed in the middle, you would be unable to boot it.

Before copying a band from another machine, double-check the partition names by printing the labels of both machines, and make sure no one is using the other machine. Also double-check with describe-partition that the world load and microcode go together. Then use this function:

si:receive-band *source-host source-band target-band* &optional *subset-start subset-size*

Copies the partition on *source-host*'s partition named *source-band* onto the local machine's partition named *target-band*. This takes about ten minutes. It types out the size of the partition in pages, and types a number every 100 pages telling how far it has gotten. It displays an entry in the who line on the remote machine saying what's going on.

The *subset-start* and *subset-size* arguments can be used to transfer only part of a partition. They are measured in blocks. The default for the first is zero, and the default for the second is to continue to the end of the data in the band. These arguments are useful for restarting a transfer that was aborted due to network problems or a crash, based on the count of hundreds of blocks that was printed out before the crash.

To go the other direction, use si:transmit-band.

si:transmit-band *source-band target-host target-band* &optional *subset-start subset-size*

This is just like si:receive-band, except you use it on the source machine instead of the target machine. It copies the local machine's partition named *source-band* onto *target-machine*'s partition named *target-band*.

It is preferable to use si:receive-band so that you are present at the machine being written on.

After transferring the band, it is good practice to make sure that it really was copied successfully by comparing the original and the copy. All of the known reasons for errors during band transfer have (of course) been corrected, but peace of mind is valuable. If the copy was not perfectly faithful, you might not find out about it until a long time later, when you use whatever part of the system that had not been copied properly.

si:compare-band *source-host source-band target-band* &optional *subset-start subset-size*

This is like si:receive-band, except that it does not change anything. It compares the two bands and complains about any differences.

Having gotten the current microcode load and system load copied into partitions on your machine, you can make them current for booting using set-current-band.

35.12 Site Options and Host Table

The Lisp Machine system has options that are set at each site. These include the network addresses of other hosts, which hosts have file servers, which host to find the system source files and patch files on, where to send bug reports, what timezone the site is located in, and many other things.

The per-site information is defined by three files: SYS: SITE; SITE LISP, SYS: SITE; LMLOCS LISP, and SYS: CHAOS; HOSTS TXT.

SYS: CHAOS; HOSTS TXT is the network host table. It gives the names and addresses of all hosts that are to be known to the Lisp Machine for any purposes. It also says what type of machine the host is, and what operating system runs on it.

SYS: SITE; LMLOCS LISP specifies various information about the Lisp Machines at your site, including its name, where it is physically located, and what the default machine for logging in should be.

SYS: SITE; SITE LISP specifies all other site-specific information. Primarily, this is contained in a call to the special form **defsite**.

defsite *site-name* (*site-option value*)... *Macro*

This special form defines the values of site-specific options, and also gives the name of the site. Each *site-option* is a symbol, normally in the keyword package, which is the name of some site option. *value* is the value for that option; it is evaluated. Here is a list of standardly defined site options:

:sys-host
The value is a string, the name of the host on which the system source files are stored. This host becomes the translation of logical host SYS.

:sys-host-translation-alist
The value is an alist mapping host names into translation-list variables. Each translation list variable's value should be an alist suitable for being the third argument to fs:add-logical-pathname-host (see page 574). The car of an element may be nil instead of a host name; then this element applies to all hosts not mentioned.

The normal place to find the system sources is on the host specified by the :sys-host keyword, in the directories specified by the translation list variable found by looking that host up in the value of the :sys-host-translation-alist keyword. If you specify a different host as the system host with si:set-sys-host, that host is also looked up in this alist to find out what directories to use there.

Here is what is used at MIT:

```
(defsite :mit
  ...
  (:sys-host-translation-alist
    '(("AI" . its-sys-pathname-translations)
      ("OZ" . oz-sys-pathname-translations)
      ("FS" . its-sys-pathname-translations)
      ("LM" . its-sys-pathname-translations)
      (nil . its-sys-pathname-translations)))
  ...)
```

```
(defconst oz-sys-pathname-translations
  '(("CC;" "<L.CC>")
    ("CHAOS;" "<L.CHAOS>")
    ("DEMO;" "<L.DEMO>")
    ...
    ("SITE;" "<L.SITE>")
    ("SYS;" "<L.SYS>")
    ("SYS2;" "<L.SYS2>")
    ...
    ("ZMAIL;" "<L.ZMAIL>")
    ("ZWEI;" "<L.ZWEI>")
    ))
```

:sys-login-name
:sys-login-password
These specify the username and password to use to log in automatically to read system patch files, microcode symbol tables and error tables. The values should be strings.

:chaos nil if the site has no Chaosnet; otherwise, a string, the name of the Chaosnet that the site is on. Names for Chaosnets will eventually be used to permit communication between Chaosnets, probably through special gateway servers. Except when multiple sites are on a single Chaosnet, normally the Chaosnet name should be the same as the site name (but as a string, not a symbol).

:standalone The value should be t for a Lisp Machine that is operated without a network connection. This causes the Lisp Machine to not to try to use the Chaosnet for getting the time. On the Lambda, the time will obtained from the SDU's clock. On the CADR, the time will be obtained from the user.

:default-associated-machine
This should be a string which is the name of a host to use as the associated host for any Lisp Machine not mentioned in the LMLOCS file.

:usual-lm-name-prefix
This should be a string which is the typical beginning of host names of Lisp Machines at your site. At MIT, it is "CADR-".

:chaos-file-server-hosts
This should be a list of names of hosts that have file servers, including Lisp Machines which other Lisp Machines should know about.

:lmfile-server-hosts
This should be a list of names of Lisp Machines that provide servers for the LMFILE file system. The entry for such a machine should be one of the nicknames of that machine. By virtue of its presence in this list, it becomes the name by which the LMFILE file system there can be accessed remotely.

:chaos-time-server-hosts
This should be a list of names of hosts that support TIME servers. These

are hosts that the Lisp Machine can ask the time of day from when you boot.

:chaos-host-table-server-hosts
This should be a list of names of hosts that support host-table servers, which can be used to inquire about hosts on networks that the Lisp Machine does not know about in its own host table.

:chaos-mail-server-hosts
This should be a list of names of hosts that support mail servers which are capable of forwarding mail to any known host.

:timezone This should be a number, the number of hours earlier than GMT of standard time in the timezone where this site is located.

:host-for-bug-reports
This should be a string, the name of the host at which bug-report mailboxes are located.

:local-mail-hosts
This should be a list of names of hosts that ZMail should consider "local" and omit from its summary display.

:spell-server-hosts
This should be a list of hosts that have spelling corrector servers.

:comsat This should be **t** if mail can be sent through the COMSAT mail demon. This is true only at MIT.

:default-mail-mode
This should be the default mode for use in sending mail. The options are :file (use COMSAT), :chaos (use one of the :chaos-mail-server-hosts), or :chaos-direct (like :chaos, but go direct to the host that the mail is addressed to whenever possible).

:gmsgs This should be **t** if GMSGS servers are available.

:arpa-gateways
This should be a list of names of hosts that can be used as gateways to the Arpanet. These hosts must provide a suitable Chaosnet server which will make Arpanet connections. It should be **nil** if your site does not have an Arpanet connection.

:arpa-contact-name
If you have Arpanet gateways, this is the Chaosnet contact name to use. Nowadays, it should be "TCP".

:dover This should be **t** if your site has a Dover printer.

:default-printer
This should be a keyword which describes the default printer for hardcopy commands and functions to use. Possible values include :dover, **nil**, or any other printer type that you define (see section 35.2, page 785).

:default-bit-array-printer
Like :default-printer, but this is the default for only **hardcopy-bit-array**

to use.

:esc-f-arg-alist
: This says what various numeric arguments to the Terminal F command mean. It is a list of elements, one for each possible argument. The car of an element is either a number or nil (which applies to Terminal F with no argument). The cdr is either :login (finger the login host), :lisp-machines (finger all Lisp Machines at this site), :read (read some hosts from the keyboard), or a list of host names.

:verify-lm-dumps
: If the value is t, Lisp Machine file system dump tapes are verified.

Other site options are allowed, and your own software can look for them.

35.12.1 Updating Site Information

To update the site files, you must first recompile the sources. Do this by

```
    (make-system 'site 'compile)
```

This also loads the site files.

To just load the site files, assuming they are compiled, do

```
    (make-system 'site)
```

load-patches does that automatically.

You should never load any site file directly. All the files must be loaded in the proper fashion and sequence, or the machine may stop working.

35.12.2 Accessing Site Options

Programs examine the site options using these variables and functions:

site-name *Variable*

The value of this variable is the name of the site you are running at, as defined in the defsite in the SITE file. You can use this in run-time conditionals for various sites.

get-site-option *keyword*

Returns the value of the site option *keyword*. The value is nil if *keyword* is not mentioned in the SITE file.

define-site-variable *variable keyword [documentation]* *Macro*

Defines a variable named *variable* whose value is always the same as that of the site option *keyword*. When new site files are loaded, the variable's value is updated. *documentation* is the variable's documentation string, as in defvar.

define-site-host-list *variable keyword* [*documentation*] *Macro*

Defines a variable named *variable* whose value is a list of host objects specified by the site option *keyword*. The value actually specified in the SITE file should be a list of host names. When new site files are loaded, the variable's value is updated. *documentation* is the variable's documentation string, as in **defvar**.

35.12.3 The LMLOCS File

The LMLOCS file contains an entry for each Lisp Machine at your site, and tells the system whatever it needs to know about the particular machine it is running on. It contains one form, a defconst for the variable machine-location-alist. The value should have an element for each Lisp Machine, of this form:

```
("MIT-LISPM-1"  "Lisp Machine One"
 "907 [Son of CONS] CADR1's Room x6765"
 (MIT-NE43 9) "OZ" ((:default-printer :dover)))
```

The general pattern is

```
(host-full-name pretty-name
 location-string
 (building floor) associated-machine site-options)
```

The *host-full-name* is the same as in the host table.

The *pretty-name* is simply for printing out for users on certain occasions.

The *location-string* should say where to find the machine's console, preferably with a telephone number. This is for the FINGER server to provide to other hosts.

The *building* and *floor* are a somewhat machine-understandable version of the location.

The *associated-machine* is the default file server host name for login on this Lisp Machine.

site-options is a list of site options, just like what goes in the defsite. These site options apply only to the particular machine, overriding what is present in the SITE file. In our example, the site option :default-printer is specified as being :dover, on this machine only.

si:associated-machine *Variable*

The host object for the associated machine of this Lisp Machine.

Concept Index

Flavor Index

Operation Index

Keyword Index

Object Creation Options

Meter Index

Variable Index

Function Index

Condition Name Index