Lisp Machine Manual

Second Preliminary Version

January 1979

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 N00014-75-C-0643.

Preface

This is a preliminary version of the Lisp Machine manual, describing both the dialect of Lisp used by the Lisp Machine, and the software environment of the Lisp Machine system. Several chapters have not been written, due to the early stage of the software they describe: this includes chapters on Graphics, the Mouse, Menus, the Eine editor, the network control program and higher level network programs, and the file system. Some of the chapters that are included describe software that is still in a state of flux, and are likely to change drastically in the next revision of this manual. The authors also plan to produce a document describing the internal formats of data objects and the instruction set of the machine.

This version of the Lisp machine manual contains only minor editorial corrections from the version of November, 1978.

Any comments, suggestions, or criticisms will be welcomed. The authors can be reached by any of the following communication paths:

ARPA Network mail to BUG-LMMAN@MIT-AI

U.S. Mail to
Daniel L. Weinreb or David A. Moon
545 Technology Square
Cambridge, Mass. 02139

MIT Multics mail to Weinreb.SIPB

Note

The software described herein was written by the Lisp Machine Group, whose current members are Alan Bawden, Bruce Edwards, Richard Greenblatt, Jack Holloway, Thomas Knight, Michael McMahon, David Moon, Michael Patton, Richard Stallman, and Daniel Weinreb.

This document was edited with the Emacs editor, and formatted by the Bolio text justifier. It was printed on the MIT A.I. Lab's Xerox Graphic Printer.

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: each person gets his own for the duration of the session. It is designed this way to relieve the problems of the running of 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.

This document is intended to serve both as a User's Guide and as a Reference Manual for the language and the Lisp Machine itself. It is hoped that anyone with some previous programming experience (not necessarily in Lisp) could learn all about the Lisp language and the Lisp Machine from this manual.

This is a *preliminary* version of the Manual. The authors are well aware that several sections are missing. Some small sections were left out in the interest of publishing a manual as quickly as possible. Several full chapters have not been written because the corresponding software has not settled down enough for a meaningful document to be written; these include chapters on the Chaos network, the mouse, and menus. Many more major software changes are expected in both the language and the system; this manual is far from the last word.

The Lisp Machine executes a new dialect of Lisp called Lisp Machine Lisp, developed at the M.I.T. Artificial Intelligence Laboratory for use in artificial intelligence research and related fields. It is closely related to 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, is based on Lisp 1.5.

1.2 Structure of the Manual

This manual attempts to document both the dialect of Lisp used on the Lisp Machine, and the system itself. The manual starts out with an explanation of the language. Chapter 2 presents some basic *predicate* functions, Chapter 3 explains the process of evaluation, and Chapter 4 introduces the basic Lisp control structures.

Next, in Chapters 4 through 12, various Lisp data types are presented, along with functions for manipulating objects of those types. These nine chapters discuss list structure, symbols, numbers, strings, arrays, closures, stack groups, and locatives.

Chapter 13 explains the "subprimitive" functions, which are primarily useful for implementation of the Lisp language itself and the Lisp Machine's "operating system". Chapter 14 explains *areas*, which give the programmer control over storage and locality of

reference.

Chapter 15 discusses the Lisp compiler, which converts Lisp programs into "machine language". Chapter 16 explains the Lisp macro facility, which allows users to write their own extensions to Lisp, and Chapter 17 goes into detail about one such extension that provides *structures*.

Chapter 18 explains the Lisp Machine's Input/Output system, including *streams* and the *printed representation* of Lisp objects. Chapter 19 describes the *package* system, which allows many name spaces within a single Lisp environment. Chapter 20 talks about how files from a file system are used from Lisp.

Chapter 21 discusses the *job system*, which allows shared access to the TV screen, and multiple processes. Chapter 22 goes into detail on the TV display itself. Chapter 23 explains how exceptional conditions (errors) can be handled by programs, handled by users, and debugged. Chapter 24 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, particularly the reference sections, to avoid confusion.

Most numbers shown are in octal radix (base eight). Spelled out numbers and numbers followed by a decimal point are in decimal. This is because, by default, Lisp Machine Lisp types out numbers in base 8; don't be surprised by this. To change it, see the documentation on the symbols **ibase** and **base** (page 157).

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

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

There are several terms which 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 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 preceeded by a "/" it is said to be *slashified*. Slashifying also turns off the effects of macro characters such as "'" and ";".

The following characters also have special meanings, and may not be used in symbols without slashification. These characters are explained in detail in the section on printed-representation (page 156).

"	String quote
#	Introduces miscellaneous reader macros
,	See `
:	Package prefix
`	List structure construction
|	Symbol quoter
⊗	Octal escape

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.

By convention, all "keyword" symbols in the Lisp machine system have names starting with a colon (:). The colon character is not actually part of the print name, but is a package prefix indicating that the symbol belongs to the package with a null name, which means the **user** package. If you are not using packages, that is, you are doing everything in the **user** package, it is not necessary to type the colon. However, it is recommended that you always put in the colon so that you will not have problems if you later put your program into a package. But it is necessary to leave out the colon in programs that must run in both Maclisp and Lisp Machine Lisp. The colon can usually be omitted when you are simply typing at the top-level of Lisp, since your typein is being read in the **user** package, but it is better to type it so you will get used to it. In this manual the colon will always be included.

Lisp Machine Lisp is descended from Maclisp, and a good deal of effort was gone through to try to allow Maclisp programs to run in Lisp Machine Lisp. There is an extensive section explaining the differences between the dialects, and how to convert Maclisp programs to work in the Lisp Machine. 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 all detail elsewhere in the manual. 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.

When the text speaks of "typing Control-Q" (for example), this means to hold down the CTRL key on the keyboard (either of the two), 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 CTRL and META. Unlike the PDP-10, there are no "control characters" in the character set; Control and Meta are merely things that can be typed on the keyboard.

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

The rest of this chapter explains more of the details of the Lisp Machine Lisp dialect. This section is also suitable for the Maclisp user, as it goes into detail about important differences between the dialects. Those Maclisp users who have skipped the previous sections should definitely read this one.

1.4 Data Types

This section enumerates the various different types of objects in Lisp Machine Lisp. The types explained below include symbols, conses, various types of numbers, two kinds of compiled code object, locatives, arrays, stack groups, and closures. With each is given the associated symbolic name, which is returned by the function **data-type** (page 111).

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

The print name is a string, which may be obtained by the function **get-pname** (page 65). This string serves as the *printed representation* (see page 154) of the symbol. The binding (sometimes also called the "value") may be any object. 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 binding. It is accessed by the **symeval** function (page 58), and updated by the **set** function (page 57). (That is, given a symbol, you use **symeval** to find out what its binding is, and use **set** to change its binding.) The definition 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 59), and updated with **fset** (page 59). The property list is a list of an even number of elements; it can be accessed directly by **plist** (page 64), and updated directly by **setplist** (page 64), although usually the functions **get**, **putprop**, and **remprop** (page 63) 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 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 and can be disregarded by the casual user.

The primitive function for creating symbols is **make-symbol** (page 66) (currently named **make-atom**), although most symbols are created by **read**, **intern**, or **fasload** (who 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 38), and updated with **rplaca** and **rplacd** (page 47). The primitive function for creating conses is **cons** (page 39).

There are several kinds of numbers in Lisp Machine Lisp. *Fixnums* represent integers in the range of -2^23 to 2^23-1. *Bignums* represent integers of arbitrary size, with more overhead than fixnums. The system automatically converts between fixnums and bignums as required. *Flonums* are floating-point numbers. *Small-flonums* are another kind of floating-point numbers, with less range and precision, but less computational overhead. Other types

of numbers are likely to be added in the future. See page 68 for full details.

The usual form of compiled code is a Lisp object called a "Function Entry Frame" or "FEF". 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 (page 126), 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 Lisp object which represents executable code is a "micro-code entry". These are the microcoded primitive functions of the Lisp system, and user functions compiled into microcode.

About the only useful thing to do with one of these objects is to *apply* it to arguments. However, some functions are provided for examining such objects, for user convenience. See **arglist** (page 61), **args-info** (page 61), **describe** (page 261), and **disassemble** (page 263).

A *locative* (see page 109) is a kind of a pointer to a single cell anywhere in the system. The contents of this cell can be accessed by either **car** or **cdr** (both do the same thing for a locative) (see page 38) and updated by either **rplaca** or **rplacd** (see page 47).

An *array* (see page 88) is a set of cells indexed by a tuple of integer subscripts. The contents of cells may be accessed and changed individually. There are several types of arrays. Some have cells which may contain any object, while others (numeric arrays) may only contain small positive numbers. Strings are a type of array; the elements are 8-bit positive numbers which encode characters.

1.5 Lambda Lists

Note: the examples in this section are examples of lambda-lists, not of Lisp forms!

A *lambda-expression* is the form of a user-defined function in Lisp. It looks like (**lambda** *lambda-list . body*). The *body* may be any number of forms. In Maclisp and Lisp 1.5, the *lambda-list* (also called a *bound-variable list*) is simply a list of symbols (which act like *formal parameters* in some other languages). When the lambda-expression is *applied* to its arguments (which act like *actual parameters* in other languages), the symbols are bound to the arguments, and the forms of the body are evaluated sequentially; the result of the last of these evaluations is returned. If the number of arguments is not the same as the length of the lambda-list, an error is generated.

In Lisp Machine Lisp the same simple lambda-lists may be used, but there are additional features accessible via certain keywords (which start with &) and by using lists as elements of the lambda-list.

The principle weakness of the simple scheme is that any function must only take a certain, fixed number of arguments. As we know, many very useful functions, such as **list**, **append**, **+**, and so on, may take 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, Lisp Machine Lisp supports *lexprs*, but they should not be used in new programs).

In general, a function in Lisp Machine Lisp has zero or more *required* parameters, followed by zero or more *optional* parameters, followed by zero or one *rest* parameter. This means that the caller must provide enough arguments so that each of the required parameters gets bound, but he may provide some extra arguments for each of the optional parameters. Also, if there is a rest parameter, he can provide as many extra arguments as he wants, and the rest parameter will be bound to a list of all these extras. Also, optional parameters may have a *default-form*, which is a form to be evaluated to produce the default argument if none is supplied.

Here is the exact explanation of how this all works. When **apply** matches up the arguments with the parameters, it follows the following algorithm:

The first required parameter is bound to the first argument. **apply** continues to bind successive required 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, then an error is caused ("too few arguments").

Next, after all required parameters are handled, **apply** continues with the optional parameters, binding each argument to each successive parameter. 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.

Finally, if there is no rest parameter and there are no remaining arguments, we are finished. If there is no rest parameter but there are still some arguments remaining, an error is caused ("too many arguments"). But if there *is* a rest parameter, it is bound to a list of all of the remaining arguments. (If there are no remaining arguments, it gets bound to **nil**.)

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

The keywords used here are **&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 required, optional, and rest.

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

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

or three arguments.

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

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

In all of the cases above, the *default-forms* for each parameter were **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. 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) b (c (symeval a)) &rest d)`
a's default-form is **'foo**, **b**'s is **nil**, and **c**'s is **(symeval a)**. Note that if the function whose lambda-list this is were called on no arguments, **a** would be bound to the symbol **foo**, and **c** would be bound to the binding 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**.

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 are like "prog variables".

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

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

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

It is important to realize that the list of arguments to which a rest-parameter is bound is not a "real" list. It is temporarily 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 **append**). The system will not detect the error of omitting to copy a rest-argument; you will simply find that you have a value which seems to change behind your back.

2. Predicates

A *predicate* is a function which 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.

By convention, the names of predicates usually end in the letter "p" (which stands for "predicate"). (See [section on naming conventions]).

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 *arg*
symbolp returns **t** if its argument is a symbol, otherwise **nil**.

nsymbolp *arg*
nsymbolp returns **nil** if its argument is a symbol, otherwise **t**.

listp *arg*
listp returns **t** if its argument is a cons, otherwise **nil**. **(listp nil)** is **nil** even though **nil** is the empty list.

nlistp *arg*
nlistp returns **t** if its argument is anything besides a cons, otherwise **nil**. This is the recommended predicate for terminating iterations or recursions on lists. It is, in fact, identical to **atom**.

atom *arg*
The predicate **atom** returns **t** if its argument is not a cons, otherwise **nil**.

fixp *arg*
fixp returns **t** if its argument is a fixnum or a bignum, otherwise **nil**.

floatp *arg*
floatp returns **t** if its argument is a flonum or a small flonum, otherwise **nil**.

small-floatp *arg*
small-floatp returns **t** if *arg* is a small flonum, otherwise **nil**.

bigp *arg*
bigp returns **t** if *arg* is a bignum, otherwise **nil**.

numberp *arg*
numberp returns **t** if its argument is any kind of number, otherwise **nil**.

stringp *arg*
stringp returns **t** if its argument is a string, otherwise **nil**.

arrayp *arg*
arrayp returns **t** if its argument is an array, otherwise **nil**. Note that strings are arrays.

subrp *arg*
subrp returns **t** if its argument is any compiled code object, otherwise **nil**. The Lisp Machine system doesn't use the term "subr", but the name of this function comes from Maclisp.

closurep *arg*
closurep returns **t** if its argument is a closure, otherwise **nil**.

locativep *arg*
locativep returns **t** if its argument is a locative, otherwise **nil**.

typep *arg*
typep is not really a predicate, but it is explained here because it is used to determine the datatype of an object. It returns a symbol describing the type of its argument, one of the following:

:symbol	A symbol.
:fixnum	A fixnum.
:flonum	A flonum.
:small-flonum	A small flonum.
:bignum	A bignum.
:list	A cons.
:string	A string.
:array	An array that is not a string.
:random	Any built-in data type that does not fit into one of the above categories.
foo	An object of user-defined data-type *foo* (any symbol). See Named Structures, page 91.

See also **data-type**, page 111.

The following functions are some other general purpose 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 '(a . b)) (eq x x) => t
```

Note that in Lisp Machine Lisp equal fixnums are **eq**; this is not true in Maclisp. Equality does not imply **eq**-ness for other types of numbers.

neq *Macro*

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

equal *x y*

The **equal** predicate returns **t** if its arguments are similar (isomorphic) objects. (cf. **eq**) Two numbers are **equal** if they have the same value (a flonum is never **equal** to a fixnum though). Two strings are **equal** if they have the same length, and the characters composing them are the same. Alphabetic case is ignored. For conses, **equal** is defined recursively as the two **car**'s being **equal** and the two **cdr**'s being equal. All other objects are **equal** if and only if they are **eq**. Thus **equal** could have been defined by:

```
(defun equal (x y)
       (or (eq x y)
           (and (numberp x) (numberp y) (= x y))
           (and (stringp x) (stringp y) (string-equal x y))
           (and (listp x)
                (listp y)
                (equal (car x) (car y))
                (equal (cdr x) (cdr y)))))
```

As a consequence of the above definition, it can be seen that **equal** need not terminate 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") => t
```

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. Even though Lisp uses the symbol **nil** to represent falseness, you shouldn't make understanding of your program depend on this fortuitously. For example, one often writes:

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

rather than

```
(cond (lst ... )
      ( ... ))
```

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

3. Evaluation

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

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

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

If *form* is a symbol, the result is the binding of *form*. If *form* is unbound, an error is signalled.

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

If *form* is a special form, indentified by a distinguished symbol as its car, it is handled accordingly; each special form works differently. All of them are documented in this manual.

If *form* is not a special form, it calls for the *application* of a function to *arguments*. The *car* of the form is a function or the name of a function. The *cdr* of the form is a list of forms which are evaluated to produce arguments, which are fed to the function. Whatever results the function *returns* is the value of the original *form*.

[Here there should be the rest of the moby description of evaluation and application, particularly multiple values. Explain the term "variables", also a very brief bit about locals and specials (fluids and lexicals??). The nature of functions should be revealed; including compiled-code, interpreted-code, arrays, stack-groups, closures, symbols. Discuss macros. Talk about function-calling in compiled code, how this is essentially identical to the **apply** function, and no need for (**sstatus uuolinks**) and the like.]

3.1 Functions and Special Forms

eval *x*

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

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

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

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

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

Note: unlike Maclisp, **eval** never takes a second argument; there are no "binding context pointers" in Lisp Machine Lisp. They are replaced by Closures (see page 102).

apply *fn arglist*

(**apply** *fn arglist*) applies the function *fn* to the list of arguments *arglist*. *arglist* should be a list; *fn* can be a compiled-code object, or a "lambda expression", i.e., a list whose **car** is the symbol **lambda**, or a symbol, in which case its definition (the contents of its function cell) is used.

Examples:

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

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

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

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

funcall *f* &rest *args*

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

Example:

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

lexpr-funcall *f* &rest *args*

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

Examples:

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

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

Note: the Maclisp functions **subrcall**, **lsubrcall**, and **arraycall** are not needed on the Lisp Machine; **funcall** is just as efficient.

quote *Special Form*

(**quote** *x*) simply returns *x*. It is useful because it takes the argument *quoted*, so that it is not evaluated by **eval**. **quote** is used to include constants in a form.

Examples:

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

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

For example,

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

is converted by read into

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

function *Special Form*

(**function** *x*) is similar to **quote**, except that it implies to the compiler that *x* is a function. In the interpreter, if *x* is a symbol (**function** *x*) returns *x*'s definition; otherwise *x* itself is returned. Because of this, using **function** rules out the possibility of later changing the function definition of *x*, including tracing it. Care is required!

comment *Special Form*

comment ignores its form and returns the symbol **comment**.

Example:

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

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

Example:

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

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

@define *Macro*

This macro turns into **nil**. It exists for the sake of the @ listing generation program, which uses it to declare names of special forms which define objects (such as functions) which @ should cross-reference.

progn *Special Form*

A progn-form looks like (**progn** *form1 form2 ...*). The *forms* are evaluated in order from left to right and the value of the last one is the result of the progn. **progn** is the primitive control structure construct for "compound statements". Although lambda-expressions, cond-forms, do-forms, and many other control structure forms use **progn** implicitly, that is, they allow multiple forms in their bodies, there are occasions when one needs to evaluate a number of forms for their side-effects and make them appear to be a single form.

Example:

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

prog1 *Special Form*

prog1 is similar to **progn**, but it returns the value of its *first* form. 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)))
```

which interchanges the values of the variables *x* and *y*.

prog1 could have been defined as:

```
(defun prog1 (&rest values)
    (car values))
```

It is actually implemented as a macro which expands into a **prog2**.

prog2 *Special Form*

prog2 is similar to **progn** and **prog1**, but it returns its *second* argument. It is included largely for Maclisp compatibility. It has two purposes: to evaluate two forms sequentially, which can be done more generally with **progn**, or to do what **prog1** is used for (c.f. **prog1** above).

let *Special Form*

let is used to bind some variables for some objects. A **let** form looks like

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

When this form is evaluated, first the *vform*s are evaluated. Then the *var*s are bound to the values returned by the corresponding *vform*s. Finally, the *bform*s are evaluated sequentially and the result of the last one returned.

let is implemented as a macro which expands into a lambda-combination; however, it is preferable to use **let** rather than **lambda** because the variables and the corresponding forms appear textually close to each other, which increases readability of the program.

See also **let-globally**, page 35.

progv *Special Form*

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

(progv *symbol-list value-list form1 form2 ...*)

first evaluates *symbol-list* and *value-list*. Then the symbols are bound to the values. In compiled code the symbols must be *special*, since the compiler has no way of knowing what symbols might appear in the *symbol-list*. If too few values are supplied, the remaining symbols are bound to **nil**. If too many values are supplied, the excess values are ignored.

After the symbols have been bound to the values, the *form*s are evaluated, and finally the symbols' bindings are undone. The result returned is the value of the last form. Note that the "body" of a **progv** is similar to that of **progn**, not that of **prog**. Example:

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

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

During the evaluation of the body of this **progv**, **foo** is bound to **bar**, **bar** is bound to **nil**, **b** is bound to **nil**, and **a** remains bound to **foo**.

See also **bind** (see page 118), which is a subprimitive which gives you maximal control over binding.

The following three functions (**arg**, **setarg**, and **listify**) exist only for compatibility with Maclisp *lexprs*.

arg *x*

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

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

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

setarg *i x*

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

listify *n*

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

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

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

3.2 Multiple Value Returns

multiple-value *Special Form*

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

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

intern returns a second value, which is **t** if the symbol returned as the first value was already on the obarray, or else **nil** if it just put it there. So if the symbol **goo** was already on the obarray, the variable **already-there-p** will be set to **t**, else it will be set to **nil**.

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-list *Special Form*

(**multiple-value-list** *form*) is another special form used on functions which may return multiple values. (**multiple-value-list** *form*) evaluates *form*, and returns a list of the values it returned.
Example:

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

This is similar to the example of **multiple-value** above; **a** will be set to a list of three elements, the three values returned by **intern**. The first is the newly interned symbol **goo**, the second is **nil** to indicate that it is newly-interned, and the third is the package on which it was interned.

multiple-value-call *Special Form*

(**multiple-value-call** (*function arg1 arg2* ...)) applies the function to the arguments, and returns from the current function with the same values as *function* returns. This only works in compiled programs.

multiple-value-return *Special Form*

(**multiple-value-return** (*function arg1 arg2* ...)) applies the function to the arguments, and returns from the current **prog** or **do** with the same values as *function* returns.

3.3 Evalhook

evalhook *Variable*

If the value of **evalhook** is non-**nil**, then special things happen in the evaluator. When a form (even an atom) is to be evaluated, **evalhook** is bound to **nil** and the function which was **evalhook**'s value is applied to one argument—the form that was trying to be evaluated. The value it returns is then returned from the evaluator. This feature is used by the **step** program (see page 255).

evalhook is bound to **nil** by **break** and by the error handler, and **setq**'ed to **nil** by errors that go back to top level and print *. 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 **evalhook**. It only applies to evaluation — whether in a read-eval-print loop, internally in evaluating arguments in forms, or by explicit use of the function **eval**. It ***does not*** have any effect on compiled function references, on use of the function **apply**, or on the "mapping" functions. (On the Lisp Machine, as opposed to Maclisp, it is not necessary to do (***rset t**) nor (**sstatus evalhook t**).)
(Also, Maclisp's special-case check for **store** is not implemented.)

evalhook *form hook*

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

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

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

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

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

(a . c)
```

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 a general iteration facility called **do**, which is explained below.

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 **and** and **or**, which are simple conditionals, and **cond**, which is a more general conditional.

Non-local exits are similar to the *leave*, *exit*, and *escape* constructs in many modern languages. They are similar to a *return*, but are more general. In Lisp, their scope is determined at run-time. They are implemented as the **catch** and ***throw** functions.

Lisp Machine Lisp also provides a multiple-process or coroutine capability. This is explained in the section on *stack-groups* (page 105).

4.1 Conditionals

and *Special Form*

(**and** *form1 form2 ...*) 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 non-**nil**, **and** returns the value of the last *form*. **and** can be used both for logical operations, where **nil** stands for *False* and **t** stands for *True*, and as a conditional expression.

Examples:

```
(and x y)

(and (setq temp (assq x y))
     (rplacd temp z))

(and (not error-p)
     (princ "There was no error."))
```

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

or *Special Form*

(**or** *form1 form2* ...) 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 non-**nil**, **or** immediately returns that value without evaluating any remaining *forms*. **or** can be used both for logical operations, where **nil** stands for False and **t** for True, and as a conditional expression.

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

cond *Special Form*

The **cond** special form consists of the symbol **cond** followed by several *clauses*. Each clause consists of a *predicate* followed by zero or more *forms*. Sometimes the predicate is called the *antecedent* and the forms are called the *consequents*.

```
(cond (antecedent consequent consequent...)
      (antecedent)
      (antecedent consequent ...)
      ... )
```

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

cond processes its clauses in order from left to right. First, the antecedent 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 forms, or consequents, which are evaluated in order from left to right. After evaluating the consequents, **cond** returns without inspecting any remaining clauses. The value of the **cond** special form is the value of the last consequent evaluated, or the value of the antecedent if there were no consequents in the clause. If **cond** runs out of clauses, that is, if every antecedent is **nil**, that is, if no case is selected, the value of the **cond** is **nil**.
Example:

```
(cond ((zerop x)      ;First clause:
       (+ y 3))       ; (zerop x) is the antecedent.
                      ; (+ y 3) is the consequent.
      ((null y)       ;A clause with 2 consequents:
       (setq y 4)     ; this
       (cons x z))    ; and this.
      (z)             ;A clause with no consequents:
                      ; the antecedent is just z.
      (t              ;An antecedent of t
       105)           ; is always satisfied.
 )                    ;This is the end of the cond.
```

if *Macro*

if allows a simple "if-then-else" conditional to be expressed as (**if** *pred-form then-form else-form*). **if** is provided for stylistic reasons; some people think it looks nicer than **cond** for the simple case it handles. (**if** *x y z*) expands into (**cond** (*x y*) (**t** *z*)).

selectq *Macro*

Many programs require **cond** forms which check various specific values of a form. A typical example:

```
(cond ((eq x 'foo) ...)
      ((eq x 'bar) ...)
      ((memq x '(baz quux mum)) ...)
      (t ...))
```

The **selectq** macro can be used for such tests. Its form is as follows:

```
(selectq key-form
         (pattern consequent consequent ...)
         (pattern consequent consequent ...)
         (pattern consequent consequent ...)
         ...)
```

Its first "argument" is a form, which is evaluated (only once) as the first thing **selectq** does. The resulting value is called the *key*. It is followed by any number of *clauses*. The **car** of each clause is compared with the key, and if it matches, the consequents of this clause are evaluated, and **selectq** returns the value of the last consequent. If there are no matches, **selectq** returns **nil**. Note that the patterns are *not* evaluated; if you want them to be evaluated use **select** rather than **selectq**.

A *pattern* may be any of:

1) A symbol	If the *key* is **eq** to the symbol, it matches.
2) A number	If the *key* is **eq** to the number, it matches. Only small numbers (*fixnums*) will work.
3) A list	If the *key* is **eq** to one of the elements of the list, then it matches. The elements of the list should be symbols or fixnums.
4) **t** or **otherwise**	The symbols **t** and **otherwise** are special keywords which match anything. Either symbol may be used, it makes no difference. **t** is accepted for compatibility with Maclisp's **caseq** construct.

Example:

```
(selectq x                  ;This is the same as the cond example
         (foo ...)          ; above.
         (bar ...)
         ((baz quux mum) ...)
         (otherwise ...))
```

select *Macro*

select is the same as **selectq**, except that the elements of the patterns are evaluated before they are used.

Example:

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

is equivalent to

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

dispatch *Macro*

(**dispatch** *byte-specifier n clauses...*) is the same as **select** (not **selectq**), but the key is obtained by evaluating (**ldb** *byte-specifier n*). *byte-specifier* and *n* are both evaluated.

Example:

```
(princ (dispatch 0202 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 will be dispatched on. [This function may get flushed.]

4.2 Iteration

prog *Special Form*

prog is a special form which provides temporary variables, sequential evaluation of forms, and a "goto" facility. A typical **prog** looks like:

```
(prog (var1 var2 (var3 init3) var4 (var5 init5))
 tag1
      statement1
      statement2
 tag2
      statement2
      . . .
)
```

var1, *var2*, ... are temporary variables. When the **prog** is entered, the values of these variables are saved. When the **prog** is finished, the saved values are restored. The initial value of a variable inside the **prog** depends on whether the variable had an associated *init* form or not; if it did, then the *init* form is evaluated and becomes

the initial value of the corresponding variable. If there was no *init* form, the variable is initialized to **nil**.
Example:

```
(prog ((a t)  b  (c 5)  (d (car '(zz . pp))))
      <body>
      )
```

The initial value of **a** is **t**, that of **b** is **nil**, that of **c** is the fixnum 5, and that of **d** is the symbol **zz**. The binding and initialization of the variables is done *sequentially*, so each one can depend on the previous ones.

The part of a **prog** after the variable list is called the *body*. An item in the body may be a symbol or a number, in which case it is called a *tag*, or some other form (i.e. a list), in which case it is called a *statement*.

After **prog** binds the temporary variables, it processes each form in its body sequentially. *tags* are skipped over. *Statements* are evaluated, and their returned values discarded. If the end of the body is reached, the **prog** returns **nil**. However, two special forms may be used in **prog** bodies to alter the flow of control. If (**return** *x*) is evaluated, **prog** stops processing its body, evaluates *x*, and returns the result. If (**go** *tag*) is evaluated, **prog** jumps to the part of the body labelled with the *tag*. *tag* is not evaluated. The "computed-**go**" (mis)feature of Maclisp is not supported.

The compiler requires that **go** and **return** forms be *lexically* within the scope of the **prog**; it is not possible for one function to **return** to a **prog** which is in progress in its caller. This restriction happens not to be enforced in the interpreter. Thus, a program which contains a **go** which is not contained within the body of a **prog** (or a **do**, see below) cannot be compiled. Since virtually all programs will be compiled at some time, the restriction should be adhered to.

Sometimes code which is lexically within more than one **prog** (or **do**) form wants to **return** from one of the outer **prog**s. However, the **return** function normally returns from the innermost **prog**. In order to make **return**ing from outer **prog**s more convenient, a **prog** may be given a *name* by which it may be referenced by a function called **return-from**, which is similar to **return** but allows a particular **prog** to be specified. If the first subform of a **prog** is a non-**nil** symbol (rather than a variable list), it is the name of the **prog**. See the description of the **return-from** special form, on page 31.
Example:

```
(prog george (a b d)
      (prog (c b)
            ...
            (return-from george (cons b d))
            ...))
```

If the symbol **t** is used as the name of a **prog**, then it will be made "invisible" to **returns**; **returns** inside that **prog** will return to the next outermost level whose name is not **t**. (**return-from t** ...) will return from a **prog** named **t**.

See also the **do** special form, which uses a body similar to **prog**. The **do**, ***catch**, and ***throw** special forms are included in Lisp Machine Lisp as an attempt to encourage goto-less programming style, which often leads to more readable, more easily maintained code. The programmer is recommended to use these functions instead of **prog** wherever reasonable.
Example:

```
(prog (x y z)  ;x, y, z are prog variables - temporaries.
   (setq y (car w) z (cdr w))      ;w is a free variable.
loop
   (cond ((null y) (return x))
         ((null z) (go err)))
rejoin
   (setq x (cons (cons (car y) (car z))
                 x))
   (setq y (cdr y)
         z (cdr z))
   (go loop)
err
   (break are-you-sure? t)
   (setq z y)
   (go rejoin))
```

do *Special Form*

The **do** special form provides a 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.

The newer variety of **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 *init*, which defaults to **nil** if it is omitted, and a repeat value *repeat*. If *repeat* is omitted, the *var* is not changed between loops.

All assignment to the index variables is done in parallel. At the beginning of the first iteration, all the *init*s are evaluated, then the *var*s are saved, then the *var*s are *set* to the values of the *init*s. To put it another way, the *var*s are **lambda**-bound to the values of the *init*s. Note that the *init*s are evaluated *before* the *var*s are bound, i.e. lexically *outside* of the **do**. At the beginning of each succeeding iteration those *var*s that have *repeat*s get **setq**'ed to the values of their respective *repeat*s. Note that all the *repeat*s are evaluated before any of the *var*s is changed.

The second element of the **do**-form is a list of an end-testing predicate *end-test*, and zero or more forms, called the *exit-form*s. At the beginning of each iteration, after processing of the *repeat*s, 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** (*not* the value of the *end-test* as you might expect) if there were no *exit-form*s. Note that the second element of the **do**-form resembles a **cond** clause.

If the second element of the form is **nil**, there is no *end-test* nor *exit-form*s, and the *body* of the **do** is executed only once. In this type of **do** it is an error to have *repeat*s. This type of **do** is a "**prog** with initial values."

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

The remainder of the **do**-form constitutes a **prog**-body; that is, **go**'s and **return** forms are understood within the **do** body, as if it were a **prog** body. When the end of the body is reached, the next iteration of the **do** begins. If a **return** form is evaluated, **do** returns the indicated value and no more iterations occur.

The older variety of **do** is:

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

The first time through the loop *var* gets the value of *init*; the remaining times through the loop it gets the value of *repeat*, which is re-evaluated each time. Note that *init* is evaluated before the value of *var* is saved, 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. The *body* is like a **prog** body. **go** may be used. If **return** is used, its argument is the value of the **do**. If the end of the **prog** body is reached, another loop begins.

Examples of the older variety of **do**:

```
(setq n (array-length foo-array))
(do i 0 (1+ i) (= i n)
  (aset 0 foo-array i))          ;zeroes out the array foo-array

(do zz x (cdr zz) (or (null zz)
                      (zerop (f (car zz)))))
                    ; this applies f to each element of x
              ; continuously until f returns zero.
              ; Note that the do has no body.
```

return forms are often useful to do simple searches:

```
(do i 0 (1+ i) (= i n)  ; Iterate over the length of foo-array.
  (and (= (aref foo-array i) 5) ; If we find an element which
                  ;equals 5,
       (return i)))      ; then return its index.
```

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))

(do ((z,list (cdr z)) ; z starts as list and is cdr'ed 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.
    (nil)             ; The end-test is nil, so this is an infinite loop.
  body)

(do ((x) (y) (z)) (nil) body)
```

is like

```
(prog (x y z) body)
```

except that when it runs off the end of the *body*,
do loops but **prog** returns **nil**.

On the other hand,

```
(do ((x) (y) (z)) nil body)
```

is identical to the **prog** above (it does not loop.)

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.

In either form of **do**, the *body* may contain 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.

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

is like `(maplist 'f x y)`.

do-named *Special Form*

do-named is just like **do** except that its first subform is a symbol, which is interpreted as the *name* of the **do**. The **return-from** special form allows a **return** from a particular **prog** or **do-named** when several are nested. See the description of such names in the explanation of the **prog** special form on page 25, and that of **return-from** on page 31.

go *Special Form*

The (**go** *tag*) special form is used to do a "go-to" within the body of a **do** or a **prog**. The *tag* must be a symbol. It is not evaluated. **go** transfers control to the point in the body labelled by a tag **eq** to the one given. If there is no such tag in the body, the bodies of lexically containing **prog**s and **do**s (if any) are examined as well. If no tag is found, an error is signalled.

Note that the **go** form is a very special form: it does not ever return a value. A **go** form may not appear as an argument to a regular function, but only at the top level of a **prog** or **do**, or within certain special forms such as conditionals which are within a **prog** or **do**. A **go** as an argument to a regular function would be not only useless but possibly meaningless. The compiler does not bother to know how to compile it correctly. **return** and ***throw** are similar.

Example:

```
(prog (x y z)
  (setq x some frob)
loop
  do something
  (and some predicate (go endtag))
  do something more
  (and (minusp x) (go loop))
endtag
  (return z))
```

return *arg*

return is used to return from a **prog** or a **do**. The value of **return**'s argument is returned by **prog** or **do** as its value. In addition, **break** recognizes the typed-in S-expression (**return** *value*) specially. If this form is typed at a **break**, *value* will be evaluated and returned as the value of **break**. If not at the top level of a form typed at a **break**, and not inside a **prog** or **do**, **return** will cause an error.

Example:

```
(do ((x x (cdr x))
     (n 0 (* n 2)))
    ((null x) n)
 (cond ((atom (car x))
        (setq n (1+ n)))
       ((memq (caar x) '(sys boom bleah))
        (return n))))
```

return is, like **go**, a special form which does not return a value.

return can also be used to return multiple values from a **prog** or **do**, by giving it multiple arguments. For example,

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

This function is like **assq**, but it returns an additional value which is the index in the table of the entry it found. See the special forms **multiple-value** (page 19) and **multiple-value-list** (page 19).

return-from *Special Form*

A **return-from** form looks like (**return-from** *name form1 form2 form3*). The *forms* are evaluated sequentially, and then are returned from the innermost containing **prog** or **do-named** whose name is *name*. See the description of **prog** (page 25) in which named **prog**s and **do**s are explained, and that of **do-named** (page 30).

return-list *list*

list must not be **nil**. This function is like **return** except that the **prog** returns all of the elements of *list*; if *list* has more then one element, the **prog** does a multiple-value return.

To direct the returned values to a **prog** or **do-named** of a specific name, use (**return-from** *name* (**return-list** *list*)).

defunp *Macro*

Usually when a function uses **prog**, the **prog** form is the entire body of the function; the definition of such a function looks like (**defun** *name arglist* (**prog** *varlist* ...)). For convenience, the **defunp** macro can be used to produce such definitions. A **defunp** form expands as follows:

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

expands into

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

4.3 Non-local Exits

***catch** *tag form*

catch** is the Lisp Machine Lisp function for doing structured non-local exits. (catch** *tag form*) evaluates *form* and returns its value, except that if, during the evaluation of *form*, (***throw** *tag y*) should be evaluated, ***catch** immediately returns *y* without further evaluating *x*. Note that the *form* argument is ***not*** evaluated twice; the special action of ***catch** happens during the evaluation of its arguments, not during the execution of ***catch** itself.

The *tag*'s are used to match up ***throw**'s with ***catch**'s. (***catch 'foo** *form*) will catch 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).

- The values **t** and **nil** for *tag* are special and mean that all throws are to be caught. These are used 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.

***catch** returns up to four values; trailing null values are not returned for reasons of microcode simplicity, however the values not returned will default to **nil** if they are received with the **multiple-value** special form. If the catch completes normally, the first value is the value of *form* and the second is **nil**. If a ***throw** occurs, the first value is the second argument to ***throw**, and the second value is the first argument to ***throw**, the tag thrown to. The third and fourth values are the third and fourth arguments to ***unwind-stack** if that was used in place of ***throw**, otherwise **nil**. To summarize, the four values returned by ***catch** are the value, the tag, the active-frame-count, and the action.

Example

```
(*catch 'negative
        (mapcar (function (lambda (x)
                          (cond ((minusp x)
                                 (*throw 'negative x))
                                (t (f x)) )))
                y)
        )
```

which returns a list of f of each element of y if they are all positive, otherwise the first negative member of y.

Note: The Lisp Machine Lisp functions ***catch** and ***throw** are improved versions of the Maclisp functions **catch** and **throw**. The Maclisp ones are similar in purpose, but take their arguments in reversed order, do not evaluate the *tag*, and may be used in an older form in which no *tag* is supplied. Compatibility macros are supplied so that programs using the Maclisp functions will work.

***throw** *tag value*

***throw** is used with ***catch** as a structured non-local exit mechanism.

(***throw** *tag x*) throws the value of *x* back to the most recent ***catch** labelled with *tag* or **t** or **nil**. Other ***catches** are skipped over. Both *x* and *tag* are evaluated, unlike the Maclisp **throw** function.

The values **t** and **nil** for *tag* are reserved. **nil** may not be used, because it would cause an ambiguity in the returned values of ***catch**. **t** invokes a special feature whereby the entire stack is unwound, and then a coroutine transfer to the invoking stack-group is done. 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 completely unwind a stack. It leaves the stack-group in a somewhat inconsistent state; it is best to do a **stack-group-preset** immediately afterwards.

See the description of ***catch** for further details.

catch *Macro*
throw *Macro*

catch and **throw** are provided only for Maclisp compatibility. They expand as follows:

```
(catch form tag) ==> (*catch (quote tag) form)
(throw form tag) ==> (*throw (quote tag) form)
```

The forms of **catch** and **throw** without tags are not supported.

***unwind-stack** *tag value active-frame-count action*

This is a generalization of ***throw** provided for program-manipulating programs such as the error handler.

tag and *value* are the same as the corresponding arguments to ***throw**.

active-frame-count, if non-**nil**, is the number of frames to be unwound. 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, a stack-group call is forced to the previous stack-group, generally the error handler. The unwound stack-group is left in *awaiting-return* state, such that the value returned when the stack-group is resumed will become the value returned by the frame, (i.e. the *value* argument to ***unwind-stack** will be ignored in this case, and the value passed to the stack group when it is resumed will be used instead.)

Note that if both *active-frame-count* and *action* are **nil**, ***unwind-stack** is identical to ***throw**.

unwind-protect *Macro*

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 facility of Lisp creates a situation in which the above code won't work, however: if **hairy-function** should do a ***throw** to a ***catch** which is outside of the **progn** form, then **(turn-off-water-faucet)** will never be evaluated (and the faucet will presumably be left running).

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** does a ***throw** which attempts to quit out of the evaluation of the **unwind-protect**, the **(turn-off-water-faucet)** form will be evaluated in between the time of the ***throw** and the time at which the ***catch** returns. If the **progn** returns normally, then the **(turn-off-water-faucet)** is evaluated, and the **unwind-protect** returns the result of the **progn**. One thing to note is that **unwind-protect** cannot return multiple values.

The general form of **unwind-protect** looks like

```
(unwind-protect protected-form
                form1
                form2
                ...)
```

protected-form is evaluated, and when it returns or when it attempts to quit out of the **unwind-protect**, the *form*s are evaluated.

let-globally *Macro*

let-globally is similar in form to **let** (see page 17). The difference is that **let-globally** does not *bind* the variables; instead, it *sets* them, and sets up an **unwind-protect** (see page 34) to set them back. The important difference between **let-globally** and **let** is that when the current stack group (see page 105) cocalls some other stack group, the old values of the variables are *not* restored.

catch-all *Macro*

(catch-all *form*) is like (***catch** *some-tag form*) except that it will catch a ***throw** to any tag at all. Since the tag thrown to is the second returned value, the caller of **catch-all** may continue throwing to that tag if he wants. The one thing that **catch-all** will not catch is a ***throw** to **t**. **catch-all** is a macro which expands into ***catch** with a *tag* of **nil**.

4.4 Mapping

map *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** will proceed down the lists *x1*, *x2*, ..., *xn* in parallel. The first argument to *f* will come from *x1*, the second from *x2*, etc. The iteration stops as soon as any of the lists is exhausted.

There are five other mapping functions besides **mapcar**. **maplist** is like **mapcar** except that the function is applied to the list and successive cdr's of that list rather than to successive elements of the list. **map** 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,

```
(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* will be 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 acceptable to **apply** - it cannot be a macro or the name of a special form. Of course, there is nothing wrong with using functions which have optional and rest parameters.

Here is a table showing the relations between the six map functions.

		applies function to	
		successive sublists	successive elements
returns	its own second argument	map	mapc
	list of the function results	maplist	mapcar
	nconc of the function results	mapcon	mapcan

There are also functions (**mapatoms** and **mapatoms-all**) for mapping over all symbols in certain packages. See the explanation of packages (page 176).

5. Manipulating List Structure

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, a degree of control is provided over the action taken when there is an attempt to apply one of them to a symbol or a number. There are four mode-switches known as the *car-number mode*, *cdr-number mode*, *car-symbol mode*, and *cdr-symbol mode*. Here are the meanings of the values of these mode switches:

car-number = 0	*car* of a number is an error. This is the default.
car-number = 1	*car* of a number is **nil**.
cdr-number = 0	*cdr* of a number is an error. This is the default.
cdr-number = 1	*cdr* of a number is **nil**.
car-symbol = 0	*car* of a symbol is an error.
car-symbol = 1	*car* of **nil** is **nil**, but the *car* of any other symbol is an error. This is the default.
car-symbol = 2	*car* of any symbol is **nil**.
car-symbol = 3	*car* of a symbol is its print-name.
cdr-symbol = 0	*cdr* of a symbol is an error.
cdr-symbol = 1	*cdr* of **nil** is **nil**, but the *cdr* of any other symbol is an error. This is the default.
cdr-symbol = 2	*cdr* of any symbol is **nil**.
cdr-symbol = 3	*cdr* of **nil** is **nil**, *cdr* of any other symbol is its property list.

The values of the mode switches can be altered with the function **set-error-mode** (see page 264). They are stored as byte fields in the special variable **%m-flags**. The reason that the two *symbol* modes default in that fashion is to allow programs to **car** and **cdr** off the ends of lists without having to check, which is sometimes useful. A few system functions depend on **car** and **cdr** of **nil** being **nil**, although they hadn't ought to, so things may break if you change these modes.

The value of 3 for the *symbol* modes exists for compatibility with ancient versions of Maclisp, and should not be used for any other reasons. (The appropriate functions are **get-pname** (see page 65) and **plist** (see page 64).) Note: unlike Maclisp, the values of the symbols **car** and **cdr** are not used; the various mode switches above serve their purpose. Also unlike Maclisp, this error checking is always done, even in compiled code, regardless of the value of ***rset**.

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)
```

There are two reasons this exists: one is that you might want the arguments to **cons** evaluated in the other order, and the other is that the compiler might convert calls to **cons** into calls to **xcons** for efficiency. In fact, it doesn't.

cons-in-area *x y area-number*

This function creates a *cons* in a specific *area*. (*Areas* are an advanced feature of storage management; 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*)

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 page 135.

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.

5.2 Lists

The following section explains some of the basic functions provided for dealing with *lists*. There has been some confusion about the term *list* ever since the beginnings of the language: for the purposes of the following descriptions, a list is the symbol **nil**, or a cons whose cdr is a list. Note well that although we consider **nil** to be a *list* (the list of zero elements), it is a symbol and not a cons, and the **listp** predicate is not true of it (but perhaps **listp** will be changed in the future).

last *list*
last returns the last cons of *list*. If *list* is **nil**, it returns **nil**.
Example:

```
(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))) ))
```

length *list*
length returns the length of *list*. The length of a list is the number of top-level conses in it.
Examples:

```
(length nil) => 0
(length '(a b c d)) => 4
(length '(a (b c) d)) => 3
```

length could have been defined by:

```
(defun length (x)
    (cond ((atom x) 0)
          ((1+ (length (cdr x)))) ))
```

or by:

```
(defun length (x)
    (do ((n 0 (1+ n))
         (y x (cdr y)))
        ((atom y) n) ))
```

first *Macro*
second *Macro*
third *Macro*
fourth *Macro*
fifth *Macro*
sixth *Macro*
seventh *Macro*

```
(first x) ==> (car x)
(second x) ==> (cadr x)
(third x) ==> (caddr x)
(fourth x) ==> (cadddr x)
etc.
```

rest1 *Macro*
rest2 *Macro*
rest3 *Macro*
rest4 *Macro*

```
(rest1 x) ==> (cdr x)
(rest2 x) ==> (cddr x)
(rest3 x) ==> (cdddr x)
(rest4 x) ==> (cddddr x)
```

nth *n list*

(**nth** *n list*) returns the *n*'th element of *list*, where the zeroth element is the *car* of the list.

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**, whose precise equivalent is the 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*) *cdr*s *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. This is the same as InterLisp's function **nth**. **nthcdr** is defined by:

```
(defun nthcdr (n list)
    (do ((i 0 (1+ i))
         (list list (cdr list)))
        ((= i n) list)))
```

list &rest *args*

list 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 default-cons-area (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 two arguments.

Example:

```
(list* 'a 'b 'c 'd) => (a b c . d)
```

This is like

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

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 *area size*

This creates and returns a list containing *size* elements, each of which is **nil**. *size* should be a fixnum. The list is allocated in the area specified; if you are not using areas in any special way, just use the value of the symbol **default-cons-area**.

Example:

```
(make-list default-cons-area 3) => (nil nil nil)
```

Of course, this function is not usually used when the value of the second argument is a constant; if you want a list of three **nil**s, it is easy enough to type **(nil nil nil)**. **make-list** is used when the number of elements is to be computed while the program is being run.

make-list and **cons** are the two primitive list-creation functions which all the other functions call. The difference is that **make-list** creates a *cdr-coded* list (see page 48).

circular-list &rest *args*

circular-list 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.

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)
```

To make a copy of the top level of a list, that is, to copy the list but not its elements, use (**append** *x* **nil**).

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:

```
(defun append (&rest args)
    (and 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 cdr-coding the list it produces, using *cdr-next* except at the end where a full node is used to link to the last argument, unless the last argument was **nil** in which case *cdr-nil* is used.

reverse *list*

reverse creates a new list whose elements are the elements of *list* taken in reverse order. **reverse** does not modify its argument, unlike **nreverse** which is faster but does modify its argument.

Example:

```
(reverse '(a b (c d) e)) => (e (c d) b a)
```

reverse could have been defined by:

```
(defun reverse (x)
    (do ((l x (cdr l))          ; scan down argument,
         (r nil                 ; putting each element
            (cons (car l) r)))  ; into list, until
        ((null l) r)))          ; no more elements.
```

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 43)

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 is 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.
```

nreverse *list*

nreverse reverses its argument, which should be a list. The argument is destroyed by **rplacd**'s all through the list (cf. **reverse**).

Example:

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

nreverse could have been defined by:

```
(defun nreverse  (x)
    (cond ((null x) nil)
          ((nreversel x nil))))

(defun nreversel (x y)              ;auxiliary function
    (cond ((null (cdr x)) (rplacd x y))
          ((nreversel (cdr x) (rplacd x y)))))
          ;; this last call depends on order of argument evaluation.
```

Currently, **nreverse** does something inefficient with *cdr-coded* lists, however this will be changed. In the meantime **reverse** might be preferable in some cases.

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)
          ((nreversel x y)) ))
```

using the same **nreversel** as above.

push *Macro*

The form is (**push** *item place*), where *item* is an arbitrary object and *place* is a reference to a cell containing a list. Usually *place* is the name of a variable. *item* is consed onto the front of the list.

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. In general, (**push** *item place*) expands into (**setf** *place* (**cons** *item place*)). (See page 146 for an explanation of **setf**.)

pop *Macro*

The form is (**pop** *place*). The result is the **car** of the contents of *place*, and as a side-effect the **cdr** of the contents is stored back into *place*.

Example:

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

In general, (**pop** *place*) expands into (**progl** (**car** *place*) (**setf** *place* (**cdr** *place*))). (See page 146 for an explanation of **setf**.)

butlast *list*

This creates and returns a list with the same elements as *list*, excepting the last element.

Examples:

```
(butlast '(a b c d)) => (a b c)
(butlast '((a b) (c d)) => ((a b))
(butlast '(a)) => nil
(butlast nil) => nil
```

The name is from the phrase "all elements but the last".

nbutlast *list*

This is the destructive version of **butlast**; it changes the cdr of the second-to-last cons of the list to nil. If there is no second-to-last cons (that is, if the list has fewer than two elements) it returns **nil**.

Examples:

```
(setq foo '(a b c d))
(nbutlast foo) => (a b c)
foo => (a b c)
(nbutlast '(a)) => nil
```

firstn *n list*

firstn 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 will be **nil**.

Example:

```
(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)
```

ldiff *list sublist*

list should be a list, and *sublist* should be a sublist of *list*, i.e. one of the conses that make up *list*. **ldiff** (meaning List Difference) will return a new list, whose elements are those elements of *list* that appear before *sublist*.

Examples:

```
(setq x '(a b c d e))
(setq y (cdddr x)) => (d e)
(ldiff x y) => (a b c)
```

but

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

since the sublist was not **eq** to any part of the list.

5.3 Alteration of List Structure

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 not copied but is physically altered; hence caution should be exercised when using these functions, as strange side-effects can occur if portions of list structure become shared unbeknownst to the programmer. The **nconc**, **nreverse**, **nreconc**, and **nbutlast** functions already described, and the **delq** family described later, have the same property. However, they are normally not used for this side-effect; rather, the list-structure modification is purely for efficiency and compatible non-modifying functions are provided.

rplaca *x y*

(**rplaca** *x y*) changes the car of *x* to *y* and returns (the modified) *x*. *x* should be a cons, but *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*

(**rplacd** *x y*) changes the cdr of *x* to *y* and returns (the modified) *x*. *x* should be a cons, but *y* may be any Lisp object.

Example:

```
(setq x '(a b c))
(rplacd x 'd) => (a . d)
Now x => (a . d)
```

Note to Maclisp users: **rplacd** should not be used to set the property list of a symbol, although there is a compatibility mode in which it will work. See **car** (page 38). The right way to set a property list is with **setplist** (see page 64).

subst *x y z*

(**subst** *x y z*) substitutes *x* for all occurrences of *y* in *z*, and returns the modified copy of *z*. The original *z* is unchanged, as **subst** recursively copies all of *z* replacing elements **eq** to *y* as it goes. If *x* and *y* are **nil**, *z* is just copied, which is a convenient way to copy arbitrary list structure.

Example:

```
(subst 'Tempest 'Hurricane
       '(Shakespeare wrote (The Hurricane)))
    => (Shakespeare wrote (The Tempest))
```

subst could have been defined by:

```
(defun subst (x y z)
    (cond ((eq z y) x)      ;if item eq to y, replace.
          ((atom z) z)      ;if no substructure, return arg.
          ((cons (subst x y (car z))  ;otherwise recurse.
                 (subst x y (cdr z))))))
```

Note that this function is not "destructive"; that is, it does not change the *car* or *cdr* of any already-existing list structure.

sublis *alist S-expression*

sublis makes substitutions for symbols in an S-expression (a structure of nested lists). The first argument to **sublis** is an association list (see the next section). The second argument is the S-expression in which substitutions are to be made. **sublis** looks at all symbols in the S-expression; if a symbol appears in the association list occurrences of it are replaced by the object it is associated with. The argument is not modified; new conses are created where necessary and only where necessary, so the newly created structure shares as much of its substructure as possible with the old. For example, if no substitutions are made, the result is **eq** to the old S-expression.

Example:

```
(sublis '((x . 100) (z . zprime))
        '(plus x (minus g z x p) 4))
=> (plus 100 (minus g zprime 100 p) 4)
```

5.4 Cdr-Coding

There is an issue which those who must be concerned with efficiency will need to think about. In the Lisp Machine there are actually two kinds of lists; normal lists and *cdr-coded* lists. Normal lists take two words for each *cons*, while cdr-coded lists require only one word for each *cons*. The saving is achieved by taking advantage of the usual structure of lists to avoid storing the redundant cdrs which link together the conses which make up the list. Ordinarily, **rplacd**'ing such a list would be impossible, since there is no explicit representation of the cdr to be modified. However, in the Lisp machine system it is merely somewhat expensive; a 2-word ordinary cons must be allocated and linked into the list by an invisible pointer. This is slower than an ordinary **rplacd**, uses extra space, and slows down future accessing of the list.

One should try to use normal lists for those data structures that will be subject to **rplacd**ing operations, including **nconc** and **nreverse**, and cdr-coded lists for other structures. The functions **cons**, **xcons**, **ncons**, and their area variants make normal lists. The functions **list**, **list***, **list-in-area**, **make-list**, and **append** make cdr-coded lists. The other list-creating functions, including **read**, currently make normal lists, but this should not be relied upon. Some functions, such as **sort**, take special care to operate efficiently on cdr-coded lists (**sort** treats them as arrays). **nreverse** is rather slow on cdr-coded lists, currently, since it simple-mindedly uses **rplacd**, however this will be changed.

It is currently *not* planned that the garbage collector compact ordinary lists into cdr-coded lists. (**append** *x* **nil**) is a suitable way to copy a list, converting it into cdr-coded form.

5.5 Tables

Lisp Machine Lisp includes several functions which simplify the maintenance of tabular data structures of several varieties. The simplest is a plain list of items, which models (approximately) the concept of a *set*. 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. Set union, intersection, and difference functions are easily written using these.

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.

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 page 144).

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. Lisp Machine lisp includes a hashing function (**sxhash**) which aids in the construction of more efficient, hairier structures.

memq *item list*

(**memq** *item list*) returns **nil** if *item* is not one of the elements of *list*. Otherwise, it returns the portion of *list* beginning with the first occurrence of *item*. The comparison is made by **eq**. *list* is searched on the top level only. 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 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:

```
(*catch 'lose
        (rplaca (or (memq x z)
                    (*throw 'lose nil))
                y)
        )
```

memq could have been defined by:

```
(defun memq (item list)
    (cond ((atom list) nil)
          ((eq item (car list)) list)
          ((memq item (cdr list))) ))
```

memq is hand-coded in microcode and therefore especially fast.

member *item list*

member is like **memq**, except **equal** is used for the comparison, instead of **eq**.

member could have been defined by:

```
(defun member (item list)
    (cond ((null list) nil)
          ((equal item (car list)) list)
          ((member item (cdr list))) ))
```

mem *predicate item list*

mem is the same as **memq** except that it takes an extra argument which should be a predicate of two arguments, which is used for the comparison instead of **eq**. **(mem 'eq a b)** is the same as **(memq a b)**. **(mem 'equal a b)** is the same as **(member a b)**.

mem is usually used with equality predicates other than **eq** and **equal**, such as **=**, **char-equal** or **string-equal**.

delq *item list* &optional *n*

(delq *item list*) returns the *list* with all top-level 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)
```

The latter is *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 the number of occurrences of *item* in the list, all occurrences of *item* in the list will be 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 7777777))  ;7777777 as infinity.
    (cond ((or (atom list) (zerop n)) list)
          ((eq item (car list))
           (delq item (cdr list) (1- n)))
          ((rplacd list (delq item (cdr list) n)))))
```

delete *item list* &optional *n*

delete is the same as **delq** except that **equal** is used for the comparison instead of **eq**.

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)**. (c.f. **mem**, page 50)

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**.

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**). (c.f. **mem** page 50)

rem-if *predicate list*
predicate should be a function of one argument. **rem-if** makes a new list by applying *predicate* to all of the elements of *list* and removing the ones for which the predicate returns non-**nil**. The function's name means "remove if this condition is true".

rem-if-not *predicate list*
predicate should be a function of one argument. **rem-if-not** makes a new list by applying *predicate* to all of the elements of *list* and removing the ones for which the predicate returns **nil**. The function's name means "remove if this condition is not true"; i.e. it keeps the elements for which *predicate* is true.

del-if *predicate list*
del-if is just like **rem-if** except that it modifies *list* rather than creating a new list. See **rem-if**.

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 **rem-if-not**.

every *list predicate* &optional *step-function*
every 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.

some *list predicate* &optional *step-function*
some returns **t** if *predicate* returns non-**nil** when applied to some element of *list*, 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.

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**.

sxhash *S-expression*

sxhash computes a hash code of an S-expression, and returns it as a fixnum, which may be positive or negative. A property of **sxhash** is that (**equal** *x y*) implies (= (**sxhash** *x*) (**sxhash** *y*)). The number returned by **sxhash** is some possibly large number in the range allowed by fixnums. It is guaranteed that:

1) **sxhash** for a symbol will always be positive.
2) **sxhash** of any particular expression will be constant in a particular implementation for all time, probably.
3) **sxhash** of any object of type **random** will be zero.
4) **sxhash** of a fixnum will = that fixnum.

Here is an example of how to use **sxhash** in maintaining hash tables of S-expressions:

```
(defun knownp (x)      ;look up x in the table
  (prog (i bkt)
    (setq i (plus 76 (remainder (sxhash x) 77)))
      ;The remainder should be reasonably randomized between
      ;-76 and 76, thus table size must be > 175 octal.
    (setq bkt (aref table i))
      ;bkt is thus a list of all those expressions that hash
      ;into the same number as does x.
    (return (memq x bkt))))
```

To write an "intern" for S-expressions, one could

```
(defun sintern (x)
  (prog (bkt i tem)
    (setq bkt (aref table
                    (setq i (+ 2n-2 (\ (sxhash x) 2n-1)))))
        ;2n-1 and 2n-2 stand for a power of 2 minus one and
        ;minus two respectively.  This is a good choice to
        ;randomize the result of the remainder operation.
    (return (cond ((setq tem (memq x bkt))
                   (car tem))
                  (t (aset (cons x bkt) table i)
                     x)))))
```

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) now
```

A typical trick is to say (**cdr** (**assq** x y)). Assuming 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))) ))
```

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**). (c.f. **mem** page 50)

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*).

rassoc *item alist*

rassoc means *reverse assoc*. It is like **assoc**, but it tries to find an element of *alist* whose *cdr* (not car) is *equal* to *item*. **rassoc** is defined by:

```
(defun rassoc (item in-list)
    (do l in-list (cdr l) (null l)
      (and (equal 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)))
```

pairlis *cars cdrs*

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))
```

find-position-in-list *item list*

find-position-in-list looks down *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.

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
```

find-position-in-list-equal *item list*

find-position-in-list-equal is exactly the same as **find-position-in-list**, except that the comparison is done with **equal** instead of **eq**.

5.6 Sorting

Several functions are provided for sorting arrays and lists. These functions use algorithms which always terminate no matter what sorting predicate is used, provided only that the predicate always terminates. The array sort is not necessarily *stable*; that is, equal items may not stay in their original order. However the list sort *is* stable.

After sorting, the argument (be it list or array) is rearranged internally so as to be completely ordered. In the case of an array argument, this is accomplished by permuting the elements of the array, 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 **append**, as appropriate.

Should the comparison predicate cause an error, such as a wrong type argument error, the state of the list or array being sorted is undefined. However, if the error is corrected the sort will, of course, proceed correctly.

The sorting package is smart about cdr-coded lists.

sort *table predicate*

The first argument to **sort** is an array or a list. The second is a predicate, which must be applicable to all the objects in the array or list. 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 sort the contents of the array or list under the ordering imposed by the predicate, and returns the array or list modified into sorted order, i.e. its modified first argument. Note that since sorting requires many comparisons, and thus many calls to the predicate, sorting will be much faster if the predicate is a compiled function rather than interpreted.
Example:

```
(defun mostcar (x)
    (cond ((symbolp x) x)
          ((mostcar (car x)))))

(sort 'fooarray
      (function (lambda (x y)
          (alphalessp (mostcar x) (mostcar y)))))
```

If **fooarray** contained these items before the sort:

```
(Tokens (The lion sleeps tonight))
(Carpenters (Close to you))
((Rolling Stones) (Brown sugar))
((Beach Boys) (I get around))
(Beatles (I want to hold your hand))
```

then after the sort **fooarray** would contain:

```
((Beach Boys) (I get around))
(Beatles (I want to hold your hand))
(Carpenters (Close to you))
((Rolling Stones) (Brown sugar))
(Tokens (The lion sleeps tonight))
```

sortcar *x predicate*

sortcar is exactly like **sort**, but the items in the array or list being sorted should all be conses. **sortcar** takes the **car** of each item before handing two items to the predicate. Thus **sortcar** is to **sort** as **mapcar** is to **maplist**.

The spelling of the names of the next two functions will be corrected at some point.

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 a 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.

6. Symbols

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 *binding* or *value*, since it is what you get when you evaluate the symbol. The binding of symbols to values allows symbols to be used as the implementation of *variables* in programs.

The value cell can also be *empty*, referring to *no* Lisp object, in which case the symbol is said to be *unbound*. This is the initial state of a symbol when it is created. An attempt to evaluate an unbound symbol causes an error.

The binding of a symbol can be changed either by *lambda-binding* or by *assignment*. The difference is that when a symbol is lambda-bound, its previous value is saved away, to be restored later, whereas assignment discards the previous value.

The symbols **nil** and **t** are always bound to themselves; they may not be assigned nor lambda-bound. (The error of changing the value of **t** or **nil** is not yet detected, but it will be.)

When *closures* are in use, the situation is a little more complicated. See the section on closures.

When a Lisp function is *compiled*, most of its variables are compiled into *local variables*, which are not represented by means of symbols. However the compiler recognizes usage of the **setq** special form, and of the **set** and **value-cell-location** functions with a quoted argument, as referring to variables rather than symbols, and generates the appropriate code to access the corresponding local variable rather than the symbol.

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)
```

will either set **c** to **foo** or set **d** to **foo**.

setq *Special Form*

The special form (**setq** *var1 form1 var2 form2...*) is the "variable assignment statement" of Lisp. First *form1* is evaluated and the result is assigned to *var1*, using **set**, then *form2* is evaluated and the result is assigned to *var2*, and so forth. **setq** returns the last value assigned, i.e. the result of the evaluation of its last argument.

Example:

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

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

psetq *Macro*

A **psetq** form is just like a **setq** form, except that the assignments happen in parallel; first all of the forms are evaluated, and then the symbols are set to the resulting values.

Example:

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

symeval *sym*

symeval is the basic primitive for retrieving a symbols's value. **(symeval** *sym***)** returns *sym*'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** causes an error.

boundp *sym*

boundp returns **t** if *sym* is bound; otherwise, it returns **nil.**

makunbound *sym*

makunbound causes *sym* to become unbound.

Example:

```
(setq a 1)
a => 1
(makunbound 'a)
a => causes an error.
```

makunbound returns its argument.

value-cell-location *sym*

value-cell-location returns a locative pointer to *sym*'s value cell. See the section on locatives.

[Must explain about external vs internal value cell.]

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 *applied* 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 *FEF* containing a compiled program, 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 empty, and it can be lambda-bound or assigned. The following functions are analogous to the value-cell related functions in the previous section.

fsymeval *sym*
: **fsymeval** returns *sym*'s definition, the contents of its function cell. If the function cell is empty, **fsymeval** causes an error.

fset *sym x*
: **fset** stores *x*, which may be any Lisp object, into *sym*'s function cell. It returns *x*.

fboundp *sym*
: **fboundp** returns **nil** if *sym*'s function cell is empty, i.e. *sym* is undefined. Otherwise it returns **t**.

fmakunbound *sym*
: **fmakunbound** causes *sym*'s to be undefined, i.e. its function cell to be empty. It returns *sym*.

function-cell-location *sym*
: **function-cell-location** returns a locative pointer to *sym*'s function cell. See the section on locatives.

The usual means of putting a function in a symbol's function cell (*defining* the symbol) is by means of the **defun** special form. Macros are put in a symbol's function cell by means of the **macro** special form.

defun *Special Form*
: **defun** is used for defining functions. A **defun** form looks like:

```
(defun name type lambda-list
  body)
```

The *type* is only for Maclisp compatibility, and is optional and usually absent. The *lambda-list* is as described on page 6 and may contain "&-keywords".

Examples:

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

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

A list (**lambda** *lambda-list* . *body*) is left in the function cell of *name*.

For compatibility, the Maclisp *types* **expr**, **fexpr**, and **macro**, and Maclisp *lexprs* (which have an atomic lambda-list) are recognized and the corresponding Lisp Machine flavor of **defun** is assumed.

macro *Special Form*

macro is used for defining macros. Its form is:

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

Examples:

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

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

In the function cell of *name* is placed a cons whose car is the symbol **macro**, and whose cdr is a **lambda**-expression of the form (**lambda** (*arg*) . *body*).

Much of the time it is more convenient and clear to use a macro-defining macro such as **defmacro** (see page 137) to define macros.

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

This is the same as (**fset** *symbol definition*) except that it makes some checks and saves the old definition. **defun**, **macro**, **undefun**, **load**, and the compiler call **fset-carefully** when they define functions.

fset-carefully prints a message and asks the user if the current package (value of **package**) is not allowed to redefine the *symbol*. Specifying *force-flag* non-**nil** suppresses this check.

The previous definition, if any, of *symbol* is saved on the **:previous-definition** property. If it is a list, it is also saved on the **:previous-expr-definition** property. These properties are used by the **undefun** function (page 61), which restores the previous definition, and the **uncompile** function (page 126), which restores the previous interpreted definition.

If *symbol* is not a symbol, but a list (*name prop*), then the definition is put on *name*'s *prop* property, the package error check is not done, and the old definition is not saved. This is used to implement the **(defun (*name prop*) ...)** feature.

undefun *symbol*
If *symbol* has a **:previous-definition** property, **undefun** interchanges it with *symbol*'s function definition. This undoes the effect of a **defun**, **compile**, etc.

arglist *function*
arglist is given a function, and returns its best guess at the nature of the function's *lambda*-list.

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

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

Since **arglist** cannot be relied upon to return the exactly correct answer, it is not very useful in programs; it exists to be called by the user to get a little documentation on how to call a function. For program-usable information, use the function **args-info**.

args-info *function*
args-info returns a fixnum called the "numeric argument descriptor" of the *function*, which describes the way the function takes arguments. The information in it 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 with the number), and those starting with "%%" are byte descriptors (meant to be used with **ldb**).

Here are the fields:

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

%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.

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

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

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

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

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

%args-info *function*
This is an internal function of **args-info**; it is like **args-info** but only works for compiled-code objects. It exists because it has to be in the microcode anyway, for **apply**.

6.3 The Property List

Every symbol has associated with it a *property list*, which is a list used for associating "attributes" with symbols. A property list has an even number of elements. Each pair of elements constitutes a *property*; the first of the pair is a symbol called the *indicator*, and the second is a Lisp object called the *value* or, more loosely, the *property*. The indicator serves as the name of the property, and the value as the value of the property. Here is an example of the property list of a symbol named **b1** which is being used by a program which deals with blocks:

```
(color blue on b6 associated-with (b2 b3 b4))
```

There are three properties, and so the list has six elements. The first property's indicator 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 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**.

When a symbol is created, its property list is initially **nil**.

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 Lisp Machine lisp. The compiler (see page 126) and the editor use several properties, which are documented in those sections.

It is also possible to have a "disembodied" property list, which is not associated with any symbol. A disembodied property list is a cons. Its car may be used for any purpose. The property list resides in its cdr. The way to create a disembodied property list is with (**ncons nil**). In all of the functions in this section, disembodied property lists may be used as well as symbols; for brevity, the text speaks only of symbols.

get *sym indicator*

get looks up *sym*'s *indicator* property. If it finds such a property, it returns the value; otherwise, it returns **nil**.

Example: If the property list of **foo** is (**baz 3**), then

```
(get 'foo 'baz) => 3
(get 'foo 'zoo) => nil
```

getl *sym indicator-list*

getl is like **get**, except that the second argument is a list of indicators. **getl** searches down *sym*'s property list for any of the indicators in *indicator-list*, until it finds a property whose indicator is one of the elements of *indicator-list*.

getl returns the portion of *sym*'s property list beginning with the first such property which it found. So the *car* of the returned list is an indicator, and the *cadr* is the property value. If none of the indicators on *indicator-list* are on the property list, **getl** returns **nil**.

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)
```

putprop *sym x indicator*

This gives *sym* an *indicator*-property of *x*. After this is done, (**get** *sym indicator*) will return *x*.

Example:

```
(putprop 'Nixon 'not 'crook)
```

If *sym* already has a property with the name *indicator*, then that property is removed first; this insures that **getl** will always find the property that was added most recently.

defprop *Special Form*

defprop is a form of **putprop** with unevaluated arguments, which is sometimes more convenient for typing.

Example:

```
(defprop foo bar next-to)
```

is the same as

```
(putprop 'foo 'bar 'next-to)
```

remprop *sym indicator*

This removes *sym*'s *indicator* property, by splicing it out of the property list. It returns that portion of *sym*'s property list of which the former *indicator*-property was the **car**.

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 *sym* has no *indicator*-property, then **remprop** has no side-effect and returns **nil**.

plist *sym*

This returns the property list of *sym*.

setplist *sym property-list*

This sets the property list of *sym* to *property-list*. **setplist** is to be used with caution, 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.

property-cell-location *sym*

This returns a locative pointer to the location of *sym*'s property-list cell. See the section on locatives.

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 *print*ed, **print** types out the print-name. For more information, see the section on the *reader* (see page 156) and *printer* (see page 154).

samepnamep *sym1 sym2*

This predicate returns **t** if the two symbols *sym1* and *sym2* have **equal** print-names; that is, if their printed representation is the same. Upper and lower case letters are normally considered the same. If either or both of the arguments is a string instead of a symbol, then that string is used in place of the print-name.

Examples:

```
(samepnamep 'xyz (maknam '(x y z)) => t

(samepnamep 'xyz (maknam '(w x y)) => nil

(samepnamep 'xyz "xyz") => t
```

This is the same function as **string-equal** (see page 80).

get-pname *sym*

This returns the print-name of the symbol *sym*.

Example:

```
(get-pname 'xyz) => "xyz"
```

print-name-cell-location *sym*

This returns a locative pointer to the location of *sym*'s print-name cell. See the section on locatives. Note that the contents of this cell is not actually the print name, but the symbol header, an object which may not be directly manipulated. Use **get-pname**, the microcode primitive which knows how to extract the pname from the symbol header.

6.5 The Creation and Interning of Symbols

Normally, one wants to refer to the same symbol every time the same print-name-like string is typed. So, when **read** sees such a character-string in the input to Lisp, it looks in a table called the *obarray* for some symbol with that print-name. If it finds such a symbol, then that is what it returns; otherwise, it creates a symbol with that print-name (using the **make-symbol** function, see below), enters that symbol on the obarray, and returns it. The sub-function of **read** which performs these functions is called **intern**, and when a symbol has been entered on the obarray it is said to be *interned*.

A symbol can also be *uninterned*, indicating that it is not on the obarray and cannot be referred to simply by typing its print name. Such symbols can be used as objects within a data-structure, but can cause trouble during debugging because they cannot be "typed in" directly, yet they look just like interned symbols when "typed out".

Actually there can be many obarrays; the Lisp Machine system includes a feature called the *package system* (see page 176) which keeps track of multiple *packages* or *name spaces* and their interrelationships, using separate obarrays for each package.

make-symbol *pname* &optional *value definition plist package*

This creates a new uninterned symbol, whose print-name is the string *pname*. You may optionally supply the value binding, the function definition binding, the property list, and the owning package. These default to unbound, unbound, **nil**, and **nil** respectively. The package should be **nil** if the symbol is not going to be interned.

Examples:

```
(setq a (make-symbol "foo")) => foo
(symeval a) => ERROR!

(setq a (make-symbol "foo" 'bar)) => foo
(symeval a) => bar
```

Note that the symbol is *not* interned; it is simply created and returned.

copysymbol *sym copy-p*

This returns a new uninterned symbol with the same print-name as *sym*. If *copy-p* is non-**nil**, then the initial value and function-definition of the new symbol will be the same as those of *sym*, and the property list of the new symbol will be a copy of *sym*'s. If *copy-p* is **nil**, then the new symbol will be unbound and undefined, and its property list will be **nil**.

gensym &optional *x*

gensym 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 character 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 the first character of the string or of the print-name. After handling the argument, **gensym** creates a symbol as it would with no argument.

Examples:

```
if   (gensym) => g0007
then (gensym 'foo) => f0008
     (gensym 40) => f0032
     (gensym) => f0033
```

Note that the number is in decimal and always has four digits, and the prefix is always one character.

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".

package-cell-location *symbol*
Returns a locative pointer to *symbol*'s package cell, which contains the package (see page 176) which owns *symbol*.

7. Numbers

Lisp Machine Lisp includes several types of numbers, with different characteristics. Most numeric functions will 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. In Lisp Machine Lisp, 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 Lisp Machine Lisp are:

fixnum — Fixnums are 24-bit 2's complement binary integers. These are the "preferred, most efficient" type of number.

bignum — Bignums are arbitrary-precision binary integers.

flonum — Flonums are floating-point numbers. They have a mantissa of 32 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.

small-flonum — Small flonums are another form of floating-point number, with a mantissa of 18 bits and an exponent of 7 bits, providing a precision of about 5 digits and a range of about 10^19. Small flonums are useful because, like fixnums, they don't require any storage. Computing with small flonums is more efficient than with regular flonums.

Numbers are different from other objects in that they don't "have identity." To put it another way, **eq** does not work on them. Numbers do not behave "like objects." Fixnums and small flonums are exceptions to this rule; some system code knows that **eq** works on fixnums used to represent characters or small integers, and uses **memq** or **assq** on them.

The Lisp machine automatically converts between fixnums and bignums as necessary when computing with integers. That is, if the result of a computation with fixnums is too large to be represented as a fixnum, it will be represented as a bignum. If the result of a computation with bignums is small enough to be represented as a fixnum, it will be.

The Lisp machine never automatically converts between flonums and small flonums, since this would lead either to inefficiency or to unexpected numerical inaccuracies. The user controls whether floating-point calculations are done in large or small precision by the type of the original input data.

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. This will signal an error.

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 a

fixnum meets a bignum, the result is (usually) a bignum. When a fixnum or a bignum meets a small flonum or a flonum, the result is a small flonum or a flonum (respectively). When a small flonum meets a regular flonum, the result is a regular flonum.

Unlike Maclisp, Lisp Machine Lisp does not have number declarations in the compiler. Note that because fixnums and small flonums are "inums" (require no associated storage) they are as efficient as declared numbers in Maclisp.

The different types of numbers are distinguished by their printed representations. A leading or embedded decimal point, and/or an exponent separated by "e", indicates a flonum. If a number has an exponent separated by "s", it is a small flonum. Small flonums require a special indicator so that naive users will not be accidentally tricked into computing with the lesser precision. Fixnums and bignums have similar printed representations; the number is a bignum if it is too big to be a fixnum.

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.

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.

signp *Special Form*

signp is used to test the sign of a number. It is present only for Maclisp compatibility, and is not recommended for use in new programs. (**signp** *test x*) returns **t** if *x* is a number which satisfies the *test*, **nil** if it is not. *test* is not evaluated, but *x* is. *test* can be one of the following:

l	$x < 0$
le	$x \leq 0$
e	$x = 0$
n	$x \neq 0$
ge	$x \geq 0$
g	$x > 0$

Examples:

```
(signp le 12) => t
(signp n 0) => nil
(signp g 'foo) => nil
```

See also the data-type predicates **fixp**, **floatp**, **bigp**, **small-floatp**, and **numberp** (page 9).

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.

= *x y*

Returns **t** if *x* and *y* are numerically equal.

greaterp *x y* &rest *more-args*

greaterp compares its arguments from left to right. If any argument is not greater than the next, **greaterp** returns **nil**. But if the arguments are monotonically strictly decreasing, the result is **t**.

Examples:

```
(greaterp 4 3) => t
(greaterp 4 3 2 1 0) => t
(greaterp 4 3 1 2 0) => nil
```

> *x y*

Returns **t** if *x* is strictly greater than *y*, and **nil** otherwise.

>= *Macro*

≥ *Macro*

Returns **t** if *x* is greater than or equal to *y*, and **nil** otherwise.

lessp *x y* &rest *more-args*

lessp compares its arguments from left to right. If any argument is not less than the next, **lessp** returns **nil**. But if the arguments are monotonically strictly increasing, the result is **t**.

Examples:

```
(lessp 3 4) => t
(lessp 1 1) => nil
(lessp 0 1 2 3 4) => t
(lessp 0 1 3 2 4) => nil
```

< *x y*

Returns **t** if *x* is strictly less than *y*, and **nil** otherwise.

<= *Macro*
≤ *Macro*
Returns **t** if *x* is less than or equal to *y*, and **nil** otherwise.

≠ *Macro*
Returns **t** if *x* is not equal to *y*, and **nil** otherwise.

7.2 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.

plus &rest *args*
+ &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.

difference *arg* &rest *args*
Returns its first argument minus all of the rest of its arguments.

- *arg* &rest *args*
-$ *arg* &rest *args*
With only one argument, - is the same as **minus**; it returns the negative of its argument. With more than one argument, - is the same as **difference**; it returns its first argument minus all of the rest of its arguments.

times &rest *args*
***** &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.

quotient *arg* &rest *args*
Returns the first argument divided by all of the rest of its arguments.

// *arg* &rest *args*
//$ *arg* &rest *args*
The name of this function is written // rather than / because / is the quoting character in Lisp syntax and must be doubled. With more than one argument, // is the same as **quotient**; 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*).

Examples:

```
(// 3 2) => 1          ;Fixnum division truncates.
(// 3 2.0) => 1.5
(// 3 2.0s0) => 1.5s0
(// 4 2) => 2
(// 12. 2. 3.) => 2
```

add1 *x*
1+ *x*
1+$ *x*

(**add1** x) is the same as (**plus** x 1).

sub1 *x*
1- *x*
1-$ *x*

(**sub1** x) is the same as (**difference** x 1). Note that the short name may be confusing: (**1-** x) does *not* mean 1-x; rather, it means x-1.

remainder *x y*
**** *x y*

Returns the remainder of *x* divided by *y*. *x* and *y* may not be flonums nor small flonums.

gcd *x y*
**** *x y*

Returns the greatest common divisor of *x* and *y*. *x* and *y* may not be flonums nor small flonums.

expt *x y*
^ *x y*
^$ *x y*

Returns *x* raised to the *y*'th power. *y* must be a fixnum. [I guess this is incompatible with Maclisp.]

max &rest *args*

max returns the largest of its arguments.
Example:

```
(max 1 3 2) => 3
```

max requires at least one argument.

min &rest *args*

min returns the smallest of its arguments.
Example:

```
(min 1 3 2) => 1
```

min requires at least one argument.

abs *x*

Returns |*x*|, the absolute value of the number *x*. **abs** could have been defined by:

```
(defun abs (x)
    (cond ((minusp x) (minus x))
          (t x)))
```

minus *x*

Returns the negative of *x*.

Examples:

```
(minus 1) => -1
(minus -3) => 3
```

***dif** *x y*
***plus** *x y*
***quo** *x y*
***times** *x y*

These are the internal micro-coded arithmetic functions. There is no reason why anyone should need to refer to these explicitly, since the compiler knows how to generate the appropriate code for **plus**, **+**, etc. These names are only here for Maclisp compatibility.

The following functions are provided to allow specific conversions of data types to be forced, when desired.

fix *x*

Converts *x* to a fixnum.

float *x*

Converts *x* to a flonum.

small-float *x*

Converts *x* to a small flonum.

7.3 Random Functions

random &optional *arg* (*array* **si:random-array**)

(**random**) returns a random fixnum, positive or negative. If *arg* is present, a fixnum between 0 and *arg*-1 inclusive is returned. If *array* is present, the given array is used instead of the default one (see below). [The random algorithm should be described.]

si:random-create-array *size offset seed* &optional (*area* **default-array-area**)

Creates, initializes and returns a random-number-generator array. This is used for more advanced applications of the pseudo-random number generator, in which it is desirable to have several different controllable resettable sources of random numbers. For the exact meaning of the arguments, read the code.

size is the size of the array, *offset* is an integer less than *size*, *seed* is a fixnum. This

calls **si:random-initialize** on the random array before returning it.

si:random-initialize *array*
array must be a random-number-generator array, such as is created by **si:random-create-array**. It reinitializes the contents of the array from the seed (calling **random** changes the contents of the array and the pointers, but not the seed).

si:random-array *Variable*
The value of **si:random-array** is the default random-number-generator array. It is created if **random** is called and **si:random-array** is unbound. A random-number-generator array has a leader which is a structure with the following elements:

si:random-fill-pointer
The fill-pointer, the length of the array.

si:random-seed
The seed from which to initialize the contents.

si:random-pointer-1
The first pointer.

si:random-pointer-2
The second pointer.

7.4 Logical Operations on Numbers

Except for **lsh** and **rot**, these functions operate on either fixnums or bignums. As a compromise between consistency and Maclisp compatibility, there are some funny rules about negative numbers. Normally these functions will not accept negative inputs and will not produce negative results. However, if all the arguments to be logically combined are fixnums, the result will always be a fixnum, and consequently may be negative. In this case negative fixnums are accepted as input, and treated as the corresponding 24-bit 2's-complement representation.

logior &rest *args*
Returns the bit-wise logical *inclusive or* of its arguments. A minumum of one argument is required.
Example:

```
(logior 4002 67) => 4067
```

logxor &rest *args*
Returns the bit-wise logical *exclusive or* of its arguments. A minumum of one argument is required.
Example:

```
(logxor 2531 7777) => 5246
```

logand &rest *args*

Returns the bit-wise logical *and* of its arguments. A minumum of one argument is required.

Example:

```
(logand 3456 707) => 406
```

boole *fn* &rest *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. A minimum of two arguments is required.

Examples:

```
(boole 1 x y) = (logand x y)
(boole 6 x y) = (logxor x y)
```

logand, **logior**, and **logxor** are usually preferred over **boole**.

bit-test *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)))
```

ldb-test *ppss y*

ldb-test is a predicate which returns **t** if any of the bits designated by the byte specifier *ppss* are 1's in *y*. 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 ppss y) ==> (not (zerop (ldb ppss y)))
```

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.

Examples:

```
(lsh 4 1) => 10     ;(octal)
(lsh 14 -2) => 3
(lsh -1 1) => -2
```

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 24-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.

Examples:

```
(rot 1 2) => 4
(rot 1 -2) => 20000000
(rot -1 7) => -1
(rot 15 24.) => 15
```

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; note that if *x* is negative its absolute value is used.

haulong *x*

This returns the number of significant bits in *x*. *x* may be a fixnum or a bignum. The result does not depend on the sign of *x*. The result is the least integer not less than the base-2 logarithm of |*x*|+1.

Examples:

```
(haulong 0) => 0
(haulong 3) => 2
(haulong -7) => 3
```

7.5 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. Byte specifiers are 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, counting from the right in bits. A position of zero means that the byte is at the right end of the number. For example, the byte-specifier 0010 (i.e., 10 octal) refers to the lowest eight bits of a word, and the byte-specifier 1010 refers to the next eight bits. These byte-specifiers will be stylized below as *ppss*. The maximum value of the *ss* digits is 30 (octal), since a byte must fit in a fixnum although bytes can be loaded from and deposited into bignums. The format of byte-specifiers is taken from the pdp-10 byte instructions.

ldb *ppss num*

ppss specifies a byte of *num*, which is returned as a number, right-justified. The *ss* bits of the byte starting at bit *pp* are the lowest *ss* bits in the returned value, and the rest of the bits in the returned value are zero. The name of the function, **ldb**, means "load byte".

Example:

```
(ldb 0303 567) => 6
```

mask-field *ppss num*

This is similar to **ldb**; however, the specified byte of *num* is returned as a number in position *pp* of the returned word, instead of position 0 as with **ldb**.

Example:

```
(mask-field 0303 567) => 60
```

dpb *byte ppss num*

Returns a number which is the same as *num* except in the bits specified by *ppss*. The low *ss* bits of *byte* are placed in those bits. *byte* is interpreted as being right-justified, as if it were the result of **ldb**.

Example:

```
(dpb 2 0303 567) => 527
```

deposit-field *byte ppss num*

This is like **dpb**, except that *byte* is not taken to be left-justified; the *ppss* bits of *byte* are used for the *ppss* bits of the result, with the rest of the bits taken from *num*.

Example:

```
(deposit-field 20 0303 567) => 527
```

%logldb *ppss fixnum*

%logldb is like **ldb** except that it only loads out of fixnums and doesn't worry about negative numbers.

%logdpb *byte ppss fixnum*

%logdpb is like **dpb** except that it only deposits into fixnums and doesn't worry about negative numbers.

7.6 24-Bit Numbers

Sometimes it is desirable to have a form of arithmetic which has no overflow checking (which would produce bignums), and truncates results to the word size of the machine. In Lisp Machine Lisp, this is provided by the following set of functions. Their answers are only correct 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 should be fixnums.

%24-bit-difference *x y*
Returns the difference of *x* and *y* modulo 2^24. Both arguments should be fixnums.

%24-bit-times *x y*
Returns the product of *x* and *y* modulo 2^24. Both arguments should be fixnums.

7.7 Double-Precision Arithmetic

These peculiar functions are useful in programs that don't want to use bignums for one reason or another.

%multiply-fractions *num1 num2*
Returns bits 24 through 46 (the most significant half) of the product of *num1* and *num2*. If you call this and **%24-bit-times** on the same arguments *num1* and *num2*, regarding them as integers, you can combine the results into a double-precision product. If *num1* and *num2* are regarded as fractions, $-1 \leq num < 1$, **%multiply-fractions** returns 1/2 of their correct product as a fraction. (The name of this function isn't too great.)

%divide-double *dividend[24:46] dividend[0:23] 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 division by zero or if the quotient won't fit in single precision.

%remainder-double *dividend[24:46] dividend[0:23] divisor*
Divides the double-precision number given by the first two arguments by the third argument, and returns the remainder. Causes an error if division by zero.

%float-double *high24 low24*
high24 and *low24*, which must be fixnums, are concatenated to produce a 48-bit unsigned positive integer. A flonum 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. Strings

Strings are a type of array which are constants (they self-evaluate) and have as their printed representation a sequence of characters enclosed in quote marks, for example **"foo bar"**. Strings are the right data type to use for text-processing.

The functions described in this section provide a variety of useful operations on strings. Several of the functions actually work on any type of 1-dimensional array and may be useful for other than string processing. **art-16b** arrays (arrays of 16-bit positive numbers) are often used as strings; the extra bits allow for an expanded character set.

In place of a string, most of these functions will accept a symbol or a fixnum as an argument, and will coerce it into a string. Given a symbol, its print name, which is a string, will be used. Given a fixnum, a 1 character long string containing the character designated by that fixnum will be used.

Note that the length of a string is computed using **array-active-length**, so that if a string has an array-leader, element 0 of the leader (called the *fill pointer*) will be taken as the length.

Since strings are arrays, the usual array-referencing function **aref** is used to extract the characters of the string as fixnums. For example,

```
(aref "frob" 1) => 162  ;lower-case r
```

It is also legal to store into strings (using **aset**). As with **rplaca** on lists, this changes the actual object; one must be careful to understand where side-effects will propagate to.

8.1 String Manipulation

character *x*

> **character** coerces *x* to a single character, represented as a fixnum. If *x* is a number, it is returned. If *x* is a string or an array, its first element is returned. If *x* is a symbol, the first character of its pname is returned. Otherwise, an error occurs.

char-equal *ch1 ch2*

> This is the primitive for comparing characters for equality; many of the string functions call it. *ch1* and *ch2* must be fixnums. The result is **t** if the characters are equal ignoring case and font, otherwise **nil**. **%%ch-char** is the byte-specifier for the portion of a character which excludes the font information.

char-lessp *ch1 ch2*

> This is the primitive for comparing characters for order; many of the string functions call it. *ch1* and *ch2* must be fixnums. The result is **t** if *ch1* comes before *ch2* ignoring case and font, otherwise **nil**.

string *x*

string coerces *x* into a string. Most of the string functions apply this to their string arguments. If *x* is a string or an array, it is returned. If *x* is a symbol, its pname is returned. If *x* is a number, a 1-character long string containing it is returned. Otherwise, an error occurs.

string-length *string*

string-length returns the number of characters in *string*. This is 1 if *string* is a number, the **array-active-length** (see page 100) if *string* is an array, or the **array-active-length** of the pname if *string* is a symbol.

string-equal *string1 string2* &optional *(idx1 0) (idx2 0) lim1 lim2*

string-equal compares two strings, returning **t** if they are equal and **nil** if they are not. The comparison ignores the extra "font" bits in 16-bit strings, ignores font-change and other formatting characters (characters with numeric values between 240 and 377), and ignores alphabetic case. **equal** calls **string-equal** if applied to two strings.

The optional arguments *idx1* and *idx2* are the starting indices into the strings. The optional arguments *lim1* and *lim2* are the final indices; the comparison stops just *before* the final index. *lim1* and *lim2* default to the lengths of the strings. These arguments are provided so that you can efficiently compare substrings.
Examples:

```
(string-equal "Foo" "foo") => t
(string-equal "foo" "bar") => nil
(string-equal "element" "select" 0 1 3 4) => t
```

string-lessp *string1 string2*

string-lessp compares two strings using dictionary order. The result is **t** if *string1* is the lesser, and **nil** if they are equal or *string2* is the lesser.

substring *string start* &optional *end area*

This 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*

nsubstring is the same as **substring** except that the substring is not copied; instead an indirect array (see page 92) 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 the same amount of storage as a **substring** 12 characters long.

string-append &rest *strings*

Any number of strings are copied and concatenated into a single string. With a single argument, **string-append** simply copies it. If the first argument is an array, the result will be an array of the same type. Thus **string-append** can be used to copy and concatenate any type of 1-dimensional array.
Example:

```
(string-append 41 "foo" 41) => "!foo!"
```

string-trim *char-list string*

This returns a **substring** of *string*, with all characters in *char-list* stripped off of the beginning and end.
Example:

```
(string-trim '(40) "  Dr. No  ") => "Dr. No"
```

string-left-trim *char-list string*

This returns a **substring** of *string*, with all characters in *char-list* stripped off of the beginning.

string-right-trim *char-list string*

This returns a **substring** of *string*, with all characters in *char-list* stripped off of the end.

char-upcase *ch*

If *ch*, which must be a fixnum, is a lower-case alphabetic character its upper-case form is returned; otherwise, *ch* itself is returned. If font information is present it is preserved.

char-downcase *ch*

If *ch*, which must be a fixnum, is a upper-case alphabetic character its lower-case form is returned; otherwise, *ch* itself is returned. If font information is present it is preserved.

string-upcase *string*

Returns a copy of *string*, with all lower case alphabetic characters replaced by the corresponding upper case characters.

string-downcase *string*

Returns a copy of *string*, with all upper case alphabetic characters replaced by the corresponding lower case characters.

string-reverse *string*
Returns a copy of *string* with the order of characters reversed. This will reverse a 1-dimensional array of any type.

string-nreverse *string*
Returns *string* with the order of characters reversed, smashing the original string, rather than creating a new one. If *string* is a number, it is simply returned without consing up a string. This will reverse a 1-dimensional array of any type.

string-search-char *char string* &optional *(from 0)*
string-search-char searches through *string* starting at the index *from*, which defaults to the beginning, and returns the index of the first character which is **char-equal** to *char*, or **nil** if none is found.
Example:

```
(string-search-char 101 "banana") => 1
```

string-search-not-char *char string* &optional *(from 0)*
string-search-not-char searches through *string* starting at the index *from*, which defaults to the beginning, and returns the index of the first character which is *not* **char-equal** to *char*, or **nil** if none is found.
Example:

```
(string-search-char 102 "banana") => 1
```

string-search *key string* &optional *(from 0)*
string-search 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.
Example:

```
(string-search "an" "banana") => 1
(string-search "an" "banana" 2) => 3
```

string-search-set *char-list string* &optional *(from 0)*
string-search-set searches through *string* looking for a character which is in *char-list*. The search begins at the index *from*, which defaults to the beginning. It returns the index of the first character which is **char-equal** to some element of *char-list*, or **nil** if none is found.
Example:

```
(string-search-set '(116 117) "banana") => 2
```

string-search-not-set *char-list string* &optional *(from 0)*
string-search-not-set searches through *string* looking for a character which is not in *char-list*. The search begins at the index *from*, which defaults to the beginning. It returns the index of the first character which is not **char-equal** to any element of *char-list*, or **nil** if none is found.
Example:

```
(string-search-not-set '(141 142) "banana") => 2
```

string-reverse-search-char *char string* &optional *from*

string-reverse-search-char searches through *string* in reverse order, starting from the index one less than *from*, which defaults to the length 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.

Example:

```
(string-reverse-search-char 156 "banana") => 4
```

string-reverse-search-not-char *char string* &optional *from*

string-reverse-search-not-char searches through *string* in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, and returns the index of the first character which is *not* **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.

Example:

```
(string-reverse-search-not-char 101 "banana") => 4
```

string-reverse-search *key string* &optional *from*

string-reverse-search searches for the string *key* in the string *string*. The search proceeds in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, 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*.

Example:

```
(string-reverse-search "na" "banana") => 4
```

string-reverse-search-set *char-list string* &optional *from*

string-reverse-search-set searches through *string* in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, and returns the index of the first character which is **char-equal** to some element of *char-list*, 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.

```
(string-reverse-search-set '(141 142) "banana") => 5
```

string-reverse-search-not-set *char-list string* &optional *from*

string-reverse-search-not-set searches through *string* in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, and returns the index of the first character which is not **char-equal** to any element of *char-list*, 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.

```
(string-reverse-search-not-set '(141 156) "banana") => 0
```

See also **intern** (page 184), which given a string will return "the" symbol with that print name.

8.2 Maclisp-compatible Functions

alphalessp *string1 string2*

(**alphalessp** *string1 string2*) is equivalent to (**string-lessp** *string1 string2*).

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 (however **aref** will 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 (however **aref** will not coerce symbols or numbers into strings).

ascii *x*

ascii is like **character**, but returns a symbol whose printname is the character instead of returning a fixnum.

Examples:

```
(ascii 101) => A
(ascii 56) => /.
```

The symbol returned is interned in the **user** package.

maknam *char-list*

maknam returns an uninterned symbol whose print-name is a string made up of the characters in *char-list*.

Example:

```
(maknam '(a b 60 d)) => ab0d
```

implode *char-list*

implode is like **maknam** except that the returned symbol is interned in the current package.

The **samepnamep** function is also provided; see page 65.

8.3 Formatted Output

format *destination control-string* &rest *args*

format is used to produce formatted output. **format** outputs the characters of *control-string*, except that a tilde ("~") introduces a directive. The character after the tilde, possibly preceded by arguments and modifiers, specifies what kind of formatting is desired. Some directives use an element of **args** to create their output.

The output is sent to *destination*. If *destination* is **nil**, a string is created which contains the output. If *destination* is a stream, the output is sent to it. If *destination* is **t**, the output is sent to **standard-output**.

A directive consists of a tilde, optional decimal numeric arguments 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. Examples of control strings:

```
"~S"        ; This is an S directive with no arguments.
"~3,4:@s"   ; This is an S directive with two arguments, 3 and 4,
            ;   and both the colon and atsign flags.
```

The kinds of directives will now be described. *arg* will be used to refer to the next argument from *args*.

~D — *arg*, a number, is printed as a decimal integer. ~*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 40 (space).

~O — This is just like ~D but prints in octal instead of decimal.

~F — *arg* is printed in floating point. Not yet implemented nor fully defined.

~E — *arg* is printed in exponential notation. Not yet implemented nor fully defined.

~A — *arg*, any Lisp object, is printed without slashification (like **princ**). ~*n*A inserts spaces on the right, if necessary, to make the column width at least *n*. ~*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*, 1 for *colinc* and *minpad*, and 40 (space) for *padchar*.

~S — This is just like ~A, but *arg* is printed *with* slashification (like **prin1** rather than **princ**).

~C — (**character** *arg*) is printed as a keyboard character, whose bits are described by the **%%kbd-** fields (see page 152). Control and meta bits

are printed as a preceding alpha (control), beta (meta), or epsilon (control and meta); the characters alpha, beta, epsilon, and equivalence-sign are preceded by an equivalence-sign to quote them. With the colon flag (i.e. ~:C), the control and meta bits, as well as non-printing characters (those in the 200 to 377 range) are spelled out. With both colon and atsign, characters which are typed in using the "TOP" key produce something like "≥ (Top-S)".

~P — If *arg* is not **1**, a lower-case s is printed.

~* — *arg* is ignored. ~*n** ignores the next *n* arguments.

~% — Outputs a newline. ~*n*% outputs *n* newlines. No argument is used.

~& — 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 newline.

~| — Outputs a formfeed. ~*n*| outputs *n* formfeeds.

~X — Outputs a space. ~*n*X outputs *n* spaces.

~T — Spaces over to a given column. ~*n*,*m*T will output sufficient spaces to move the cursor to column *n*. If the cursor is already past column *n*, it will output 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 *only* works properly on streams that support the **:read-cursorpos** and **:set-cursorpos** stream operations (see page 167). On other streams (and when **format** is creating a string), any ~T operation will simply output two spaces.

~~ — Outputs a tilde. ~*n*~ outputs *n* tildes.

~*n*C — "Goes to" the *n*th argument. ~**0**C goes back to the first argument in *args*. Directives after a ~*n*C will take sequential arguments after the one gone to.

~[— This begins a set of alternative control strings. The alternatives 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 *arg* is out of range no alternative is selected. After the selected alternative has been processed, the control string continues after the ~].

~:[*false*~;*true*~] selects the *false* control string if *arg* is **nil**, and selects the *true* control string otherwise.

~; — Separates alternatives after ~[.

~] — Ends a ~[construction.

~R *arg* is printed as a cardinal English number, e.g. four. With the colon modifier, *arg* is printed as an ordinal number, e.g. fourth. With the atsign modifier, *arg* is printed as a Roman numeral, e.g. IV. With both atsign and colon, *arg* is printed as an old Roman numeral, e.g. IIII.

In place of a numeric argument to a directive, you can put the letter V, which takes an argument from *args* as an argument 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.

The user can define his 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\).
Examples:

```
(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." 1003)
     => "The character Meta-β (Top-X) is strange."
(setq n 3)
(format nil "~D item~P found." n n) => "3 items found."
(format nil "~R dog~:[s are~; is~] here." n (= n 1))
     => "three dogs are here."
```

format also allows *control-string* to be a list of strings and lists, which is processed from left to right. Strings are interpreted as in the simple case. Lists are taken as extended directives; the first element is the directive letter, and the remaining elements are the numeric arguments to the directive. If the car of a list is r . recognized directive, the list is simply evaluated as a form; anything it writes to .e **standard-output** stream will appear in the result of **format**.

For formatting Lisp code (as opposed to text and tables), there is the Grind package. See <not-yet-written>.

9. Arrays

9.1 What Arrays Are

An *array* is a Lisp object that consists of a group of cells, each of which may contain a Lisp object. The individual cells are selected by numerical ***subscripts***.

There are many types of arrays. Some types of arrays can hold Lisp objects of any type; the other types of arrays can only hold fixnums. The array types are known by a set of symbols symbols whose names begin with "**art-**" (for ARray Type).

array-types *Variable*
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.

array-types *array-type-code*
An array of the array type symbols, indexed by their internal numeric codes.

array-elements-per-q *Variable*
array-elements-per-q is an association list (see page 52) which associates each array type symbol with the number of array elements stored in one word, for an array of that type.

array-elements-per-q *array-type-code*
This is an array, indexed by the internal codes of the array types, containing the number of array elements stored in one word, for an array of that type.

array-bits-per-element *Variable*
The value of **array-bits-per-element** is an association list (see page 52) 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*
This is an array, indexed by the internal codes of the array types, containing the number of bits per cell for unsigned numeric arrays, and **nil** for full-Lisp-object-containing array.

array-element-size *array*
Given an array, returns the number of bits that fit in an element of that array. For non-numeric arrays, the result is 24., assuming you will be storing unsigned fixnums in the array.

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** will take an **art-q-list** array and return a list object 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 will change, and if you store into the array, the corresponding element of the list will change the same way.

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-1b** array is a fixnum, and only one bit (the least significant) is remembered in the array; all of the others are discarded. Similarly, in an **art-2b** array, only the two least significant bits are remembered. So if you store a 5 into an **art-2b** array, for example, 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 (decimal) of an **art-1b** array or 2 elements of an **art-16b** array will fit into one word).

Character strings are implemented by the **art-string** array type. This type acts similarly to the **art-8b**; its elements must be fixnums, of which only the least significant eight bits are stored. However, many important system functions, including **read**, **print**, and **eval**, treat **art-string** arrays very differently from the other kinds of arrays. These arrays are usually called *strings*, and an entire chapter of this manual deals with functions which manipulate them.

There are three types of arrays which exist only for the purposes 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 page 105).

There are also two array types which exist only for the TV output device; these are called **art-tvb** and **art-tvb-pixel**. The former holds one bit of a fixnum (like an **art-1b** array), and the latter is more complicated. Their use is described in the section on the TV (see page 210).

9.2 How Arrays Work

The *dimensionality* of an array (or, 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 dimensionality may be any integer from one to seven, inclusively.

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.

The most basic primitive subrs 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. 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. The Lisp machine 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 101) is also supported. This form of array referencing is considered to be obsolete, and should not be used in new programs.

Here are some issues of Maclisp compatibility:

Fixnum arrays do not exist (however, see the Lisp machine's small-positive-number arrays). Flonum arrays do not (currently) exist. "Un-garbage-collected" arrays do not exist. Readtables and obarrays are represented as arrays, but unlike Maclisp special array types are not used. See the descriptions of **read** (page 159) and **intern** (page 184) for information about readtables and obarrays (packages). There are no "dead" arrays, nor are Multics "external" arrays provided.

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 only caused if the subscripts would have resulted in a reference to storage outside of the array; so if you have a 2 by 7 array and refer to an element with subscripts 3 and 1, no error will be caused despite the fact that the reference is invalid; but if you refer to element 1 by 100, an error will be caused. In other words, any subscript error which is not detected will only refer to somewhere else in your array, and not to any other part of storage.

loadarrays and **dumparrays** are not provided. However, arrays can be put into "QFASL" files; see the section on fasloading (page 194).

9.3 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. It can be stored in and examined by a special set of functions which are analogous to 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 dimensionality of the array.

By convention, the zeroth element of the array leader of an array is used to hold the number of elements in the array that are "active" in some sense. When the zeroth element is used this way, it is called a *fill pointer*. Specifically, if a string (an array of type **art-string**) has seven elements, but it has a fill pointer of five, then only elements zero through four of the string are considered to be "active"; the string's printed representation will be five characters long, string-searching functions will stop after the fifth element, etc.

The second element is also used in conjunction with the "named structure" feature; see below.

[Note: The named-structure feature is going to be revised in the future, and the following material will become incorrect.]

Any array may be a *named structure*. Several functions (currently the printer and **describe**), when given an array, check to see if the array is a named structure and take special action accordingly.

Within each named structure array there is a symbol called the *named structure symbol*. If the array has a leader, then the symbol is found in element 1 of the leader; otherwise it is found in element 0 of the array. (Note: if a numeric-type array is to be a named structure, it must have a leader, since a symbol cannot be stored in any element of a numeric array.)

The symbol should be defined as a function. The functions which know about named structures will apply this function to several arguments. The first is a "keyword" symbol to identify the calling function, and the second is the named structure itself. The rest of the arguments passed depend on the caller; any named structure function should have a "**&rest**" parameter to absorb any extra arguments that might be passed. Just what the function is expected to do depends on the keyword it is passed as its first argument.

Using named structures, you can control the printed representation of your array, and also you can control what a user gets if he tries to **describe** it. Currently, the keyword will be **:print** for the printer, and **:describe** for **describe**. See the documentation on the printer and on **describe** for explanations of what the named structure function should do.

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 consists of 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. 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 as its fourth argument, it will create a displaced array refering to that location of virtual memory. 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 could be damaged by the garbage collector. 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. See the description of internal array formats on <not-yet-written>.

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 its fourth argument. 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 dimensionality, the manner of accessing the elements differs. Thus, by creating a one-dimensional array of nine elements which was indirected to a second, two-dimensional array of three elements by three, then the elements could be accessed in either a one-dimensional or a two-dimensional manner. Even more complex effects can be produced if the new array is of a different type than the old array; see the description of internal array formats on <not-yet-written>.

It is also possible to create a one-dimensional 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*, and is specified at the time the indirect array is created, by giving a fixnum to **make-array** as its sixth argument. The **nsubstring** function (see page 80) creates such arrays.

9.4 Basic Array Functions

make-array *area type dims* &optional *displaced-p leader index-offset named-structure*
This creates and returns an array, according to various specifications.

The *area* parameter specifies the area in which to allocate the array's storage; if you are not concerned with areas, simply use the value of **default-array-area**. For convenience, if *area* is **nil**, **default-array-area** is used instead.

type should be a symbolic name of an array type; the most common of these is **art-q**. The elements of the array are initialized according to the type: if the array is of a type whose elements may only be fixnums, then every element of the array will initially be **0**; otherwise, every element will initially be **nil**. See the description of array types on page 88.

dims should be a list of fixnums which are the dimensions of the array; the length of the list will be the dimensionality of the array. For convenience, if the dimensionality should be one, the single dimension may be provided as a fixnum in place of the list.
Examples:

```
(setq a (make-array nil 'art-q 5))          ; Create a one-d array
                                            ;of 5 elements.

(setq b (make-array nil 'art-4b '(3 4)))    ; Create a four-bit two-d
                                            ;array, 3 by 4.
```

If *displaced-p* is not **nil**, then the array will be a *displaced* array. *displaced-p* may either be a fixnum, to create a regular displaced array which refers to a certain section of virtual address space, or an array, to create an indirect array (see page 92).

If *leader* is not **nil**, then the array will be given a leader. If *leader* is a fixnum, the array's leader will be *leader* elements long, and its elements will be initialized to **nil**. *Leader* may also be a list, in which case the length of the leader is equal to that of the list, and the elements are initialized to the elements of the list, in reverse order (i.e., the car of the list is stored in the highest-subscripted location in the leader).

If *index-offset* is present, *displaced-p* should be an array, and *index-offset* should be a fixnum; it is made to be the index-offset of the created indirect array. (See page 92.)

If *named-structure* is not **nil**, it is a symbol to be stored in the named-structure cell element of the array. The array created will be a named structure.

Examples:

```
(make-array nil 'art-q 5 nil 3)    ;;leader 3 elements long.
(setq a (make-array nil 'art-1b 100 nil '(t nil)))
(array-leader a 0) => nil
(array-leader a 1) => t
```

make-array returns the newly-created array, and also returns, as a second value, the number of words allocated from *area* in the process of creating the array.

array-displaced-p *array*

array may be any kind of array. This predicate returns **t** if *array* is any kind of displaced array (including indirect arrays). Otherwise it returns **nil**.

array-indirect-p *array*

array may be any kind of array. This predicate returns **t** if *array* is an indirect array. Otherwise it returns **nil**.

array-indexed-p *array*

array may be any kind of array. This predicate returns **t** if *array* is an indirect array with an index-offset. Otherwise it returns **nil**.

adjust-array-size *array new-size*

array should be a one-dimensional array. Its size is changed to be *new-size*. If this results in making *array* smaller, then the extra elements are lost; if *array* is made bigger, the new elements are initialized in the same fashion as **make-array** (see page 93): either to **nil** or **0**. [Currently there is a bug which causes initialization to zero not to work.]

Example:

```
(setq a (make-array nil 'art-q 5))
(aset 'foo a 4)
(aref a 4) => foo
(adjust-array-size a 2)
(aref a 4) => ERROR
```

If the size of the array is being increased, **adjust-array-size** must allocate a new array somewhere; it then alters *array* so that references to it will be made to the new array instead, by means of an "invisible pointer". **adjust-array-size** will return this new array if it creates one, and otherwise it will return *array*. Be careful 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**) which share the same contents.

return-array *array*

Return *array* to free storage. If it is displaced, this returns the pointer, not the data pointed to. Currently does nothing if the array is not at the end of its area. This will eventually be renamed to **reclaim**, when it works for other objects than arrays.

aref *array* &rest *subscripts*
Returns the element of *array* selected by the *subscripts*. The *subscripts* must be fixnums and their number must match the dimensionality of *array*.

ar-1 *array i*
array should be a one-dimensional array, and *i* should be a fixnum. This returns the *i*'th element of *array*.

ar-2 *array i j*
array should be a two-dimensional array, and *i* and *j* should be fixnums. This returns the *i* by *j*'th element of *array*.

ar-3 *array i j k*
array should be a three-dimensional array, and *i*, *j*, and *k* should be fixnums. This returns the *i* by *j* by *k*'th element of *array*.

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 dimensionality of *array*.

as-1 *x array i*
array should be a one-dimensional array, and *i* should be a fixnum. *x* may be any object. *x* is stored in the *i*'th element of *array*. **as-1** returns *x*.

as-2 *x array i j*
array should be a two-dimensional array, and *i* and *j* should be fixnums. *x* may be any object. *x* is stored in the *i* by *j*'th element of *array*. **as-2** returns *x*.

as-3 *x array i j k*
array should be a three-dimensional array, and *i*, *j*, and *k* should be fixnums. *x* may be any object. *x* is stored in the *i* by *j* by *k*'th element of *array*. **as-3** returns *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 dimensionality of *array*.

ap-1 *array i*
array should be a one-dimensional array whose elements contain Lisp objects, and *i* should be a fixnum. This returns a locative pointer to the *i*'th element of *array*. See the explanation of locatives, page 109.

ap-2 *array i j*

array should be a two-dimensional array whose elements contain Lisp objects, and *i* and *j* should be fixnums. This returns a locative pointer to the *i* by *j*'th element of *array*. See the explanation of locatives, page 109.

ap-3 *array i j k*

array should be a three-dimensional array whose elements contain Lisp objects, and *i*, *j*, and *k* should be fixnums. This returns a locative pointer to the *i* by *j* by *k*'th element of *array*. See the explanation of locatives, page 109.

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 3 dimensions the compiler uses the slightly less efficient form since the special routines only exist for 1, 2, and 3 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 existing programs use these functions. New programs should use **aref**, **aset**, and **aloc**.

arraycall *ignored array* &rest *subscripts*

(**arraycall nil** *array sub1 sub2...*) is the same as (**aref** *array sub1 sub2...*). It exists for Maclisp compatibility.

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 array which has been referenced.

g-l-p *array*

array should be an **art-q-list** array. This returns a list which shares the storage of *array*. The **art-q-list** type exists so that **g-l-p** can be used.

Example:

```
(setq a (make-array nil 'art-q-list 4))
(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)
```

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**.

arraydims *array*

array may be any array; it also may be a symbol whose function cell contains an array, for Maclisp compatibility (see page 100). It returns a list whose first element is the symbolic name of the type of *array*, and whose remaining elements are its dimensions.

Example:

```
(setq a (make-array nil 'art-q '(3 5)))
(arraydims a) => (art-q 3 5)
```

array-dimensions *array*

array-dimensions returns a list whose elements are the dimensions of *array*.

Example:

```
(setq a (make-array nil 'art-q '(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*).

array-in-bounds-p *array* &rest *subscripts*

This function checks whether the *subscripts* are all legal subscripts for *array*, and returns **t** if they are; otherwise it returns **nil**.

array-length *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** (see page 100)).

Example:

```
(array-length (make-array nil 'art-q 3)) => 3
(array-length (make-array nil 'art-q '(3 5)))
                    => 17  ;octal, which is 15. decimal
```

array-/#-dims *array*

Returns the dimensionality of *array*. Note that the name of the function includes a "#", which must be slashified if you want to be able to compile your program with the compiler running in Maclisp.

Example:

```
(array-/#-dims (make-array nil 'art-q '(3 5))) => 2
```

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 dimensionality of *array*, this returns the *n*'th dimension of *array*. If *n* is **0**, it returns the length of the leader of *array*; if *array* has no leader it returns **nil**. If *n* is any other value, it returns **nil**.

Examples:

```
(setq a (make-array nil 'art-q '(3 5) nil 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-type *array*

Returns the symbolic type of *array*.

Example:

```
(setq a (make-array nil 'art-q '(3 5)))
(array-type a) => art-q
```

fillarray *array x*

Note: for the present, all arrays concerned must be one-dimensional.

array may be any type of array, or, for Maclisp compatibility, a symbol whose function cell contains an array. 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 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.

fillarray returns *array*.

listarray *array* &optional *limit*

Note: for the present, all arrays concerned must be one-dimensional.

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*.

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. *Presently the first subscript varies fastest in multi-dimensional arrays (opposite from Maclisp).* If *to* is shorter than *from*, the excess is ignored. If *from* is shorter than *to*, the rest of *to* is filled with **nil** if it is a q-type array or 0 if it is a numeric array. **t** is always returned.

9.5 Named Structures

Named structures were introduced at the beginning of the chapter. This section presents various functions which operate on named structures.

named-structure-p *x*
This predicate returns **t** if *x* is a named structure; otherwise it returns **nil**.

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

make-array-into-named-structure *array*
array is made to be a named structure, and is returned.

9.6 Array Leaders

Array leaders were introduced at the beginning of the chapter. This section presents various functions which operate on array leaders.

array-has-leader-p *array*
array may be any array. This predicate returns **t** if *array* has a leader; otherwise it returns **nil**.

array-leader-length *array*
array may be any array. This returns the length of *array*'s leader if it has one, or **nil** if it does not.

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**.

ap-leader *array i*
array should be an array with a leader, and *i* should be a fixnum. This returns a locative pointer to the *i*'th element of *array*'s leader. See the explanation of locatives, page 109. This is analogous to **aloc**.

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, which is at the beginning of this section.

array-push *array x*

array must be a one-dimensional array which has a fill pointer, and *x* may be any object. **array-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 **array-push** returns **nil**; otherwise, the two actions (storing and incrementing) happen uninterruptibly, and **array-push** returns the *former* value of the fill pointer (one less than the one it leaves in the array). 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.

array-push-extend *array x*

array-push-extend is just like **array-push** except that if the fill pointer gets too large, the array is grown to fit the new element; i.e. it never "fails" the way **array-push** does, and so never returns **nil**.

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 has 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.

copy-array-contents-and-leader *from to*

This is just like **copy-array-contents** (see page 98), but the leaders of *from* and *to* are also copied.

9.7 Maclisp Array Compatibility

Note: the functions in this section should not be used in new programs.

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. However, the ***rearray**, **loadarrays**, and **dumparrays** functions are not provided. Also, **flonum**, **readtable**, and **obarray** type arrays are not supported.

array &quote *symbol type* &eval &rest *dims*
This creates an **art-q** type array in **default-array-area** with the given dimensions. (That is, *dims* is given to **make-array** as its third 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*
This is just like **array**, except that all of the arguments are evaluated.

store &quote *array-ref x*
x may be any object; *array-ref* should be a form which references an array. First *x* is evaluated, then *array-ref* is evaluated, and then the value of *x* is stored into the array cell which was referenced by the evaluation of *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, and should never be used by programs.

10. Closures

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

10.1 What a Closure Is

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

For example, consider the following sequence of Lisp forms:

```
(setq a 3)

((lambda (a)
    (print (+ a 6)))
 10)

(print a)
```

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

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

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

Now, if we evaluate the form

```
(setq a
      ((lambda (x)
               (closure '(x) (function car)))
       3))
```

what happens is that a new value cell is created for **x**, and its contents is a fixnum **3**. Then a closure is created, which remembers the function **car**, the symbol **x**, and that value cell. Finally the old value cell of **x** is restored, and the closure is returned. Notice that the new value cell is still around, because it is still known about by the closure. When the closure is applied, this value cell will be restored and the value of **x** will be **3**.

Because of the way closures are implemented, the variables to be closed over must not get turned into "local variables" by the compiler. Therefore, all such variables should be declared special.

In the Lisp Machine's implementation of closures, lambda-binding never really allocates any storage to create new value cells. Value cells are only created (sometimes) by the **closure** function itself. Thus, implementors of large systems need not worry about storage allocation overhead from this mechanism if they are not using closures. See the section on internal formats.

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

10.2 Examples of the Use of Closures

This section gives some examples of things that can be done easily and elegantly with closures, which would be difficult to do without them.

We will start with a simple example of a generator. A *generator* is a kind of function which is called successively to obtain successive elements of a sequence. We will implement a function **make-list-generator**, which takes a list, and returns a generator which will return successive elements of the list. When it gets to the end it should return **nil**.

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

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

```
(defun make-list-closure (l)
    (closure '(l)
             (function (lambda ()
                               (progl (car l)
                                      (setq l (cdr l)))))))
```

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 value cell for **l**. Each of these value cells was created when the **make-list-closure** function was entered, and the value cells are remembered by the closures.

10.3 Function Descriptions

closure *var-list function*

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

symeval-in-closure *closure symbol*

This returns the binding of *symbol* in the environment of *closure*; that is, it returns what you would get if you restored the value cells known about by *closure* and then evaluated *symbol*. This allows you to "look around inside" a closure.

set-in-closure *closure symbol x*

This sets the binding of *symbol* in the envorinment of *closure* to *x*; that is, it does what would happen if you restored the value cells known about by *closure* and then set *symbol* to *x*. This allows you to change the contents of the value cells known about by a closure.

let-closed *Macro*

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

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

expands into

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

11. 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. 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 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 remembers all functions which were active at the time of the resumption (that is, the running function, its caller, its caller's caller, etc.), and where in each function the computation was up to. In other words, the entire control stack (or regular pdl) is saved. In addition, the bindings that were present are saved also; that is, the environment stack (or special pdl) is saved. When the state of the current stack group is saved away, all of its bindings are undone, and when the state is restored, the bindings are put back. Note that although bindings are temporarily undone, unwind-protect handlers are *not* run (see **let-globally**).

There are several ways that a resumption can happen. First of all, there are several Lisp functions, described below, which resume some other stack group. When some stack group (call it **c**) calls such a function, it is suspended in the state of being in the middle of a call to that function. When someone eventually resumes **c**, the function will return. The arguments to these functions and the returned values can therefore be used to pass information back and forth between stack groups. Secondly, if an error is signalled, the current stack group resumes an error handler stack group, which handles the error in some way. Thirdly, a *sequence break* can happen, which transfers control to a special stack group called the *scheduler* (see page 195).

Note: the following discussion of resumers is incomplete, and the way they work is being changed anyway.

Each stack group has a *resumer*. *c*'s resumer is some other stack group, which essentially is the last stack group to resume *c*. This is not completely right, however, because some resume-forms set the resumed stack group's resumer, and some don't. So *c*'s resumer is actually the last stack group to resume *c* by means of one of the types of resume-form which does set the resumer.

si:%current-stack-group-previous-stack-group *Variable*
The binding of this variable is the resumer of the current stack group.

There are currently four kinds of resume-forms:

1) If *c* calls *s* as a function with an argument *x*, then *s* is resumed, and the object transmitted is *x*. *s*'s resumer is now *c*.

2) If *c* evaluates (**stack-group-return** *x*), then its resumer is resumed, and the object transmitted is *x*. The resumer's resumer is not affected.

3) If *c* evaluates (**stack-group-resume** *s x*), then *c* is resumed, and the object transmitted is *x*. *c*'s resumer is not affected. (This is not currently implemented.)

4) 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 returning, its resumer is resumed, and the value transmitted is *x*. The resumer's resumer is not affected. *c* is left in a state from which it cannot be resumed again; any attempt to resume it would signal an error.

There is one other way a stack group can be resumed. If the running stack group *c* gets a microcode trap, then the error handler stack group is resumed. The object transmitted is **nil**, and the error handler's resumer is set to *c*. This kind of resuming will only happen to the error handler, so regular programs should not see it.

11.1 What is Going On Inside

The stack group itself holds a great deal of state information. First of all, it contains the control stack, or "regular PDL". The control stack is what you are shown by the backtracing commands of the error handler (currently the Control-B and Meta-B commands); it remembers the function which is running, its caller, its caller's caller, and so on, and remembers the point of execution of each function (i.e. the "return addresses" of each function). Secondly, it contains the environment stack, or "special PDL". This contains all of the values saved by **lambda**-binding. Finally, it contains various internal state information (contents of machine registers and so on).

When one stack group resumes a second, the first thing that happens is that (some of) the state of the processor is saved in the first stack group. Next, all of the bindings in effect are undone; each stack group has its own environment, and the bindings done in one stack group do not affect another stack group at all. Then the second stack group's bindings are restored, its machine state is restored, and the second stack group proceeds from where it left off. While these things are happening, the transmitted object is passed into the second stack group, and optionally the second stack group's resumer is made to be the first stack group.

si:%current-stack-group *Variable*

The value of **si:%current-stack-group** is the stack group which is currently running. A program can use this variable to get its hands on its own stack group.

make-stack-group *name* &optional *options*

This creates and returns a new stack group. *name* may be any symbol; it is used to identify and print the stack group. Each option is a keyword followed by a value for that option; any number of options may be given, including zero. The options are not too useful; most calls to **make-stack-group** don't have any options at all. The options are:

:sg-area The area in which to create the stack group structure itself. Defaults to **default-array-area**.

:regular-pdl-area
The area in which to create the regular PDL. Note that this may not be any area; only certain areas may hold regular PDL, because accessing a regular PDL as memory must go through special microcode which checks an internal cache called the *pdl buffer*. Defaults to **error-linear-pdl-area**.

:special-pdl-area
The area in which to create the special PDL. Defaults to **default-array-area**.

:regular-pdl-size
Length of the regular PDL to be created. Defaults to 3000.

:special-pdl-size
Length of the special PDL to be created. Defaults to 400.

:car-sym-mode
The "error mode" which determines the action taken when there is an attempt to apply **car** to a symbol. This, and the other "error mode" options, are documented with the fucntions **car** and **cdr**. Defaults to **1**.

:car-num-mode
As above, for applying **car** to a number. Defaults to 0.

:cdr-sym-mode
As above, for applying **cdr** to a symbol. Defaults to 1.

:cdr-num-mode
As above, for applying **cdr** to a number. Defaults to 0.

:swap-sv-on-call-out

:swap-sv-of-sg-that-calls-me

: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.

:safe If **1** (the default), a strict call-return discipline among stack-groups is enforced. If **0**, no restriction on stack-group switching is imposed.

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. **stack-group-preset** is used to initialize a stack group just after it is made, but it may be done to any stack group at any time.

stack-group-return *x*

Let *s* be the current stack group's resumer; **stack-group-return** will resume *s*, transmitting the value *x*. *s*'s resumer is not affected.

stack-group-resume *s x*

stack-group-resume will resume *s*, transmitting the object *x*. *s*'s resumer is not affected. This function is not currently implemented.

12. Locatives

12.1 Cells and 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. Most programmers will never need them.

A cell is a machine word which contains a (pointer to a) Lisp object. 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 array of *n* elements has *n* cells provided the array is not a numeric array. However, a numeric array contains a different kind of cell, which cannot be pointed to by a locative.

There are a set of functions which create locatives to cells; the functions are documented with the kind of object to which they create a pointer. See **ap-1**, **ap-leader**, **car-location**, **value-cell-location**, etc. The macro **locf** (see page 146) can be used to convert a form which accesses a cell to one which creates a locative pointer to that cell: for example,

```
(locf (fsymeval x)) ==> (function-cell-location x)
```

12.2 Functions Which Operate on Locatives

Either of the functions **car** and **cdr** (see page 38) may be given a locative, and will return the contents of the cell at which the locative points.

For example,

```
(car (value-cell-location x))
```

is the same as

```
(symeval x)
```

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 (value-cell-location x) y)
```

is the same as

```
(set x y)
```

If you mix locatives and lists, then it matters whether you use **car** and **rplaca** or **cdr** and **rplacd**, and care is required. For example, this function takes advantage of **value-cell-location** to cons up a list in forward order without special-case code. 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 sort-of-mapcar (fcn lst)
  (do ((lst lst (cdr lst))
       (res nil)
       (loc (value-cell-location 'res)))
      ((null lst) res)
    (rplacd loc
            (setq loc (ncons (funcall fcn (car lst)))))))
```

You might expect this not to work if it was compiled and **res** was not declared special, since non-special compiled variables are not represented as symbols. However, the compiler arranges for it to work anyway.

13. 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. Subprimitives usually have names which start with a % character. The "primitives" described in other sections of the manual typically use subprimitives to accomplish their work. The subprimitives take the place of machine language in other systems, to some extent. Subprimitives are normally hand-coded in microcode.

Subprimitives by their very nature cannot do full checking. Improper use of subprimitives can destroy the environment.

13.1 Data Types

data-type *arg*

data-type returns a symbol which is the name for the internal data-type of the "pointer" which represents *arg*. 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.

si:dtp-symbol
The object is a symbol.

si:dtp-fix The object is a fixnum; the numeric value is contained immediately in the pointer field.

si:dtp-small-flonum
The object is an immediate small floating-point number.

si:dtp-extended-number
The object is a flonum or a bignum. This value will be used for future numeric types.

si:dtp-list The object is a cons.

si:dtp-locative
The object is a locative pointer.

si:dtp-array-pointer
The object is an array.

si:dtp-fef-pointer
The object is a fef.

si:dtp-u-entry
The object is a microcode entry.

si:dtp-closure
The object is a closure.

si:dtp-stack-group
The object is a stack-group.

si:dtp-instance
The object is an "active object". These are not documented yet.

si:dtp-entity The same as **dtp-closure** except it is a kind of "active object". These are not documented yet.

si:dtp-select-method
Another type associated with "active objects" and not documented yet.

si:dtp-header An internal type used to mark the first word of a multi-word structure.

si:dtp-array-header
An internal type used in arrays.

si:dtp-symbol-header
An internal type used to mark the first word of a symbol.

si:dtp-instance-header
An internal type used to mark the first word of an instance.

si:dtp-null Nothing to do with **nil**. This is used in unbound value and function cells.

si:dtp-trap The zero data-type, which is not used. This hopes to detect microcode errors.

si:dtp-free This type is used to fill free storage, to catch wild references.

si:dtp-external-value-cell-pointer
An "invisible pointer" used for external value cells, which are part of the closure mechanism (see page 102). and used by compiled code to address value and function cells.

si: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 113).

si: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.

si:dtp-one-q-forward
An "invisible pointer" used to indicate that the single cell containing it has been moved elsewhere.

si:dtp-gc-forward
This is used by the copying garbage collector to flag old objects that

have already been copied.

q-data-types *Variable*
The value of **q-data-types** is a list of all of the symbolic names for data types described above under **data-type**. (the symbols whose print names begin with "**dtp-**")

q-data-types *type-code*
An array, indexed by the internal numeric data-type code, which contains the corresponding symbolic names.

13.2 Creating Objects

make-list *area size*
This function makes a cdr-coded list of **nil**s of a specified length in a specified area. *area* is which area to create it in, which may be either a fixnum or a symbol whose value will be used. *size* is the number of words to be allocated. Each word has cdr code *cdr-next*, except for the last which has *cdr-nil*.

This function is to be used only for making lists. If making a "structure" (any data type that has a header), use one of the two functions below. This is because the two classes of object must be created in different storage regions, for the sake of system storage conventions and the garbage collector.

%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 are initialized as in **make-list**, currently.

%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.

structure-forward *old-object new-object*
This causes references to *old-object* to actually reference *new-object*, by storing invisible pointers in *old-object*. It returns *old-object*.

13.3 Pointer Manipulation

It should again be emphasized that improper use of these functions can destroy the Lisp environment, primarily because of interactions between the garbage collector and the illegal pointers that can be created by these sub-primitives.

%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*
This makes up a pointer, with *data-type* in the data-type field and *pointer* in the pointer field, and returns it. 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*
This returns a pointer with *data-type* in the data-type field, and *pointer* plus *offset* in the pointer field. 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.

%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.

%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, containing additional "unboxed Q's" following their "boxed Q's".

%structure-total-size *object*
Returns the total number of words occupied by the representation of *object*.

13.4 Special Memory Referencing

%store-conditional *pointer old new*
This is the basic locking primitive. *pointer* points 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 is not changed.

The following four functions are for I/O programming.

%unibus-read *address*
Returns the contents of the register at the specified Unibus address, as a fixnum. 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* will 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 a fixnum which is the low 24 bits of 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.

%xbus-write *io-offset data*
Writes the pointer field of *data*, which should be a fixnum, into the register at the specified Xbus address. The high eight bits of the word written are always zero. *io-offset* is an offset into the I/O portion of Xbus physical address space. This is guaranteed to write the location exactly once.

%p-contents-offset *base-pointer offset*
This 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.

%p-contents-as-locative *pointer*

Given a pointer to a memory location containing a pointer which isn't allowed to be "in the machine" (typically an invisible pointer) this function returns the contents of the location as a **dtp-locative**. I.e. it changes the disallowed data type to locative so that you can safely look at it and see what it points to.

%p-contents-as-locative-offset *base-pointer offset*

This 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*, fetches the contents of that location, and returns it with the data type changed to **dtp-locative** in case it was a type which isn't allowed to be "in the machine" (typically an invisible pointer). This is 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-store-contents *pointer value*

value is stored into the data-type and pointer fields of the location addressed by *pointer*. The cdr-code and flag-bit fields remain unchanged. *value* is returned.

%p-store-contents-offset *value base-pointer offset*

This 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 stores *value* into the data-type and pointer fields of that location. The cdr-code and flag-bit fields remain unchanged. *value* is returned.

%p-store-tag-and-pointer *pointer miscfields pntrfield*

Creates a *Q* by taking 8 bits from *miscfields* and 24 bits from *pntrfield*, and stores that into the location addressed by *pointer*. The low 5 bits of *miscfields* become the data-type, the next bit becomes the flag-bit, 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-ldb *ppss pointer*

This is like **ldb** but gets a byte from the location addressed by *pointer*. Note that you can load bytes out of the data type etc. bits, not just the pointer field, and that the word loaded out of need not be a fixnum. The result returned is always a fixnum, unlike **%p-contents** and friends.

%p-ldb-offset *ppss base-pointer offset*

This checks the cell pointed to by *base-pointer* for a forwarding pointer. Having followed forwarding pointers to the real structure pointed to, the byte specified by *ppss* is loaded from the contents of the location addressed by the forwarded *base-pointer* plus *offset*, and returned as a fixnum. This is the way to reference byte fields within a structure without violating system storage conventions.

%p-dpb *value ppss pointer*

The *value*, a fixnum, is stored into the byte selected by *ppss* in the word addressed by *pointer*. **nil** is returned. You can use this to alter data types, cdr codes, etc.

%p-dpb-offset *value ppss base-pointer offset*

This checks the cell pointed to by *base-pointer* for a forwarding pointer. Having followed forwarding pointers to the real structure pointed to, the *value* is stored into the byte specified by *ppss* in the location addressed by the forwarded *base-pointer* plus *offset*. **nil** is returned. This is the way to alter unboxed data within a structure without violating system storage conventions.

%p-mask-field *ppss pointer*

This is similar to **%p-ldb**, except that the selected byte is returned in its original position within the word instead of right-aligned.

%p-mask-field-offset *ppss base-pointer offset*

This is similar to **%p-ldb-offset**, except that the selected byte is returned in its original position within the word instead of right-aligned.

%p-deposit-field *value ppss pointer*

This is similar to **%p-dpb**, except that the selected byte is stored from the corresponding bits of *value* rather than the right-aligned bits.

%p-deposit-field-offset *value ppss base-pointer offset*

This is similar to **%p-dpb-offset**, except that the selected byte is stored from the corresponding bits of *value* rather than the right-aligned bits.

%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-flag-bit *pointer*

Extracts the flag-bit field of the contents of the location addressed by *pointer* and returns it as a fixnum.

%p-store-pointer *pointer value*
Clobbers the pointer field of the location addressed by *pointer* to *value*, and returns *value*.

%p-store-data-type *pointer value*
Clobbers the data-type field of the location addressed by *pointer* to *value*, and returns *value*.

%p-store-cdr-code *pointer value*
Clobbers the cdr-code field of the location addressed by *pointer* to *value*, and returns *value*.

%p-store-flag-bit *pointer value*
Clobbers the flag-bit field of the location addressed by *pointer* to *value*, 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 from 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".

bind *locative value*
[This will be renamed to **%bind** in the future.] 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 FEF that executed the **bind** instruction".

%halt
Stops the machine.

13.5 The Paging System

[Someday this will discuss how it works.]

si:%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.

si:%compute-page-hash *virtual-address*
This makes the hashing function for the page hash table available to the user.

si:%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.

si:%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.

si:%disk-restore *high-16-bits low-16-bits*
Loads virtual memory from the partition named by the catenation 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).

si:%disk-save *physical-mem-size high-16-bits low-16-bits*
Copies virtual memory into the partition named by the catenation 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**.

13.6 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.

%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. Currently the data-type is missing from this number.

default-cons-area *Variable*
The area number of the default area in which new data are to be consed. This is normally **working-storage-area**.

si:%initial-fef *Variable*
The function which is called when the machine starts up. Normally **si:lisp-top-level**.

%error-handler-stack-group *Variable*
The stack group which receives control when a microcode-detected error occurs. This stack group cleans up, signals the appropriate condition, or enters the debugger.

si:%current-stack-group *Variable*
The stack group which is currently running.

%initial-stack-group *Variable*
The stack group in which the machine starts up.

si:%current-stack-group-state *Variable*
The **sg-state** of the currently-running stack group.

si:%current-stack-group-previous-stack-group *Variable*
The resumer of the currently-running stack group.

si:%current-stack-group-calling-args-pointer *Variable*
The argument list of the currently-running stack group.

si:%current-stack-group-calling-args-number *Variable*
The number of arguments to the currently-running stack group.

si:%trap-micro-pc *Variable*
The microcode address of the most recent error trap.

si:%count-first-level-map-reloads *Variable*
The number of times the first-level virtual-memory map was invalid and had to be reloaded from the page hash table.

si:%count-second-level-map-reloads *Variable*
The number of times the second-level virtual-memory map was invalid and had to be reloaded from the page hash table.

si:%count-pdl-buffer-read-faults *Variable*
The number of read references to the pdl buffer which happened as virtual memory references which trapped.

si:%count-pdl-buffer-write-faults *Variable*
The number of read references to the pdl buffer which happened as virtual memory references which trapped.

si:%count-pdl-buffer-memory-faults *Variable*
The number of virtual memory references which 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.)

si:%count-disk-page-reads *Variable*
The number of pages read from the disk.

si:%count-disk-page-writes *Variable*
The number of pages written to the disk.

si:%count-disk-errors *Variable*
The number of recoverable disk errors.

si:%count-fresh-pages *Variable*
The number of fresh (newly-consed) pages created in core, which would have otherwise been read from the disk.

si:%aging-rate *Variable*
The number of age steps per disk read or write. This parameter controls how long a page must remain unreferenced before it is evicted from main memory.

si:%count-aged-pages *Variable*
The number of times the page ager set an age trap on a page, to determine whether it was being referenced.

si:%count-age-flushed-pages *Variable*
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.

%mar-low *Variable*
A fixnum which is the inclusive lower bound of the region of virtual memory subject to the MAR feature.

%mar-high *Variable*
A fixnum which is the inclusive upper bound of the region of virtual memory subject to the MAR feature.

%self *Variable*
The instance which has just been called. (See <not-yet-written>.)

%method-class *Variable*
The class in which the current method was found. (See <not-yet-written>.)

inhibit-scheduling-flag *Variable*
If non-**nil**, no process other than the current process can run.

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.

14. Areas

[Note: this chapter will be completely rewritten in the next edition of this manual, to reflect the existence of the garbage collector. The present chapter is very incomplete.]

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 his program, among other things. By putting related data together, locality can be greatly increased. Whenever a new object is created, for instance with **cons**, the area to be used can optionally be specified. There is a default Working Storage area which collects those objects which 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. All pointers out of a static area can be collected into an "exit vector", eliminating any need for the garbage collector to look at that area. As an important example, an English-language dictionary can be kept inside the Lisp without adversely affecting the speed of garbage collection. 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 will dispatch on an attribute 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.

Since the garbage collector is not yet implemented, the features mentioned in the previous two paragraphs are not either. Also, with the implementation of the garbage collector will come a new, more sophisticated area scheme. The two most visible effects of the new scheme will be that garbage will be collected, and that areas will be able to shrink and grow. When this happens, it will be documented; stay tuned. Most of this chapter will become inoperative at this time, so don't depend on it.

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.

The following variables hold the areas most often used:

default-cons-area *Variable*

The value of this variable is the number of the area to which all of the creators of conses (**cons**, **xcons**, **list**, **append**, etc.) use by default. It is initially the number of **working-storage-area**. Note that you can either bind this variable or use functions such as **cons-in-area** (see page 39) which take an area as an explicit argument.

default-array-area *Variable*

The value of this variable is the number of the area which **make-array** uses by default. It is initially the number of **working-storage-area**.

define-area *name size*

Create a new area whose name is the symbol *name*. The size of the area will be *size* words, rounded up to the nearest multiple of the machine page size. **define-area** fills in all of the area tables appropriately, and returns the number of the created area.

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. Regions will be explained later.

We will now list those areas with which the user may need to be concerned. This section will be expanded later.

area-name *Variable*

Indexed by area number. Contains the area's name (a symbol).

area-name

The function definition of **area-name** is an array of area names, indexed by area numbers.

working-storage-area *Variable*
This is the normal value of **default-cons-area** and **default-array-area**. Most working data are consed in this area.

permanent-storage-area *Variable*
This is to be used for "permanent" data, which will (almost) never become garbage. Unlike **woring-storage-area**, the contents of this area are not continually copied by the garbage collector.

sys:p-n-string *Variable*
Print names are stored here.

sys:nr-sym *Variable*
This contains most of the symbols in the Lisp world, except **t** and **nil**.

macro-compiled-program *Variable*
FEFs are put here by the compiler and by **fasload**.

15. The Compiler

15.1 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 will 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 format of FEFs and the instruction set are explained in <not-yet-written>.

There are three ways to invoke the compiler from the Lisp Machine. First, you may have an interpreted function in the Lisp environment which you would like to compile. The function **compile** is used to do this. Second, you may have code in an editor buffer which you would like to compile. The EINE 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 compiler can translate this file into a QFASL file. Loading in the QFASL file is like reading in the source file, except that the functions in the source file will be compiled. The **qc-file** function is used for translating source files into QFASL files.

15.2 How to Invoke the Compiler

compile *symbol*

symbol should be defined as an interpreted function (its definition should be a lambda-expression). The compiler converts the lambda-expression into a FEF, saves the lambda-expression as the **:previous-expr-definition** and **:previous-definition** properties of *symbol*, and changes *symbol*'s definition to be the FEF. (See **fset-carefully**, page 60. (Actually, if *symbol* is not defined as a lambda-expression, **compile** will try to find a lambda-expression in the **:previous-expr-definition** property of *symbol* and use that instead.)

uncompile *symbol*

If *symbol* is not defined as an interpreted function and it has a **:previous-expr-definition** property, then **uncompile** will restore the function cell from the value of the property. This "undoes" the effect of **compile**.

qc-file *filename* &optional *output-file load-flag in-core-flag package*

The file *filename* is given to the compiler, and the output of the compiler is written to a file whose name is *filename* except with an FN2 of "QFASL". The input format for files to the compiler is described on page 127. Macro definitions and **special** declarations created during the compilation will be undone when the compilation is finished.

The optional arguments allow certain modifications to this procedure. *output-file* lets you change where the output is written. *package* lets you specify in what package the source file is to be read. Normally the system knows, or asks interactively, and you need not supply this argument. *load-flag* and *in-core-flag* are incomprehensible; you don't want to use them.

qc-file-load *filename*
qc-file-load compiles a file and then loads it in.

See also the **disassemble** function (page 263), which lists the instructions of a compiled function in symbolic form.

The compiler can also be run in Maclisp on ITS. On the MIT-AI machine, type :LISPM1;QCMP. It will type out "READY" and leave you at a Maclisp top level. Then type (**qc-file** *filename*), expressing *filename* in Maclisp form.
Example:

```
(qc-file '((lispm) foo >))
```

15.3 Input to the Compiler

The purpose of **qc-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. **qc-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 S-expressions 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 which is not of a type known specially to the compiler, then that form will be 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 (**setq x 3**), then the compiler will put in the QFASL file instructions to set **x** to 3 at load time.

However, sometimes we want to put things in the file that are not merely meant to be translated into QFASL form. One such occasion is top level macro definitions; the macros must actually get defined within the compiler in order that the compiler be able to expand them at compile time. So when a macro form is seen, it should (sometimes) be evaluated at compile time, and should (sometimes) be put into the QFASL file.

Another thing we sometimes want to put in a file is compiler declarations. These are forms which should be evaluated at compile time to tell the compiler something. They should not be put into the QFASL file.

Therefore, a facility exists to allow the user to tell the compiler just what to do with a form. One might want a form to be:

Put into the QFASL file (translated), or not.

Evaluated within the compiler, or not.

Evaluated if the file is read directly into Lisp, or not.

Two forms are recognized by the compiler to allow this. The less general but Maclisp compatible one is **declare**; the completely general one is **eval-when.**

An **eval-when** form looks like

```
(eval-when times-list
           form1
           form2
           ...)
```

The *times-list* may contain any 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 will 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.

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.

A **declare** form looks like **(declare** *form1 form2* ...**)**. **declare** is defined in Lisp as a special form which does nothing; so the forms within a **declare** are not evaluated at **eval** time. The compiler does the following upon finding *form* within a **declare**: if *form* is a call to either **special** or **unspecial**, *form* is treated as **(load compile)**; otherwise it is treated as **(compile)**.

If a form is not enclosed in an **eval-when** nor a **declare**, then the times at which it will be evaluated depend on the form. The following table summarizes at what times evaluation will take place for any given form seen at top level by the compiler.

(eval-when *times-list form1* ...**)**
times-list

(declare (special ...)) or **(declare (unspecial ...))**
(load compile)

(declare *anything-else*)
(compile)

(special ...) or **(unspecial ...)**
(load compile eval)

(macro ...) or **(defstruct ...)**
(load compile eval)

(comment ...) Ignored

(begf ...) or **(endf ...)**
Ignored but may one day put something in the QFASL file.

(compiler-let ((*var val*) ...) *body...*)
At **(compile eval)** time, processes the body with the indicated variable bindings in effect. Does nothing at **load** time.

(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.**

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 (quote compile) 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 will all be evaluated, and the **(quote compile)** will harmlessly evaluate to the symbol **compile** and be ignored.)

eval-when *Special Form*

An **eval-when** form looks like

```
(eval-when times-list form1 form2 ...)
```

If one of the element of *times-list* is the symbol **eval**, then the *forms* are evaluated; otherwise **eval-when** does nothing.

But when seen by the compiler, this special form does the special things described above.

declare *Special Form*

declare does nothing, and returns the symbol **declare.**

But when seen by the compiler, this special form does the special things described above.

15.4 Compiler Declarations

This section describes functions meant to be called during compilation, and variables meant to be set or bound during compilation, by using **declare** or **local-declare.**

local-declare *Special Form*

A **local-declare** form looks like

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

Each *decl* is consed onto the list **local-declarations** while the *form*s are being evaluated (in the interpreter) or compiled (in the compiler). There are two uses for this. First, it can be used to pass information from outer macros to inner macros. Secondly, the compiler will specially interpret certain *decl*s as local declarations, which only apply to the compilations of the *form*s. It understands the following forms:

(special *var1 var2* ...**)**
: The variables *var1*, *var2*, etc. will be treated as special variables during the compilation of the *form*s.

(unspecial *var1 var2* ...**)**
: The variables *var1*, *var2*, etc. will be treated as local variables during the compilation of the *form*s.

(macro *name* **lambda** (x) *body***)**
: *name* will be defined as a macro during the compilation of the *form*s. Note that the **cddr** of this item is a function.

special *Special Form*

(special *var1 var2* ...**)** causes the variables to be declared to be "special" for the compiler.

unspecial *Special Form*

(unspecial *var1 var2* ...**)** removes any "special" declarations of the variables for the compiler.

The next three declarations are primarily for Maclisp compatibility.

***expr** *Special Form*

(*expr *sym1 sym2* ...**)** declares *sym1*, *sym2*, etc. to be names of functions. 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** *Special Form*

(***lexpr** *sym1 sym2 ...*) declares *sym1*, *sym2*, etc. to be names of functions. 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** *Special Form*

(***fexpr** *sym1 sym2 ...*) declares *sym1*, *sym2*, etc. to be names of special forms. In addition it prevents these names from appearing in the list of functions referenced but not defined printed at the end of the compilation.

There are some advertised variables whose compile-time values affect the operation of the compiler. The user may set these variables by including in his file forms such as

```
(declare (setq open-code-map-switch t))
```

run-in-maclisp-switch *Variable*

If this variable is non-**nil**, the compiler will try to warn the user about any constructs which will not work in Maclisp. By no means will all Lisp machine system functions not built in to Maclisp be cause for warnings; only those which could not be written by the user in Maclisp (for example, ***catch**, **make-array**, **value-cell-location**, etc.). Also, lambda-list keywords such as **&optional** and initialized **prog** variables will 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 will try 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**'d. 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 will attempt to produce inline code for the mapping functions (**mapc**, **mapcar**, etc., but not **mapatoms**) if the function being mapped is an anonymous lambda-expression. This allows that function to reference the local variables of the enclosing function without the need for special declarations. The generated code is also more efficient. The default value is T.

all-special-switch *Variable*

If this variable is non-**nil**, the compiler regards all variables as special, regardless of how they were declared. This provides full compatibility with the interpreter at the cost of efficiency. The default is **nil**.

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 and 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.

retain-variable-names-switch *Variable*

This controls whether the generated FEFs remember the names of the variables in the function; such information is useful for debugging (the **arglist** function uses it, see page 61), but it increases the size of the QFASL file and the FEFs created. The variable may be any of

nil No names are saved.

args Names of arguments are saved.

all Names of arguments and **&aux** variables are saved.

The default value of this symbol is **args**, and it should usually be left that way.

compiler-let *Macro*

(**compiler-let** ((*variable value*)...) *body*...), syntactically identical to **let**, allows compiler switches to be bound locally at compile time, during the processing of the *body* forms.

Example:

```
(compiler-let ((open-code-map-switch nil))
              (map (function (lambda (x) ...)) foo))
```

will prevent the compiler from open-coding the **map**. When interpreted, **compiler-let** is equivalent to **let**. This is so that global switches which affect the behavior of macro expanders can be bound locally.

inhibit-style-warnings *Macro*

(**inhibit-style-warnings** *form*) prevents the compiler from performing style-checking on the top level of *form*. Style-checking will still be 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)))
```

will not warn that **value-cell-location** will not work in Maclisp, but

```
(inhibit-style-warnings (setq bar (value-cell-location foo)))
```

will warn, since **inhibit-style-warnings** applies only to the top level of the form inside it (in this case, to the **setq**).

15.5 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 which 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 which returns the original form unchanged (and **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. This is somewhat like a Markov algorithm. Only after all the optimizers have been tried and have done nothing is an ordinary macro definition processed. This is so that the macro definitions, which will be seen by the interpreter, can be overridden for the compiler by an optimizer.

15.6 Files that Maclisp Must Compile

Certain programs are intended to be run both in Maclisp and in Lisp Machine Lisp. These files need some special conventions. For example, such Lisp Machine constructs as **&aux** and **&optional** must not be used. In addition, **eval-when** must not be used, since only the Lisp Machine compiler knows about it. All **special** declarations must be enclosed in **declare**s, so that the Maclisp compiler will see them. It is suggested that you turn on **run-in-maclisp-switch** in such files, which will warn you about a lot of bugs.

The macro-character combination "#Q" causes the object that follows it to be visible only when compiling for the Lisp Machine. The combination "#M" causes the following object to be visible only when compiling for Maclisp. These work only on subexpressions of the objects in the file, however. To conditionalize top-level objects, put the macros **if-for-lispm** and **if-for-maclisp** around them. (You can only put these around a single object.) The **if-for-lispm** macro turns off **run-in-maclisp-switch** within its object, preventing spurious warnings from the compiler. The **#Q** macro-character does not do this, since it can be used to conditionalize any S-expression, not just a top-level form.

There are actually three possible cases of compiling: you may be compiling on the Lisp Machine for the Lisp Machine; you may be compiling in Maclisp for the Lisp Machine (with :LISPM1;QCMP); or you may be compiling in Maclisp for Maclisp (with COMPLR). (You can't compile for Maclisp on the Lisp Machine because there isn't a Lisp Machine Lisp version of COMPLR.) To allow a file to detect any of these conditions it needs to, the

following macros are provided:

if-for-lispm *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 the Lisp Machine. If the Lisp Machine interpreter sees this it will evaluate *form* (the macro expands into *form*).

if-for-maclisp *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 Lisp Machine interpreter sees this it will ignore it (the macro expands into **nil**).

if-for-maclisp-else-lispm *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 *Macro*
On the Lisp Machine, (**if-in-lispm** *form*) causes *form* to be evaluated; in Maclisp, *form* is ignored.

if-in-maclisp *Macro*
In Maclisp, (**if-in-maclisp** *form*) causes *form* to be evaluated; on the Lisp Machine, *form* is ignored.

When you have two definitions of one function, one conditionalized for one machine and one for the other, indent the first "(**defun**" by one space, and the editor will put both function definitions together in the same file-section.

In order to make sure that those macros and macro-characters are defined when reading the file into the Maclisp compiler, you must make the file start with a prelude, which will have no effect when you compile on the real machine. The prelude can be found in "AI: LMDOC; .COMPL PRELUD"; this will also define most of the standard Lisp Machine macros and reader macros in Maclisp, including **defmacro** and the backquote facility.

Another useful facility is the form (**status feature lispm**), which evaluates to **t** when evaluated on the Lisp machine and to **nil** when evaluated in Maclisp.

16. Macros

16.1 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 one argument. **eval** applies the function to the form it was originally given. Then it takes whatever is returned, and evaluates that in lieu of the original form.

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

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

This thing is a macro: it is a *cons* whose *car* is the symbol **macro**. What happens if we try to evaluate a form (**first** '(**a b c**))? Well, **eval** sees that it has a list whose *car* is a symbol (namely, **first**), so it looks at the definition of the symbol and sees that it is a *cons* whose *car* is **macro**; the definition is a macro. **eval** takes the *cdr* of the cons, which is a lambda expression, and *applies* it to the original form that **eval** was handed. So it applies (**lambda** (x) (**list** 'car (**cadr** x))) to (**first** '(**a b c**)). 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 what the function returns. **eval** now evaluates this new form in place of the original form. (**car** '(**a b c**)) returns **a**, and so the result is that (**first** '(**a b c**)) returns **a**.

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

Macros are normally defined using the **macro** special form. A macro definition looks like this:

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

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

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

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

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

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

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

Of course, this macro is of limited usefulness. The reason is that the form in the *cadr* of the **increment** form had better be a symbol. If you tried (**increment** (**car** x)), it would be translated into (**setq** (**car** x) (**1+** (**car** x))), and **setq** would complain.

You can see from this discussion that macros are very different from functions. A function would not be able to tell what kind of subforms are around in a call to itself; they get evaluated before the functions 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; it 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 symbol 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)
  (cons 'do
        (cons (cadr x)
              (cons (caddr x)
                    (cons (list '1+ (cadr x))
                          (cons (list '> (cadr x) (cadddr x))
                                (cddddr x)))))))
```

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

16.2 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 would 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 "(**cadr** 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.

16.2.1 Defmacro

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

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

The (**var lower upper . body**) is a *pattern* to match against the body of the macro (to be more precise, to match against the *cdr* of the argument to the macro). **defmacro** tries to match the two lists

```
(var lower upper . body)
```

and

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

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

defmacro *Macro*

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

(**defmacro** *name pattern . body*)

The *pattern* may be anything made up out of symbols and conses. It is matched against the body of the macro form; both *pattern* and the form are *car*'ed and *cdr*'ed identically, and whenever a symbol is hit in *pattern*, the symbol is bound to the corresponding part of the form. All of the symbols in *pattern* can be used as

variables within *body*. *name* is the name of the macro to be defined. *body* is evaluated with these bindings in effect, and is returned to the evaluator.

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 it 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.

There is another version of **defmacro** which defines displacing macros (see page 141). **defmacro** has other, more complex features; see page 143.

16.2.2 Backquote

Now we deal with the other problem: the long strings of calls to **cons** and **list**. For this we must introduce some *reader macros*. Reader macros are not the same as normal macros, and they are not described in this chapter; see page 156.

The backquote facility is used by giving a backquote character ("←", ASCII code 140 octal), followed by a form. If the form does not contain any use of the comma macro-character, the form will simply be quoted. For example,

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

So in the simple cases, backquote is just like the regular single-quote macro. The way to get it to do interesting things is to include a use of the 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)
```

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

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.

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

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

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

If a comma inside a backquote form is followed by an "atsign" 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 are put into the list being created by the backquote. In other words, instead of generating a call to the **cons** function, backquote generates a call to **append**. For example, if **a** is bound to (x y z), then '(1 ,a 2) would evaluate to (1 (x y z) 2), but '(1 ,@a 2) would evaluate to '(1 x y z 2).

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

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

into

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

You could define the macro by

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

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. The following example illustrates the use of nested backquotes in the writing of macro-defining macros.

This example is a very simple version of **defstruct** (see page 147). You should first understand the basic description of **defstruct** before proceeding with this example. The **defstruct** below does not accept any options, and only allows the simplest kind of items; that is, it only allows forms like

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

We would like this form to expand into

```
(progn
 (defmacro item1 (x)
      `(aref ,x 1))
 (defmacro item2 (x)
      `(aref ,x 2))
 (defmacro item3 (x)
      `(aref ,x 3))
 (defmacro item4 (x)
      `(aref ,x 4))
 ...)
```

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)
          (cons 'progn (nreverse ans)))
       (setq ans
             (cons `(defmacro ,(car item-list) (x)
                              `(aref ,x ',i))
                   ans))))
```

The interesting part of this definition is the body of the (inner) **defmacro** form: **`(aref ,x ',i)**. Instead of using this backquote construction, we could have written **(list 'aref x ,i)**; that is, the "',i" acts like a comma which matches the outer backquote, while the "," preceeding 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.

16.3 Aids for Debugging Macros

mexp

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

16.4 Displacing Macros

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

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

displace *form expansion*

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

```
(si:displaced original-form expansion)
```

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

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

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

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

we would write

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

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

defmacro-displace *Macro*

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

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

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

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

16.5 Advanced Features of Defmacro

(To be supplied.)

(The basic matter is that you can use **&optional** and **&rest** with **defmacro**. The interactions between **&optional**'s initialization, and the fact that the "lambda-list" in **defmacro** can be arbitrary list structure are not clear. If you need to use this feature, try it out.)

16.6 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.

macroexpand-1 *form* &optional *compilerp*

If *form* is a macro form, this expands it (once) and returns the expanded form. Otherwise it just returns *form*. If **compilerp** is **t**, **macroexpand-1** will search the compiler's list of intenally defined macros (**sys:macrolist**) for a definition, as well as the function cell of the *car* of *form*. *compilerp* defaults to **nil**.

macroexpand *form* &optional *compilerp*

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*. *compilerp* has the same meaning as in **macroexpand-1**.

17. Defstruct

17.1 Introduction to Structure Macros

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 chapter 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.

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 5-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. 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. **defstruct** 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)
     ship-x-position
     ship-y-position
     ship-x-velocity
     ship-y-velocity
     ship-mass)
```

(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 macro which expands into an **aref** form; that is, **(ship-x-position foo)** would turn into **(aref foo 0)**. All of the "elements" are defined to refer to sequentially increasing elements of the array, e.g., **(ship-mass foo)** would turn into **(aref foo 4)**. So a ship is really implemented as an array, although that fact is kept hidden. These macros are called the ***accessor macros***, as they are used to access elements of the structure.

defstruct will also define **make-ship** to be a macro which expands into a call to **make-array** which will create an array of the right size (namely, 5 elements). So (**setq x (make-ship)**) will make a new ship, and x will be bound to it. This macro is called the *constructor macro*, because it constructs a new structure.

We also want to be able to change the contents of a structure. To do this, we use the **setf** macro (see page 146), as follows (for example):

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

Here x is bound to a ship, and after the evaluation of the **setf** form, the **ship-x-position** of that ship will be 100. The way this works is that the **setf** form expands into (**aset 100 x 0**); again, this is invisible to the programmer.

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 and as lists. (For array-leaders, the accessor macros expand into calls to **array-leader**, and for lists, to **car, cadr, caddr,** and so on.)

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

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

17.2 Setf and Locf

In Lisp, for each function to *access* (read) any piece of information, there is almost always a corresponding function to *update* (write) it as well. For example, **symeval** accesses a symbol's value cell, and **set** updates it. **array-leader** accesses the contents of an array leader element, and **store-array-leader** updates it. The knowledge of how these functions correspond is accessible through a macro called **setf**.

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 *Macro*

setf takes a form which *accesses* something, and "inverts" it to produce a corresponding form to *update* the thing. The form for **setf** is

(**setf** *access-form value*)

It expands into an update form, which stores the result of evaluating the form *value* into the place referenced by the *access-form*.

Examples:

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

locf *Macro*

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

(**locf** *access-form*)

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)
```

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

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

As a special case, **setf** and **locf** allow a variable as the reference. In this case they turn into **setq** and **value-cell-location**, respectively.

For the sake of efficiency, the code produced by **setf** and **locf** does not preserve order of evaluation of the argument forms. This is only a problem is the argument forms have interacting side-effects. In addition, the value produced by **setf** is dependant on the structure type and is not guaranteed; **setf** should be used for side effect only.

17.3 How to Use Defstruct

defstruct *Macro*

A call to **defstruct** looks like:

```
(defstruct (name option-1 option-2 ...)
           item-1
           item-2
           ...)
```

name must be a symbol; it is the name of the structure. It is used for many different things, explained below.

option-n may be either a symbol (which should be one of the recognized option names, listed below) or a list (whose *car* should be one of the option names and the rest of which should be "arguments" to the option).

item-n may be in any of three forms:

(1) *item-name*

(2) (*item-name default-init*)

(3) ((*item-name-1 byte-spec-1 default-init-1*)
(*item-name-2 byte-spec-2 default-init-2*)
...)

item-name must always be a symbol, and each *item-name* is defined as an access macro. Each *item* allocates one entry of the physical structure, even though in form (3) several access macros are defined.

In form (1), *item-name* is simply defined as a macro to return the corresponding element of the structure. The constructor macro will initialize that entry to **nil** (or **0** in a numeric array) by default. In form (2), the access macro is defined the same way, but the default initialization is provided by the user of **defstruct**.

In form (3), several access macros are defined, and each one refers to the single structure element allocated for this *item*. However, if *byte-spec* is a fixnum, the access macros will **ldb** that byte from the entry (see the function **ldb**, page 77). *byte-spec* may also be **nil**, in which case the usual form of access macro is defined, returning the entire entry in the structure. Note that it is possible to define two or more different overlapping byte fields. (If more than one of these has a *default-init* the results of initializing the entry are undefined and unpredictable.) For example, if the third *item* of a call to **defstruct** were

```
((foo-high-byte 1010)
 (foo-low-byte 0010)
 (foo-whole-thing nil))
```

then (**foo-high-byte foo**) would expand to (**ldb 1010 (aref foo 2)**), and (**foo-whole-thing foo**) would expand to (**aref foo 2**).

Form (3) can also be used if you want to have an element with more than one access macro. By putting ((**foo nil**) (**bar nil**)), both **foo** and **bar** will be defined identically.

17.4 Options to Defstruct

Note that options which take no arguments may be given as just a symbol, instead of a list.

:array The structure should be implemented as an array. This is the default. (No arguments.)

:array-leader The structure should implemented as be an array-leader. (No arguments.)

:list The structure should be implemented as a list. (No arguments.)

:grouped-array
See page 150.

:times Used by grouped arrays. See page 150.

:size Takes one argument, a symbol. The symbol gets *set* to the size of the structure, at load-time (not compile-time).

:size-macro One argument, a symbol. The symbol gets defined as a macro, which expands into the size of the structure.

:constructor One argument, a symbol which will be the name of the constructor macro. If the option is not present, the name of the constructor will be made by concatenating "**make-**" with the *name* of the structure. If the argument is **nil**, do not define any constructor macro.

:named-structure
One optional argument. If present, the argument is the named structure symbol. If not, the named structure symbol will be the *name* of the structure. This causes the constructor to create named structure arrays (and thus may not be used with the **:list** option) and automatically allocate the appropriate slot in the structure and put the symbol there.

:default-pointer
One argument. The access macros will be defined in such a way that if they are called on no "arguments", the argument to the **:default-pointer** option will be used instead. (Normally, access macros will signal an error if their "argument" is missing.)

:make-array One argument, arguments to the **make-array** function. See below.

:include See page 150.

17.5 Using the Constructor Macro

If the argument to the **:constructor** option is **nil**, no constructor macro is defined. But otherwise, **defstruct** creates a constructor macro, which will create an instance of the structure. This section explains how the constructor macro interprets its "arguments".

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

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

Each *symbol* may be either a name of an *item* of the structure, or a specially recognized keyword. All *form*s are evaluated.

If *symbol* is the name of an *item*, then that element of the created structure will be initialized to the value of *form*. If no *symbol* is present for a given item, then the item will be initialized in accordance with the default initialization specified in the call to **defstruct**. If the **defstruct** itself also did not specify any initialization, the element will be initialized to **nil**, unless the structure is implemented by a *numeric array*, in which case it will be initialized to **0**. (In other words, the initialization specified to the constructor overrides the initialization specified to **defstruct**.)

There are two symbols which are specially recognized by the constructor. One is **:make-array**, which should only be used for *array* and *array-leader* type structures. The value of *form* is used as the argument list to the **make-array** function call created by the constructor. This way, you can specify the area in which you wish the structure to be created, the type of the array to be created, and so on. Of course, if you provided *all* of the arguments to **make-array**, the constructor would not be able to do its job; so the constructor overrides your specifications of certain elements. If the structure is *array* type, your specification of the array's dimensions (the third argument to **make-array**) is ignored; if it is of *array-leader* type, the array-leader argument (the fifth argument to **make-array**) is ignored. Also, in both cases the named-structure argument (the seventh argument to **make-array**) is ignored. They are ignored because it is the constructor macro's job to fill them in. If the list you provide is shorter than the number of arguments to **make-array**, it will be as if you had given the missing elements as **nil**. Similarly, if your list is too long, the extra elements will be ignored. If you do not provide the **:make-array** keyword at all, the arguments default from the value of the **:make-array** option to **defstruct**. If you did not even provide that, the default argument lists are:

For *array*s: **(default-array-area 'art-q** *whatever* **nil nil nil** *whatever***)**

For *array-leader*s:
(default-array-area 'art-q 0 nil *whatever* **nil** *whatever***)**

The second keyword recognized by the constructor is **:times**, which should only be used for **grouped-arrays**. Its value is the number of repetitions of the structure in the grouped-array. If **:times** is not provided, it defaults from the **:times** option of **defstruct**.

If you did not even provide that, the default is 1.

17.6 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, and requires that the structure be implemented as an array, that it not have any **:include** option, and that it not be a named structure.

The accessor macros are defined to take a "first argument" which should be a fixnum, and is the index into the array of where this instance of the structure starts. It should be a multiple of the size of the structure, for things to make sense.

Note that the "size" of the structure (as given in the **:size** symbol and the **:size-macro)** 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 creator macro is given as the argument to the **:times** or **:grouped-array** option, or the **:times** keyword of the constructor macro (see below).

17.7 The :include Option.

(To be supplied)

18. The I/O System

The Lisp Machine provides a powerful and flexible system for performing input and output to peripheral devices. To allow device independent I/O (that is, to allow programs to be written in a general way so that the program's input and output may be connected with any device), the Lisp Machine I/O system provides the concept of an "I/O stream". What streams are, the way they work, and the functions to create and manipulate streams, are described in this chapter. This chapter also describes the Lisp "I/O" operations **read** and **print**, and the printed representation they use for Lisp objects.

18.1 The Character Set

The Lisp Machine normally represents characters as fixnums. The mapping between these numbers and the characters is listed here. The mapping is similar to ASCII, but somewhat modified to allow the use of the so-called SAIL extended graphics, while avoiding certain ambiguities present in ITS. For a long time ITS treated the Backspace, Control-H, and Lambda keys on the keyboard identically as character code 10 octal; this problem is avoided from the start in the Lisp Machine's mapping.

Fundamental characters are eight bits wide. Those less than 200 octal (with the 200 bit off) and only those 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.)

Characters in the range of 200 to 237 inclusive are used for special characters. Character 200 is a "null character", and is not used for anything much. Characters 201 to 215 correspond to the special keys on the keyboard such as Form and Call. The rest of this group is reserved for future expansion.

The remaining characters are used for control operations. The characters 240 to 247 inclusively mean "Switch to font 0", "Switch to font 1", etc. The rest of this group is reserved for future expansion.

In some contexts, a fixnum can hold both a character code and a font number for that character. The following byte specifiers are defined:

%%ch-char *Variable*

The value of **%%ch-char** is a byte specifier for the field of a fixnum character which holds the character code.

%%ch-font *Variable*
The value of **%%ch-font** is a byte specifier for the field of a fixnum character which holds the font number.

Characters read in from the keyboard include a character code and the Control and Meta bits. The following byte specifiers are provided:

%%kbd-char *Variable*
The value of **%%kbd-char** is a byte specifier for the field of a keyboard character which holds the normal eight-bit character code.

%%kbd-control *Variable*
The value of **%%kbd-char** is a byte specifier for the bit of a keyboard character which is 1 if either Control key was held down.

%%kbd-meta *Variable*
The value of **%%kbd-char** is a byte specifier for the field of a keyboard character which is 1 if either Meta key was held down.

%%kbd-control-meta *Variable*
The value of **%%kbd-char** is a byte specifier for the two-bit field of a keyboard character whose low bit is the Control bit, and whose high bit is the Meta bit.

%%kbd-mouse *Variable*
The value of **%%kbd-mouse** is a byte specifier for the bit in a keyboard character which indicates that the character is not really a character, but a signal from the mouse.

%%kbd-mouse-button *Variable*
The value of **%%kbd-mouse-button** is a byte specifier for the field in a mouse signal which says which button was clicked. The value is **0**, **1**, or **2** for the left, middle, and right buttons, respectively.

%%kbd-mouse-n-clicks *Variable*
The value of **%%kbd-mouse-n-clicks** is a byte specifier for the field in a mouse signal which says how many times the button was clicked. The value is one less than the number of times the button was clicked.

Since the Control and Meta bits are not part of the fundamental 8-bit character codes, there is no way to express keyboard input in terms of simple character codes. However, there is a convention which many programs accept for encoding keyboard input into character codes: if a character has its Control bit on, prefix it with an Alpha; if a character has its Meta bit on, prefix it with a Beta; if a character has both its Control and Meta bits on, prefix it with an Epsilon. To get an Alpha, Beta, Epsilon, or Equivalence into the string, quote it by prefixing it with an Equivalence.

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 bs	240 switch to font 0
201 break	211 tab	241 switch to font 1
202 clear	212 line	242 switch to font 2
203 call	213 vt	243 switch to font 3
204 escape (NOT altmode!)	214 form	244 switch to font 4
205 backnext	215 return	245 switch to font 5
206 help		246 switch to font 6
207 rubout		247 switch to font 7

216-237 reserved for future special keys
250-377 reserved for future control operations

The Lisp Machine Character Set

18.2 Printed Representation

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 its subfunctions are known as the *reader*.

This section describes in detail what the printed representation is for any Lisp object, and just what **read** does. For the rest of the chapter, the phrase "printed representation" will be abbreviated as "p.r.".

18.2.1 What the Printer Produces

The printed representation of an object depends on its type. In this section, we will consider each type of object and explain how it is printed.

Printing is done either with or without *slashification*. The non-slashified version is nicer looking in general, but if you give it to **read** it won't do the right thing. The slashified version is carefully set up so that **read** will be able to read it in. The primary effects of slashification are that special characters used with other than their normal meanings (e.g., a parenthesis appearing in the name of a symbol) are preceeded by slashes or cause the name of the symbol to be enclosed in vertical bars, and that symbols which are not from the current package get printed out with their package prefixes (a package prefix looks like a string followed by a colon).

For a fixnum: if the fixnum is negative, the printed representation begins with a minus sign ("-"). Then, the value of the variable **base** is examined. If **base** is a positive fixnum, the number is printed out in that base (**base** defaults to 8); if it is a symbol with a **:princ-function** property, the value of the property will be funcalled on two arguments: **minus** of the fixnum to be printed, and the stream to which to print it; otherwise the value of **base** is invalid. This is a hook to allow output in Roman numerals and the like. Finally, if **base** equals 10. and the variable ***nopoint** is **nil**, a decimal point is printed out. Slashification does not affect the printing of fixnums.

base *Variable*

The value of **base** is a number which is the radix in which fixnums are printed, or a symbol with a **:princ-function** property. The initial value of **base** is 8.

***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 **ibase** is not 10. at the time of reading. If ***nopoint** is non-**nil**, the trailing decimal points are suppressed. The initial value of ***nopoint** is **nil**.

For a symbol: if slashification is off, the p.r. is simply the successive characters of the print-name of the symbol. If slashification 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 section on packages (page 176) for an explanation of the package name prefix. Secondly, if the p.r. would not read in as a symbol at all (that is, if the print looks like a number, or contains special characters), then the p.r. must have some quoting for those characters, either by the use of slashes ("/") before each special character, or by the use of vertical bars ("|") around the whole name. The decision whether quoting is required is done using the readtable, so it is always accurate provided that **readtable** has the same value when the output is read back in as when it was printed.

For a string: if slashification is off, the p.r. is simply the successive characters of the string. If slashification is on, the string is printed between double quotes, and any characters inside the string which need to be preceeded by slashes will be. Normally these are just double-quote and slash. Incompatibly with Maclisp, carriage return is *not* ignored inside strings and vertical bars.

For an array which is a named structure: if the named structure has a named structure symbol which is defined as a function (which it always ought to), then that function is called on four arguments: the symbol **:print**, the object itself, the current *depth* of list structure (see below), and whether slashification is enabled. A suitable printed representation should be sent to the value of **standard-output**, which the printer lambda-binds to the correct stream. This allows a user to define his own p.r. for his named structures; examples can be found in the named structure section (see page 91). If the named structure is not "well formed" (if the symbol is undefined or not present), it is handled as if it were not a named structure, as follows.

Other arrays: the p.r. starts with a number sign and a less-than sign. 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 greater-than sign.

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.

Thus, the usual printed representations such as **(a b (foo bar) c)** are printed.

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 **prinlength**, **print** will terminate 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 **prinlevel**, then the list will be printed as "**". These two features allow a kind of abbreviated printing which is more concise and suppresses detail. Of course, neither the ellipsis nor the "**" can be interpreted by **read**, since the relevant information is lost.

prinlevel *Variable*

prinlevel can be set to the maximum number of nested lists that can be printed before the printer will give up and just print a "**". If it is **nil**, which it is initially, any number of nested lists can be printed. Otherwise, the value of **prinlevel** must be a fixnum.

prinlength *Variable*

prinlength can be set to the maximum number of elements of a list that will be printed before the printer will give up and print a "...". If it is **nil**, which it is initially, any length list may be printed. Otherwise, the value of **prinlength** must be a fixnum.

For any other data type: the p.r. starts with a number sign and a less-than sign ("<"), the "**dtp-**" symbol for this datatype, a space, and the machine address of the object. Then, if the object is a microcoded function, compiled function, or stack group, its name is printed. Finally a greater-than sign (">") is printed.

None of the p.r.'s beginning with a number sign can be read back in, nor, in general, can anything produced by named structure functions. Just what **read** accepts is the topic of the next section.

18.2.2 What The Reader Accepts

The purpose of the reader is to accept characters, interpret them as the p.r. of a Lisp object, and return a corresponding Lisp object. The reader cannot accept everything that the printer produces; for example, the p.r.'s of arrays (other than strings), compiled code objects, closures, stack groups etc. cannot be read in. However, it has many features which 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 reader understands the p.r's of fixnums in a way more general than is employed by the printer. Here is a complete description of the format for fixnums.

Let a *simple fixnum* be a string of digits, optionally preceeded by a plus sign or a minus sign, and optionally followed by a trailing decimal point. A simple fixnum will be interpreted by **read** as a fixnum. If the trailing decimal point is present, the digits will be interpreted in decimal radix; otherwise, they will be considered as a number whose radix is the value of the variable **ibase**.

ibase *Variable*

The value of **ibase** is a number which is the radix in which fixnums are read. The initial value of **ibase** is 8.

read will also understand a simple fixnum, followed by an underscore ("_") or a circumflex ("^"), followed by another simple fixnum. The two simple fixnums will be interpreted in the usual way, then the character in between indicates an operation to be performed on the two fixnums. The underscore indicates a binary "left shift"; that is, the fixnum to its left is doubled the number of times indicated by the fixnum to its right. The circumflex multiplies the fixnum to its left by **ibase** the number of times indicated by the fixnum to its right. Examples: **645_6** means **64500** (in octal) and **645^3** means **645000**.

Here are some examples of valid representations of fixnums to be given to **read**:

```
4
23456.
-546
+45^+6
2_11
```

A string of letters, numbers, and "extended alphabetic" characters is recognized by the reader as a symbol, provided it cannot be interpreted as a number. When the reader sees one, it *interns* it on a *package* (see page 176 for an explanation of interning and the package system). Symbols may start with digits; you could even have one named "-345T"; **read** will accept 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, put a slash before the strange characters. If you want to have a symbol whose print-name looks like a number, put a slash before some character in the name. You can also enclose the name of a symbol in vertical bars, which quotes all characters inside, except vertical bars and slashes, which must be quoted with slash.
Examples of symbols:

```
foo
bar/(baz/)
34w23
|Frob Sale|
```

The reader will also recognize strings, which should be surrounded by double-quotes. If you want to put a double-quote or a slash inside a string, preceed it by a slash.

Examples of strings:

```
"This is a typical string."
"That is known as a /"cons cell/" in Lisp."
```

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. Any of the following are valid:

```
(foo . bar)
(foo bar baz)
(foo . (bar . (baz . 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 exactly the same as 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**.

Whenever the reader sees any of the above, it creates new cons cells; it never returns existing list structure. This contrasts with the case for symbols, as very often **read** returns symbols that it found on the package rather than creating new symbols itself. Symbols are the only thing that work this way.

The dot that separates the 2 elements of a dotted-pair p.r. for a cons is only recognized if it is surrounded by delimiters. Thus dot may be freely used within print-names of symbols and within numbers.

18.2.3 Sharp-sign Abbreviations

The reader's syntax includes several abbreviations introduced by sharp sign (**#**). These take the general form of a sharp sign, a second character which identifies the syntax, and following arguments. Here are the currently-defined sharp-sign constructs; more are likely to be added in the future.

#/ **#/***x* reads in as the number which is the character code for the character *x*. For example, **#/a** is equivalent to **141** but clearer in its intent. This is the recommended way to include character constants in your code. Note that the slash causes this construct to be parsed correctly by the editors, Emacs and Eine.

**#***name* reads in as the number which is the character code for the non-printing character symbolized by *name*. (In the Lisp-machine compatible Maclisp environment, Lisp-machine character code is used if the file is being compiled for the Lisp machine, or ascii character code if the result is intended to be used in Maclisp.)

The following character names are recognized: **brk, clr, call, esc, back-next, help, rubout, bs, tab, lf, vt, ff, cr, sp**. These are generally self-explanatory; **cr** is the key marked return, **sp** is space, **ff** is the key marked form.

#' #'*foo* is an abbreviation for (**function** *foo*). *foo* is the p.r. of any object.

#, #,*foo* evaluates *foo* (the p.r. of a Lisp form) at read time, unless the compiler is doing the reading, in which case 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 which cannot be written with **quote**. Note that the reader does not put **quote** in front of the result of the evaluation. You must do this yourself if you want it, typically by using the ' macro-character.

#Q #Q*foo* reads as *foo* if the input is being read by the Lisp machine or being compiled to run on the Lisp machine, otherwise it reads as nothing (white space).

#M #M*foo* reads as *foo* if the input is being read into Maclisp or compiled to run in Maclisp, otherwise it reads as nothing (white space).

This is an obsolete form of #/. You write ## followed by a space, followed by the character whose character code you want.

18.2.4 The Readtable

(To be supplied.)

18.2.5 Reader Macros

(To be supplied.)

18.3 Input Functions

read &optional *stream eof-option*

read reads in the printed representation of a Lisp object from *stream*, builds a corresponding Lisp object, and returns the object. If the end-of-file is reached before a valid object starts, it returns *eof-option* instead. If the end-of-file is reached in the middle of an object, e.g. inside parentheses, an error is signalled. *stream* defaults to the value of **standard-input**, and *eof-option* defaults to **nil**.

In order to allow compatibility with Maclisp, the arguments are interpreted in a slightly more complicated way. If the *stream* is **nil**, then the value of **standard-input** is used. If *stream* is **t**, the value of **terminal-io** is used.

Maclisp allows the two arguments to be interchanged, but the Lisp machine does not.

There is an array called the readtable (see page 159) which is used to control the reader. It contains information about the syntax of each character. Initially it is set up to give the standard Lisp meanings to all the characters, but the user can change them to make

the reader usable as a lexical analyzer for a wide variety of input formats or languages. It is also possible to have several readtables describing different syntax and to switch from one to another by binding the symbol **readtable**.

The format of the readtable is not yet documented herein.

readtable *Variable*
The value of **readtable** is the current readtable.

tyi &optional *stream eof-option*
tyi inputs one character from *stream* and returns it. The arguments are the same as for **read**.

readline &optional *stream eof-option*
readline reads in a line of text, terminated by a newline. It returns the line as a character string, *without* the newline character. This function is usually used to get a line of input from the user. The arguments are the same as for **read**.

readch &optional *stream eof-option*
readch is just like **tyi**, except that instead of returning a fixnum character, it returns a symbol whose print name is the character read in. This is just like a Maclisp "character object". The symbol is interned, on the **user** package. The arguments are the same as for **read**.

tyipeek &optional *peek-type stream eof-option*
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 Lisp Machine reader are quite different.

The *stream* and *eof-option* arguments are the same as for **read**.

Note that all of these functions will echo their input if used on an interactive stream (one which supports the **:rubout-handler** operation; see below.) The functions that input more than one character at a time (**read**, **readline**) allow the input to be edited using

rubout. **tyipeek** echoes all of the characters that were skipped over if **tyi** would have echoed them; the character not removed from the stream is not echoed either.

readlist *char-list*
char-list is a list of characters. The characters may be represented by anything that the function **character** accepts: 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 will all take effect.

read-from-string *string*
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 will all take effect.
Example:

```
(read-from-string "(a b c)") => (a b c)
```

18.4 Output Functions

prin1 *x* &optional *stream*
prin1 outputs the printed representation of *x* to *stream*, with slashification (see page 154). *stream* defaults to the value of **standard-output**. If *stream* is **nil**, the value of **standard-output** is used. If it is **t**, the value of **terminal-io** is used. If it is a list of streams, then the output is performed to all of the streams (this is not implemented yet).

prin1-then-space *x* &optional *stream*
prin1-then-space is like **prin1** except that output is followed by a space.

print *x* &optional *stream*
print is just like **prin1** except that output is preceeded by a newline and followed by a space.

princ *x* &optional *stream*
princ is just like **prin1** except that the output is not slashified.

tyo *char* &optional *stream*
tyo outputs the character *char* to *stream*. The *stream* argument is the same as for **prin1**.

terpri &optional *stream*
terpri outputs a newline character to **stream**. The *stream* argument is the same as for **prin1**.

The **format** function (see page 85) 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.

The **grindef** function is useful for formatting Lisp programs. See <not-yet-written>.

cursorpos &optional *arg1 arg2*

This function exists primarily for Maclisp compatibility. Usually it is preferable to call the TV routines directly. **cursorpos** only operates on **console-io-pc-ppr** and does not work if a different font than the default is being used. The Maclisp Newio feature where one of the arguments to **cursorpos** can be a file is not supported.

(**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.

- F Moves one space to the right.
- B Moves one space to the left.
- D Moves one line down.
- U Moves one line up.
- C Clears the piece of paper.
- T Homes up (moves to the top left corner).
- E Clear from the cursor to the end of the piece of paper.
- L Clear from the cursor to the end of the line.
- K Clear the character position at the cursor.
- X B then K.
- Z Home down (moves to the bottom left corner).

exploden *x*

exploden returns a list of characters (as fixnums) which are the characters that would be typed out by (**princ** *x*) (i.e. the unslashified printed representation of *x*).
Example:

```
(exploden '(+ /12 3)) => (50 53 40 61 62 40 63 51)
```

explodec *x*

explodec returns a list of characters represented by character objects which are the characters that would be typed out by (**princ** *x*) (i.e. the unslashified printed representation of *x*).
Example:

```
(explodec '(+ /12 3)) => ( /( + /  /1 /2 /  /3 /) )
```

(Note that there are slashified spaces in the above list.)

explode *x*

explode returns a list of characters represented by character objects which are the characters that would be typed out by (**prinl** *x*) (i.e. the slashified printed representation of *x*).

Example:

```
(explode '(+ /12 3)) => ( /( + /  // /1 /2 /  /3 /) )
```

(Note that there are slashified spaces in the above list.)

flatsize *x*

flatsize returns the number of characters in the slashified printed representation of *x*.

flatc *x*

flatc returns the number of characters in the unslashified printed representation of *x*.

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)** will print 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** will use those operations instead of **:tyi** and **:tyo**, for greater efficiency. *leader-size* will be passed as the argument to the **:line-out** operation.

18.5 I/O Streams

18.5.1 What Streams Are

Many programs accept input characters and produce output characters. The method for performing input and output to one device is very different from the method for some other device. We would like our programs to be able to use any device available, but without each program having to know about each device.

In order to solve this problem, we introduce the concept of a *stream*. A stream is a source and/or sink of characters. 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 to 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.

A stream is a functional object; that is, it is something that you can apply to arguments. The first argument given to a stream is a symbol which is the name of the operation you wish to perform. The rest of the arguments depend on what operation you are doing.

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 which the stream may not support by itself, but will work anyway, albeit slowly, because the "stream default handler" can handle them. If you have a stream, there is an operation called **:which-operations** that will return a list of the names of all of the operations that are supported "natively" by the stream. *All* streams support **:which-operations**, and so it isn't in the list itself.

18.5.2 General Purpose Stream Operations

Here are some simple operations. Listed are the name of the operation, what arguments it takes, and what it does.

:tyo Takes one argument, which is a character. The stream will output that character. For example, if **s** is bound to a stream, then the form

```
(funcall s ':tyo 102)
```

will output a "B" to the stream.

:tyi Takes one optional argument, described later. The stream will input one character and return it. For example, if the next character to be read in by the stream is a "C", then the form

```
(funcall s ':tyi)
```

will return 103. Note that the **:tyi** operation will not "echo" the character in any fashion; it just does the input. The **tyi** function (see page 160) will do echoing when reading from the terminal. The 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 will return **nil** at the end of file. Otherwise it will signal an error, and print out the argument as the error message.

:untyi Takes one argument, which is a character. The stream will remember that character, and the next time a character is input, it will return the saved character. In other words, **:untyi** means "stuff this character back into the input source". For example,

```
(funcall s ':untyi 120)
(funcall s ':tyi) ==> 120
```

This operation is used by **read**, and any stream which supports **:tyi** must support **:untyi** as well. Note that you are only allowed to **:untyi** one character before doing a **:tyi**, and you aren't allowed to **:untyi** a different character than the last character you read from the stream. Some streams implement **:untyi** by saving the character, while others implement it by backing up the pointer to a buffer.

:which-operations

Takes no arguments. It returns a list of the operations supported "natively" by the stream.

Example:

```
(funcall s ':which-operations)
   ==> (:tyi :tyo :untyi :line-out :listen)
```

Any stream must either support **:tyo**, or support both **:tyi** and **:untyi**. There are several other, more advanced input and output operations which will work on any stream that can do input or output (respectively). Some streams support these operations themselves; you can tell by looking at the list returned by the **:which-operations** operation. Others will be handled by the "stream default handler" even if the stream does not know about the operation itself. However, in order for the default handler to do one of the more advanced output operations, the stream must support **:tyo**, and for the input operations the stream must support **:tyi** (and **:untyi**).

Here is the list of such operations:

:listen 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 **nil**, by virtue of the default handler. The main purpose of **:listen** is to test whether the user has hit a key, perhaps trying to stop a program in progress.

:fresh-line An output operation which takes no arguments. It 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 will do nothing, otherwise it will output a newline. For streams which don't support this, the default handler will always output a newline.

:string-out An output operation which takes one required argument, a string to output. The characters of the string are successively output to the stream. This operation is provided for two reasons; first, it saves the writing of a loop

which is used very often, and second, some streams can perform this operation much more efficiently than the equivalent **:tyo** operations. The **:string-out** operation also takes two optional arguments, which are a range of characters withing the string to output; the second argument is the index of the first character to output (defaulting to **0**), and the third is one greater than the index of the last character to 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. If the stream doesn't support **:string-out** itself, the default hander will turn it into a bunch of **:tyo**s.

:line-out An output operation which takes one argument, a string. The characters of the string, followed by a newline character, are output to the stream. If the stream doesn't support **:line-out** itself, the default hander will turn it into a bunch of **:tyo**s.

:line-in An input operation which takes one argument. The stream should input one line from the input source, and return it as a string with the newline character stripped off. Many streams will have a string which is used as a buffer for lines. If this string itself is 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 argument to **:line-in**. If the argument is **nil**, the stream will 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 the argument is **t**, the stream will make a copy. If the argument is a fixnum, *n*, then the stream will make a copy with an array leader *n* 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 the end-of-file while reading in characters, it will return the characters it has read in as a string, and return 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 newline in the input source.

:clear Takes no arguments. The stream will clear any buffered input or output. If the stream does not handle this, the default handler will ignore it. [To be renamed to **:clear-input** and **:clear-output**.]

:finish Takes no arguments. It returns when the currently pending I/O operation is completed. It does not do anything itself; it is just used to await completion of an operation. If the stream does not handle this, the default handler will ignore it.

:force-output
Takes no arguments. It causes any buffered output to be sent to the device.

If the stream does not handle this, the default handler will ignore it.

:close Takes no arguments. It causes the stream to be "closed", and no further operations should be performed on it. However, it is all right to **:close** a closed stream. If the stream does not handle **:close**, the default handler will ignore it.

:tyi-no-hang Just like **:tyi** except that if it would be neccesary to wait in order to get the character, returns **nil** instead. This lets the caller efficiently check for input being availible and get the input if there is any.

18.5.3 Special Purpose Stream Operations

There are several other defined operations which the default handler cannot deal with; if the stream does not support the operation itself, then an attempt to use it will signal an error. These are:

:read-pointer This is supported by the file stream (see page 171). It takes no arguments, and returns the position that the stream is up to in the file, as a number of characters.

:name This is supported by the file stream. It returns the name of the file open on the stream, as a string.

:rubout-handler
This is supported by interactive streams such as the **tv-terminal-stream**, and is described in its own section below (see page 173).

:untyo-mark This is used by **grind** if the output stream supports it. It takes no arguments. The stream should return some object which indicates where output has gotten up to in the stream.

:untyo This is used by **grind** 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.

:get-unique-id
This is supported by the file stream. It returns a string which identifies the file which is open, including its full name, its length, and its creation date.

:read-cursorpos
This operation is supported by piece-of-paper streams (see **tv-make-stream**, page 234). It returns two values: the current *x* and *y* positions of the cursor. It takes one argument, which is a symbol indicating in what units *x* and *y* should be; the symbols **:pixel** and **:character** are understood. This operation, and **:set-cursorpos**, are used by the **format** "~T" request (see page 86), which is why "~T" doesn't work on all streams. Any stream that supports this operation must support **:set-cursorpos** as well.

:set-cursorpos

This operation is supported by the same streams that support **:read-cursorpos**. It sets the position of the cursor. It takes three arguments: a symbol indicating the units (just like **:read-cursorpos**), the new *x* position, and the new *y* position.

18.5.4 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 which 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**.

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 which defaults to **standard-input**.

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 which defaults to **standard-output**.

error-output *Variable*

The value of **error-output** is a stream to which error messages should be sent. 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. [This seems not be used by things which ought to use it.]

query-io *Variable*

The value of **query-io** is a stream which 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 elsewhere (usually directly to the user). [This seems not be used by things which ought to use it.]

terminal-io *Variable*

The value of **terminal-io** is always the stream which connects to the user's console. For someone using the Lisp Machine from its keyboard and TV, the value will be **tv-terminal-stream**. The default values of the above four variables are streams which simply take whatever operations they are given and pass them on to whatever stream is the value of **terminal-io**. No user program should ever 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 error messages

sent to **error-output** can still get to the user by going through **terminal-io**, which is usually what is desired.

make-syn-stream *symbol*

make-syn-stream creates and returns a "synonym stream". Any operations sent to this stream will be redirected to the stream which is the value of *symbol*.

standard-input, **standard-output**, **error-output**, and **query-io** are initially bound to synonym streams which use the value of **terminal-io**.

18.5.5 Making Your Own Stream

Here is a sample output stream, which accepts characters and conses them onto a list.

```
(defun list-output-stream (op &optional arg1 &rest rest)
    (selectq op
        (:tyo
         (setq the-list (cons arg1 the-list)))
        (:which-operations '(:tyo))
        (otherwise
         (multiple-value-call
          (stream-default-handler (function list-output-stream)
                                  op arg1 rest)))))
```

The lambda-list for a stream must always have one required parameter (**op**), one optional parameter (**arg1**), and a rest parameter (**rest**). This allows an arbitrary number of arguments to be passed to the default handler. This is an output stream, and so it supports the **:tyo** operation. Note that all streams must support **:which-operations**. If the operation is not one that the stream understands (e.g. **:string-out**), it calls the **stream-default-handler**. The calling of the default handler is *required*, since the willingness to accept **:tyo** indicates to the caller that **:string-out** will work. The **multiple-value-call** (see page 19) is used so that if the default handler returns multiple values, the stream will return all of them.

Here is a typical input stream, which generates successive characters of a list.

```
(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)
            (or arg1
                (ferror nil "You got to the end of the stream.")))
           (t (prog1 (car the-list)
                     (setq the-list (cdr the-list))))))
    (:untyi
     (setq untyied-char arg1))
    (:which-operations '(:tyi :untyi))
    (otherwise
     (multiple-value-call
      (stream-default-handler (function list-input-stream)
                              op arg1 rest)))))
```

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.

The above stream uses a free variable (**the-list**) to hold the list of characters, and another one (**untyied-char**) to hold the **:untyi**ed character (if any). You might want to have several instances of this type of stream, without their interfering with one another. This is a typical example of the usefulness of closures in defining streams. The following function will take a list, and return a stream which generates successive characters of that list.

```
(defun make-a-list-input-stream (list)
  (let-closed ((list list) (untyied-char nil))
    (function list-input-stream)))
```

stream-default-handler *stream op arg1 rest*

stream-default-handler 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.:

18.6 Accessing Files

As of this writing, the Lisp Machine uses the A.I. PDP-10's file system to read and write files. The A.I. machine is accessed through the Chaos network. When the Lisp Machine is started, it will tell you whether it has successfully connected to the file system. The function **si:file-use-chaos** (not documented in this manual) allows you to initiate a connection to the file system server, and to control which machine is used as a file system.

At present, there may be no more than one file open for reading and one file open for writing at a time.

All of the functions herein are subject to change, and in general you shouldn't belive too much of this. However, it does describe the existing software.

[Blurb about "sectioned" file structure.]

open *filename options*

This is the function for accessing files. It returns a stream which is connected to the specified file. Unlike Maclisp, the **open** function only creates streams for *files*; other streams are created by other functions.

filename is the name of the file to be opened; it must be a string. Currently, files are stored on ITS and *filename* must be an ITS file name. If an ITS error (such as file not found) occurs when opening the file, a Lisp error is signalled.

options is either a single symbol or a (possibly-null) list of symbols. The following option 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.

: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.

:ascii The opposite of :fixnum. This is the default.

:single, :block
Ignored for compatibility with Maclisp.

For example, evaluating any of the forms

```
(open "info;dir >" ':in)
(open "INFO;DIR >" '(:read))
(open "DIR > INFO;" ':read)
```

will open the file "AI: INFO; DIR >", and return an input stream which will return successive characters of the file, and support the following operations: **:tyi, :untyi, :clear, :close, :name, :line-in**, and **:get-unique-id**. When the caller is finished with the stream, it should close the file by using the **:close** operation or the **close** function.

Opening a file output stream creates a new file with specified name (calling it "_LSPM_ OUTPUT" until it has been successfully closed) and returns a stream which supports the following operations: **:tyo**, **:close**, **:finish**, **:read-pointer**, **:name**, **:line-out**, and **:string-out**.

close *stream*
The **close** function simply performs the **:close** operation on *stream*.

18.6.1 Other File Operations

file-command &rest *strings*
This concatenates all of the *strings* and sends the result as a command to the PDP-10 FILE job. It returns the string which is the FILE program's response, except that if the response was empty, it returns **nil**. The returned string is special and will be clobbered by the next file operation, so you should copy it (with **string-append**, see page 81) if you want to save it.

file-command-careful &rest *strings*
This is the same as **file-command**, but if the string returned from the FILE program is not empty, it signals an error, using that string as the error-message.

file-error *Variable*
When an error such as "File Not Found" or "No Such Directory" occurs, (i.e. errors due to the current state of the file system), then instead of directly calling **error**, the file-access functions apply the value of **file-error** to the arguments upon which **error** would have been called. In fact, the default binding of **file-error** is to the symbol **error**. However, this convention allows flexibility in such programs as EINE, which may want to handle such errors specially.

This little feature is recognized as an inelegant kludge, which will be repaired when the error system is more fully developed.

file-error-status *filename*
This tries to open the file *filename*. If it gets an error, it returns the ITS error code, which will always be a small positive fixnum; otherwise it returns **nil**. In any case it leaves the file closed.

file-mapped-open *filename* &optional *(write-p* **nil***)*
Tells the pdp10 FILE program to map the specified file into its address space for random access, searching, etc. This is used in the present implementation of multi-sectioned files. **file-mapped-open** returns a stream to read from the file if *write-p* is **nil**, or write to it if *write-p* is **t**. The stream will apply only to the subrange of the file set by the latest **mapset** or **finddef** command given to the FILE job. On reading, when you get to the end of the range, it is considered the end-of-file. Giving another **mapset** or **finddef** command will make it start reading from a different range. To skip the rest of a range, do another **mapset** or **finddef** and do

a **:clear** operation on the stream. To set the range with **mapset**, do (**file-command** "**mapset** *start size*") where *start* and *size* are numbers converted to decimal. To set the range to **foo**'s definition, do (**file-command** "**finddef** " "**foo**").

file-qfasl-p *filename*
If the file is a QFASL file, return **t**; otherwise return **nil**. This works by checking the file itself, not the name; if opening the file gives an ITS error, an error is signalled.

file-exists-p *pathname*
Returns **nil** if the file *pathname* does not exist. If it does exist, returns **:qfasl** if it is a QFASL file, and otherwise **t**.

18.6.2 File Name Manipulation

file-expand-pathname *filename*
This defaults the FN2 to ">", and any other unspecified components from the current default filename. It then sets up the current default filename to be the resulting filename, and returns it. This will always return the filename in a canonical form.
Example:

```
(file-expand-pathname "lispm;foo") => "AI: LISPM; FOO >"
```

file-default-fn2 *filename fn2*
If *filename* does not specify its FN2 component, this returns a filename whose FN2 is *fn2*, and whose other components are from *filename*.
Example:

```
(file-default-fn2 "lispm;foo" "bar") => "AI: LISPM; FOO BAR"
```

file-set-fn2 *filename fn2*
Returns a filename whose FN2 is *fn2*, and whose other components are from *filename*.
Example:

```
(file-set-fn2 "lispm;foo >" "qfasl") => "AI: LISPM; FOO QFASL"
```

18.7 Rubout Handling

The rubout handler is a feature of all interactive streams, that is, streams which connect to terminals. Its purpose is to allow the user to edit minor mistakes in typein. 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 rubout handler also provides a few commands to do things like clear the screen.

The rubout handler will eventually provide the same editing commands as the editor, but at this writing they have not yet been conjoined.

The basic way that the rubout handler works is as follows. When an input function that reads an "object", such as **read** or **readline** but not **tyi**, is called to read from 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 will echo 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 function, **read** or whatever, decides it has enough input, it returns and control "leaves" the rubout handler. That was 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 which had been typed and not 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.

If an error occurs while inside the rubout handler, the error message is printed and the buffered input is redisplayed. The user can then type as he wishes; the input will be reparsed from the beginning in the usual fashion after he rubs out the characters which caused the error.

The rubout handler also recognizes the special characters Clear and Form. Form clears the screen and echoes back the buffered input. Clear is like hitting enough rubouts to flush all the buffered input.

If a Control or Meta character is typed to the rubout handler, the character is not echoed nor given to the program as input. The function which is the value of **rubout-handler-control-character-hook** is called. The default binding of this looks for control-Z and does a "Quit" if one is typed. This hook will go away when the rubout handler uses Control and Meta characters for editor commands. Note that when not inside the rubout handler, Control and Meta characters are passed through as input like ordinary characters.

rubout-handler-control-character-hook *Variable*

The value of this variable is a function called by the rubout handler when a Control or Meta character is typed. The function should take one argument, which is the character.

The way that the rubout handler is entered is complicated, since a ***catch** must be established. The variable **si:rubout-handler** is non-**nil** if the current process is inside the rubout handler. This is used to handle recursive calls to **read** from inside reader macros and the like. If **si:rubout-handler** is **nil**, and the stream being read from has **:rubout-handler** in its **:which-operations**, functions such as **read** send the **:rubout-handler** operation to the stream with arguments of the function and its arguments. The rubout handler initializes itself and establishes its ***catch**, then calls back to the specified function. If you look at the code in **read**, you will see some magic hair which is used to make sure that multiple values pass back through all this correctly. This will eventually become unnecessary.

si:rubout-handler *Variable*

t if control is inside the rubout handler in this process.

18.8 Special I/O Devices

-> pointers to separate chapter on TV, keyboard, and mouse.

-> pointer to separate chapter on Chaos net

-> how to use whatever else ought to go in here

19. Packages

19.1 The Need for Multiple Contexts

A Lisp program is a collection of function definitions. The functions are known by their names, and so each must have its own name to identify it. Clearly a programmer must not use the same name for two different functions.

The Lisp machine consists of a huge Lisp environment, in which many programs must coexist. All of the "operating system", the compiler, the EINE editor, and a wide variety of programs are provided in the initial environment. Furthermore, every program which the user uses during his 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 the user 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 them. 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 Chaos net 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 packer 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 declare a package named **chaos** to contain the Chaos net program, and another package **arpa** to hold the Arpanet program. When the Chaos net program is read into the machine, its symbols would be entered in the **chaos** package's name space. So when the Chaos net 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 Chaos net program—the right one. Similarly, the Arpanet program's **get-packet** would be read in using the **arpa** package's name space and would refer to the Arpanet program's **allocate-pbuf**.

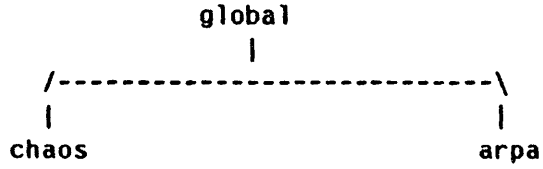
An additional function of packages is to remember the names of the files which constitute each program, making it easy to ask to load or recompile all of them at once.

To understand what is going on here, you should keep in mind how Lisp reading and loading works. When a file is gotten into the Lisp machine, either by being read or by being fasloaded, the file itself obviously cannot contain Lisp objects; it contains printed representations of those objects. When the reader encounters a printed representation of a symbol, it calls **intern** to look up that string in some name space and find a corresponding symbol to return. The package system arranges that the correct name space is used whenever a file is loaded.

19.2 The Organization of Name Spaces

We could simply let every name space be implemented as one obarray, e.g. one big table of symbols. The problem with this is that just about every name space wants to include the whole Lisp language: **car**, **cdr**, and so on should be available to every program. We would like to share the main Lisp system between several name spaces without making many copies.

Instead of making each name space be one big array, we arrange packages in a tree. Each package has a "superpackage" or "parent", from which it "inherits" symbols. Also, each package has a table, or "obarray", of its own additional symbols. The symbols belonging to a package are simply those in the package's own obarray, followed by those belonging to the superpackage. The root of the tree of packages is the package called **global**, which has no superpackage. **global** contains **car** and **cdr** and all the rest of the standard Lisp system. In our example, we might have two other obarrays called **chaos** and **arpa**, each of which would have **global** as its parent. Here is a picture of the resulting tree structure:



In order to make the sharing of the **global** package work, the **intern** function is made more complicated than in basic Lisp. In addition to the string or symbol to intern, it must be told which package to do it in. First it searches for a symbol with the specified name in the obarray of the specified package. If nothing is found there, **intern** looks at its superpackage, and then at the superpackage's superpackage, and so on, until the name is found or a root package such as **global** is reached. When **intern** reaches the root package, and doesn't find the symbol there either, it decides that there is no symbol known with that name, and adds a symbol to the originally specified package.

Since you don't normally want to worry about specifying packages, **intern** normally uses the "current" package, which is the value of the symbol **package**. This symbol serves the purpose of the symbol **obarray** in Maclisp.

Here's how that works in the above example. When the Chaos net program is read into the Lisp world, the current package would be the **chaos** package. Thus all of the symbols in the Chaos net program would be interned on the **chaos** package. If there is a reference to some well known global symbol such as **append**, **intern** would look for "**append**" on the **chaos** package, not find it, look for "**append**" on **global**, and find the regular Lisp **append** symbol, and return that. If, however, there is a reference to a symbol which the user made up himself (say it is called **get-packet**), the first time he uses it, **intern** won't find it on either **chaos** nor **global**. So **intern** will make a new symbol named **get-packet**, and install it on the **chaos** package. When **get-packet** is refered to later in the Chaos net program, **intern** will find **get-packet** on the **chaos** package.

When the Arpanet program is read in, the current package would be **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 Chaos net program got. However, if it refers to **get-packet**, it will *not* get the same one the Chaos net program got, because the **chaos** package is not being searched. Rather, the **arpa** and **global** packages are getting searched. So **intern** will create a new **get-packet** and install it on the **arpa** package.

So what has happened is that there are two **get-packets**: one for **chaos** and one for **arpa**. The two programs are loaded together without name conflicts.

19.3 Shared Programs

Now, a very important feature of the Lisp machine is that of "shared programs"; if one person writes a function to, say, print numbers in Roman numerals, any other function can call it to print Roman numerals. This contrasts sharply with PDP-10 system programs, in which Roman numerals have been independently reimplemented several times (and the ITS filename parser several dozen times).

For example, the routines to manipulate a robot arm might be a separate program, residing in a package named **arm**. If we have a second program called **blocks** (the blocks world, of course) which wanted to manipulate the arm, it would want to call functions which are defined on the **arm** obarray, and therefore not in **blocks**'s own name space. Without special provision, there would be no way for any symbols not in the **blocks** name space to be part of any **blocks** functions.

The colon character (":") has a special meaning to the Lisp reader. When the reader sees a colon preceeded by the name of a package, it will read in the next Lisp object with **package** bound to that package. The way **blocks** would call a function named **go-up** defined in **arm** would be by asking to call **arm:go-up**, because "**go-up** would be interned on the **arm** package. What **arm:go-up** means precisely is "The symbol named **go-up** in the name space of the package **arm**."

Similarly, if the **chaos** program wanted to refer to the **arpa** program's **allocate-pbuf** function (for some reason), it would simply call **arpa:allocate-pbuf**.

An important question which should occur at this point is how the names of packages are associated with their obarrays and other data. This is done by means of the "refname-alist" which each package has. This alist associates strings called *reference names* or *refnames* with the packages they name. Normally, a package's refname-alist contains an entry for each subpackage, associating the subpackage with its name. In addition, every package has its own name defined as a refname, referring to itself. However, the user can add any other refnames, associating them with any packages he likes. This is useful when multiple versions of a program are loaded into different packages. Of course, each package inherits its superpackage's refnames just as it does symbols.

In our example, since **arm** is a subpackage of **global**, the name **arm** is on **global**'s refname-alist, associated with the **arm** package. Since **blocks** is also a subpackage of **global**, when **arm:go-up** is seen the string "**arm**" is found on **global**'s refname alist.

When you want to refer to a symbol in a package which you and your superpackages have no refnames for—say, a subpackage named **foo** of a package named **bar** which is under **global**—you can use multiple colons. For example, the symbol **finish** in that package **foo** could be referred to as **foo:bar:finish**. What happens here is that the second name, **bar**, is interpreted as a refname in the context of the package **foo**.

19.4 Declaring Packages

Before any package can be referred to or loaded, it must be declared. This is done with the special form **package-declare**, which tells the package system all sorts of things, including the name of the package, the place in the package hierarchy for the new package to go, its estimated size, the files which belong in it, and some of the symbols which belong in it.

Here is a sample declaration:

```
(package-declare foo global 1000
        (("lispm;foo qfasl")
         ("lispm;bar qfasl")
         ("lispm;barmac >" defs))
        (shadow array-push adjust-array-size)
        (extern foo-entry))
```

What this declaration says is that a package named **foo** should be created as an inferior of **global**, the package which contains advertised global symbols. Its obarray should initially be large enough to hold 1000 symbols, though it will grow automatically if that isn't enough. Unless there is a specific reason to do otherwise, you should make all of your packages direct inferiors of **global**. The size you give is increased slightly to be a good value for the hashing algorithm used.

After the size comes the "file-alist". The files in the **foo** package are "lispm;foo" and "lispm;bar", both of which should be compiled, and "lispm;barmac", which should be read in as a text file. In addition, "barmac" is marked as a DEFS file, which means that the latest version of "barmac" must always be loaded before attempting to compile or load any of the other files. Typically a DEFS file contains macro definitions, compiler declarations, structure definitions, and the like. All the source files should start with

```
(pkg-contained-in "foo")
```

to help detect processing them in the wrong package. Soon it will automatically cause them to be processed in the right package, even if copied under strange names. (NOTE: **pkg-contained-in** IS NOT IMPLEMENTED YET! DON'T USE IT!)

Finally, the **foo** package "shadows" **array-push** and **adjust-array-size**, and "externs" **foo-entry**. What shadowing means is that the **foo** package should have its own versions of those symbols, rather than inheriting its superpackage's versions. Symbols by these names will be added to the **foo** package even though there are symbols on **global** already with those names. This allows the **foo** package to redefine those functions for itself without redefining them in the **global** package for everyone else. What externing means is that the **foo** package is allowed to redefine **foo-entry** as inherited from the **global** package, so that it *is* redefined for everybody. If **foo** attempts to redefine a function such as **car** which is present in the **global** package but neither shadowed nor externed, confirmation from the user will be requested.

Note that externing doesn't actually put any symbols into the **global** package. It just asserts permission to redefine symbols already there. This is deliberate; the intent is to enable the maintainers of the **global** package to keep control over what symbols are present in it. Because inserting a new symbol into the **global** package can cause trouble to unsuspecting programs which expect that symbol to be private, this is not supposed to be done in a decentralized manner by programs written by one user and used by another unsuspecting user. Here is an example of the trouble that could be caused: if there were two user programs, each with a function named **move-square**, and **move-square** were put on the **global** package, all of a sudden the two functions would share the same symbol, resulting in a name conflict. While all the definitions of the functions in **global** are actually supplied by subpackages which extern them (**global** contains no files of its own), the list of symbol names is centralized in one place, the file "ai: lispm2; global >", and this file is not changed without notifying everyone, and updating the **global** documentation.

Certain other things may be found in the declarations of various internal system packages. They are arcane and needed only to compensate for the fact that parts of those packages are actually loaded before the package system is. They should not be needed by any user package.

Your package declarations should go into separate files containing only package declarations. Group them however you like, one to a file or all in one file. Such files can be read with **load**. It doesn't matter what package you load them into, so use **user**, since that has to be safe.

If the declaration for a package is read in twice, no harm is done. If you edit the size to replace it with a larger one, the package will be expanded. If you change the file-alist, the new one will replace the old. At the moment, however, there is no way to change the list of shadowings or externals; such changes will be ignored. Also, you can't change the superpackage. If you edit the superpackage name and read the declaration in again, you will create a new, distinct package without changing the old one.

package-declare *Macro*

The **package-declare** macro is used to declare a package to the package system. Its form is:

(package-declare *name superpackage size file-alist option-1 option-2 ...*)

The interpretation of the declaration is complicated; see page 179.

19.5 Packages and Writing Code

The unsophisticated user need never be aware of the existence of packages when writing his programs. He should just load all of his programs into the package **user**, which is also what console type-in is interned in. Since all the functions which users are likely to need are provided in the **global** package, which is **user**'s superpackage, they are all available. In this manual, functions which are not on the **global** package are documented with colons in their names, so typing the name the way it is documented will work.

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. Whether for this reason or for any other, if you are loading your programs into packages other than **user** there are special constructs that you will need to know about.

One time when you as the programmer must be aware of the existence of packages is when you want to use a function or variable in another package. To do this, write the name of the package, a colon, and then the name of the symbol, as in **eine:ed-get-defaulted-file-name**. You will notice that symbols in other packages print out that way, too. Sometimes you may need to refer to a symbol in a package whose superior is not **global**. When this happens, use multiple colons, as in **foo:bar:ugh**, to refer to the symbol **ugh** in the package named **bar** which is under the package named **foo**.

Another time that packages intrude is when you use a "keyword": when you check for **eq**ness against a constant symbol, or pass a constant symbol to someone else who will check for it using **eq**. This includes using the symbol as either argument to **get**. In such cases, the usual convention is that the symbol should reside in the **user** package, rather than in the package with which its meaning is associated. To make it easy to specify **user**, a colon before a symbol, as in **:select**, is equivalent to specifying **user** by name, as in **user:select**. Since the **user** package has no subpackages, putting symbols into it will not cause name conflicts.

Why is this convention used? Well, consider the function **tv-define-pc-ppr**, which takes any number of keyword arguments. For example,

```
(tv-define-pc-ppr "foo" (list tvfont) ':vsp 6 ':sideways-p t)
```

specifies, after the two peculiar mandatory arguments, two options with names **vsp** and **sideways-p** and values **6** and **t**. The file containing this function's definition is in the **system-internals** package, but the function is available to everyone without the use of a colon prefix because the symbol **tv-define-pc-ppr** is itself inherited from **global**. But all the keyword names, such as **vsp**, are short and should not have to exist in **global**. However, it would be a shame if all callers of **tv-define-pc-ppr** had to specify **system-internals:** before the name of each keyword. After all, those callers can include programs loaded into **user**, which should by rights not have to know about packages at all. Putting those keywords in the **user** package solves this problem. The correct way to type the above form would be

```
(tv-define-pc-ppr "foo" (list tvfont) ':vsp 6 ':sideways-p t)
```

Exactly when should a symbol go in **user**? At least, all symbols which the user needs to be able to pass as an argument to any function in **global** must be in **user** if they aren't themselves in **global**. Symbols used as keywords for arguments by any function should usually be in **user**, to keep things consistent. However, when a program uses a specific property name to associate its own internal memoranda with symbols passed in from outside, the property name should belong to the program's package, so that two programs using the same property name in that way don't conflict.

19.6 Shadowing

Suppose the user doesn't like the system **nth** function; he might be a former **interlisp** user, and expecting a completely different meaning from it. Were he to say **(defun nth ---)** in his program (call it **interloss**) he would clobber the **global** symbol named "**nth**", and so affect the "**nth**" in everyone else's name space. (Actually, if he had not "externed" the symbol "**nth**", the redefinition would be caught and the user would be warned.)

In order to allow the **interloss** package to have its own **(defun nth ---)** without interfering with the rest of the Lisp environment, it must "shadow" out the global symbol "**nth**" by putting a new symbol named "**nth**" on its own obarray. Normally, this is done by writing **(shadow nth)** in the declaration of the **interloss** package. Since **intern** looks on the subpackage's obarray before **global**, it will find the programmer's own **nth**, and never the global one. Since the global one is now impossible to see, we say it has been "shadowed."

Having shadowed **nth**, if it is sometimes necessary to refer to the global definition, this can be done by writing **global:nth**. This works because the refname **global** is defined in the **global** package as a name for the **global** package. Since **global** is the superpackage of the **interloss** package, all refnames defined by **global**, including "**global**", are available in **interloss**.

19.7 Packages and Interning

The function **intern** allows you to specify a package as the second argument. It can be specified either by giving the package object itself, or by giving a string or symbol which is the name of the package. **intern** returns three values. The first is the interned symbol. The second is **t** if the symbol is old (was already present, not just added to the obarray). The third is the package in which the symbol was actually found. This can be either the specified package or one of its superiors.

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**. Bind the symbol **package** temporarily to the desired package, before calling things which call **intern**, when you want to specify the package. When you do this, the function **pkg-find-package**, which converts a string into the package it names, may be useful. While most functions that use packages will do this themselves, it is better to do it only once when **package** is bound. The function **pkg-goto** sets **package** to a package specified by a string. You shouldn't usually need to do this, but it can be useful to "put the keyboard inside" a package when you are debugging.

package *Variable*

The value of **package** is the current package; many functions which take packages as optional arguments default to the value of **package**, including **intern** and related functions.

pkg-goto &optional *pkg*

pkg may be a package or the name of a package. *pkg* is made the current package. It defaults to the **user** package.

pkg-bind *Macro*

The form of the **pkg-bind** macro is (**pkg-bind** *pkg* . *body*). *pkg* may be a package or a package name. The forms of the *body* are evaluated sequentially with the variable **package** bound to *pkg*.

Example:

```
(pkg-bind "eine"
          (read-from-string function-name))
```

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** means that the superpackages should not be searched. Thus, **intern-local** can be used to cause shadowing. **intern-local-soft** is a good low-level primitive for when you want complete control of what 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* &optional *(pkg* **package***)*
intern searches *pkg* and its superpackages sequentially, looking for a symbol whose print-name is equal to *string*. If it finds such a symbol, it returns three values: the symbol, **t**, and the package on which the symbol is interned. If it does not find one, it creates a new symbol with a print name of *string*, and returns the new symbol, **nil**, and *pkg*.

intern-local *string* &optional *(pkg* **package***)*
intern searches *pkg* (but *not* its superpackages), looking for a symbol whose print-name is equal to *string*. If it finds such a symbol, it returns three values: the symbol, **t**, and *pkg* If it does not find one, it creates a new symbol with a print name of *string*, and returns the new symbol, **nil**, and *pkg*.

intern-soft *string* &optional *(pkg* **package***)*
intern searches *pkg* and its superpackages sequentially, looking for a symbol whose print-name is equal to *string*. If it finds such a symbol, it returns three values: the symbol, **t**, and the package on which the symbol is interned. If it does not find one, it returns **nil**, **nil**, and **nil**.

intern-local-soft *string* &optional *(pkg* **package***)*
intern searches *pkg* (but *not* its superpackages), looking for a symbol whose print-name is equal to *string*. If it finds such a symbol, it returns three values: the symbol, **t**, and *pkg* If it does not find one, it returns **nil**, **nil**, and **nil**.

Each symbol rememberes which package it belongs to. While you can intern a symbol in any number of packages, the symbol will only remmeber one: normally, the first one it was interned in, unless you clobber it. This package is available as (**cdr** (**package-cell-location** *symbol*)). If the value is **nil**, the symbol believes that it is uninterned.

The printer also implicitly uses the value of **package** when printing symbols. If slashification is on, the printer tries to print something such that if it were given back to the reader, the same object would be produced. If a symbol which is not in the current name space were just printed as its print name and read back in, the reader would intern it on the wrong package, and return the wrong symbol. So the printer figures out the right colon prefix so that if the symbol's printed representation were read back in to the same package, it would be interned correctly. The prefixes only printed if slashification is on, i.e. **prin1** prints them and **princ** does not.

remob *symbol* &optional *package*
remob removes *symbol* from *package* (the names means "REMove from OBarray"). *symbol* itself is unaffected, but **intern** will no longer find it on *package*. **remob** is always "local", in that it removes only from the specified package and not from any superpackages. It returns **t** if the symbol was found to be removed. *package* defaults to the contents of the symbol's package cell, the package it is actually in. (Sometimes a symbol can be in other packages also, but this is unusual.)

mapatoms *function* &optional *(package* **package***)* *(superiors-p* **t***)*

function should be a function of one argument. **mapatoms** applies *function* to all of the symbols in *package*. If *superiors-p* is **t**, then the function is also applies to all symbols in *package*'s superpackages. Note that the function will be applied to shadowed symbols in the superpackages, even though they are not in *package*'s name space. If that is a problem, *function* can try applying **intern** in *package* on each symbol it gets, and ignore it if it is not **eq** to the result of **intern**; this measure is rarely needed.

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 of *package*'s subpackages. Since *package* defaults to the **global** package, this normally gets at all of the symbols in all packages. It is used by such functions as **apropos** and **who-calls** (see page 261)

Example:

```
(mapatoms-all
  (function
    (lambda (x)
      (and (alphalessp 'z x)
           (print x)))))
```

pkg-create-package *name* &optional *(super* **package***)* *(size* **200***)*

pkg-create-package creates and returns a new package. Usually packages are created by **package-declare**, but sometimes it is useful to create a package just to use as a hash table for symbols, or for some other reason.

If *name* is a list, its first element is taken as the package name and the second as the program name; otherwise, *name* is taken as both. In either case, the package name and program name are coerced to strings. *super* is the superpackage for this package; it may be **nil**, which is useful if you only want the package as a hash table, and don't want it to interact with the rest of the package system. *size* is the size of the package; as in **package-declare** it is rounded up to a "good" size for the hashing algorithm used.

pkg-kill *pkg*

pkg may be either a package or the name of a package. The package should have a superpackage and no subpackages. **pkg-kill** takes the package off its superior's subpackage list and refname alist.

pkg-find-package *x* &optional *(create-p* **nil***)* *(under* **"global"***)*

pkg-find-package tries to interpret *x* as a package. Most of the functions whose descriptions say "... may be either a package or the name of a package" call **pkg-find-package** to interpret their package argument.

If *x* is a package, **pkg-find-package** returns it. Otherwise it should be a symbol or string, which is taken to be the name of a package. The name is looked up on the refname alists of **package** and its superpackages, the same as if it had been typed as

part of a colon prefix. If this finds the package, it is returned. Otherwise, *create-p* controls what happens. If *create-p* is **nil**, an error is signalled. Otherwise, a new package is created, and installed as an inferior of *under*.

pkg-map-refnames *function package*

pkg-map-refnames is used by the printer to figure out the correct package prefix for symbols, when they are being printed with slashification. It is provided for sophisticated use of the package system. *package* should be the package of the symbol to be printed. *function* should be a function of two arguments which will be called successively on each reference name to be printed. The first argument to *function* is the name (as a string), and the second is the number of reference names to be printed after this one (i.e., *function* is called on successive reference names, on a decreasing fixnum which is **0** on the last call). Of course, *function* need not print the reference names; it may do anything it wants with them.

A package is implemented as a structure, created by **defstruct**. The following accessor macros are available on the **global** package:

pkg-name	The name of the package, as a string.
pkg-refname-alist	The refname alist of the package, associating strings with packages.
pkg-super-package	The superpackage of the package.

19.8 Status Information

The current package—where your type-in is being interned—is always the value of the symbol **package**. A package is a named structure which prints out nicely, so examining the value of **package** is the best way to find out what the current package is. Normally, it should be **user**, except when inside compilation or loading of a file belonging to some other package.

To get more information on the current package or any other, use the function **pkg-describe**. Specify either a package object or a string which is a refname for the desired package as the argument. This will print out everything except a list of all the symbols in the package. If you want *that*, use **(mapatoms 'print** *package* **nil)**. **describe** of a package will call **pkg-describe**.

19.9 How Packages Affect Loading and Compilation

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.

When you have mentioned a file in a package's file-alist, requesting to compile that file with **qc-file** or loading it with **load** automatically selects that package to perform the operation. This is done by inverting the package-to-file correspondence described by the file-alists and remembering the inversion in the form of **:package** properties on symbols in the **files** package (the symbol representing the file is **(intern (file-expand-pathname** *filename*) **"files"))**.

The system can get the package of a source file from its "editor property list". 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 into the QFASL file. If a file is not mentioned in a package's file-alist and doesn't have such a package specification in it, the system loads it into the current package, and tells you what it did.

To compile or load all of the files of a package, you can use the **pkg-load** function (see page 194), which uses the file-alist from the package declaration.

19.10 Subpackages

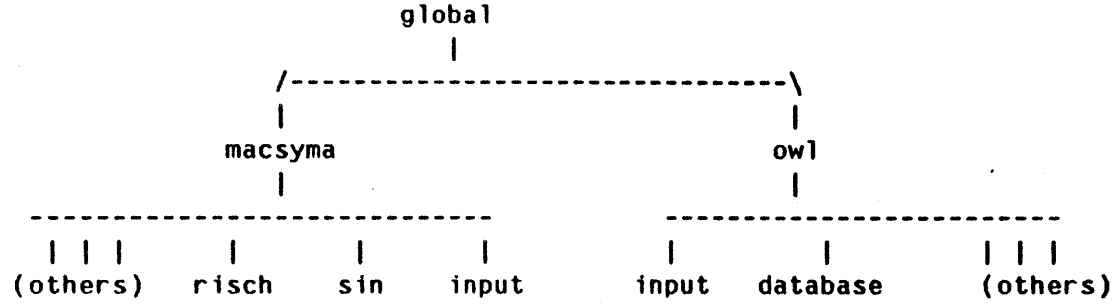
Usually, each independent program occupies one package, which is directly under **global** in the hierarchy. But large programs, such as Macsyma, are usually made up of a number of sub-programs, which are maintained by a small number of people. We would like each sub-program to have its own name space, since the program as a whole has too many names for anyone to remember. So, we can make each sub-program into its own package. However, this practice requires special care.

It is likely that there will be a fair number of functions and symbols which should shared by all of the sub-programs of Macsyma. These symbols should reside in a package named **macsyma**, which would be directly under **global**. Then, each part of **macsyma** (which might be called **sin**, **risch**, **input**, and so on) would have its own package, with the **macsyma** package as its superpackage. To do this, first declare the **macsyma** package, and then declare the **risch**, **sin**, etc. packages, specifying **macsyma** as the superpackage for each of them. This way, each sub-program gets its own name space. All of these declarations would probably be in a together in a file called something like "macpkg".

However, to avoid a subtle pitfall (described in detail in the appendix), it is necessary that the **macsyma** package itself contain no files; only a set of symbols specified at declaration time. This list of symbols is specified using **shadow** in the declaration of the **macsyma** package. At the same time, the file-alist specified in the declaration must be **nil** (otherwise, you will not be allowed to create the subpackages). The symbols residing in the **macsyma** package can have values and definitions, but these must all be supplied by files in

macsyma's subpackages (which must "**extern**" those symbols as necessary). Note that this is exactly the same treatment that **global** receives: all its functions are actually defined in files which are loaded into **system-internals** (si), **compiler**, etc.

To demonstrate the full power and convenience of this scheme, suppose there were a second huge program called **owl** which also had a subprogram called **input** (which, presumably, does all of the **input**ting for **owl**), and one called **database**. Then a picture of the hierarchy of packages would look like this:



Now, the **risch** program and the **sin** program both do integration, and so it would be natural for each to have a function called **integrate**. From inside **sin**, **sin's integrate** would be referred to as "**integrate**" (no prefix needed), while **risch's** would be referred to as "**risch:integrate**". Similarly, from inside **risch**, **risch's** own **integrate** would be called "**integrate**", whereas **sin's** would be referred to as "**sin:integrate**".

If **sin's integrate** were a recursive function, the implementor would be referring to it from within **sin** itself, and would be happy that he need not type out "**sin:integrate**" every time; he can just say "**integrate**".

From inside the **macsyma** package or any of its other sub-packages, the two functions would be referred to as "**sin:integrate**" and as "**risch:integrate**". From anywere else in the hierarchy, they would have to be called "**macsyma:sin:integrate**" and "**macsyma:risch:integrate**".

Similarly, assume that each of the **input** packages has a function called **get-line**. From inside **macsyma** or any of **macsyma's** subprograms (other than **input**), the relevant function would be called **input:get-line**, and the irrelevant one **owl:input:get-line**. The converse is true for **owl** and its sub-programs. Note that there is no problem arising from the fact that both **owl** and **macsyma** have subprograms of the same name (**input**).

You might also want to put Macsyma's **get-line** function on the **macsyma** package. Then, from anywehere inside Macsyma, the function would be called **get-line**; from the **owl** package and subpackages it could be referred to as **macsyma:get-line**.

19.11 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.

When the world begins to be loaded, there is no package system. There is one "obarray", whose format is different from that used by the package system. After sufficiently much of the Lisp environment is present for it to be possible to initialize the package system, that is done. At that time, it is necessary to split the symbols of the old-style obarray up among the various initial packages.

The first packages created by initialization are the most important ones: **global**, **system**, **user**, and **system-internals**. All of the symbols already present are placed in one of those packages. By default, a symbol goes into **system-internals**. Only those placed on special lists go into one of the others. These lists are the file "AI: LISPM2; GLOBAL >" of symbols which belong in **global**, the file "AI: LISPM2; SYSTEM >" which go in **system**, and the file "AI: LISPM2; KWDPKG >" of symbols which belong in **user** (at the moment, these are actually loaded into **global**, because not everything has been converted to use colons where necessary).

After the four basic packages exist, the package system's definition of **intern** is installed, and packages exist. Then, the other initial packages **format**, **compiler**, **eine**, etc. are declared and loaded using **package-declare** and **pkg-load**, in almost the normal manner. The exception is that a few of the symbols present before packages exist really belong in one of these packages. Their package declarations contain calls to **forward** and **borrow**, which exist only for this purpose and are meaningful only in package declarations, and are used to move the symbols as appropriate. These declarations are kept in the file "AI: LISPM; PKGDCL >".

globalize &rest *symbols*

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 a package which ought just to inherit the one from **global**. When this happens, you can correct the situation by doing (**globalize** "*symbol-name*"). This function creates a symbol with the desired name in **global**, merges whatever value, function definition, and properties can be found on symbols of that name together into the new symbol (complaining if there are conflicts), and forwards those slots of the existing symbols to the slots of the new one using one-q-forward pointers, so that they will appear to be one and the same symbol as far as value, function definition, and property list are concerned. They cannot all be made **eq** to each other, but **globalize** does the next-best thing: it takes an existing symbol from **user**, if there is one, to put it in **global**. Since people who check for **eq** are normally supposed to specify **user** anyway, they will not perceive any effect from moving the symbol from **user** into **global**.

If **globalize** is given a symbol instead of a string as argument, the exact symbol specified is put into **global**. You can use this when a symbol in another package, which should have been inherited from **global**, is being checked for with **eq**—as long as there are not *two* different packages doing so. But, if the symbol is supposed to be in **global**, there usually should not be.

19.12 Initial Packages

The initially present packages include:

global Contains advertised global functions.

user Used for interning the user's type-in. Contains all keyword symbols.

sys or **system**
Contains various internal global symbols used by various system programs.

si or **system-internals**
Contains subroutines of many advertised system functions. **si** is a subpackage of **sys**.

compiler Contains the compiler and fasload. **compiler** is a subpackage of **sys**.

eine Contains the Eine editor.

chaos Contains the Chaos net controller.

supdup Contains the Supdup program.

peek Contains the Peek program.

format Contains the function **format** and its associated subfunctions.

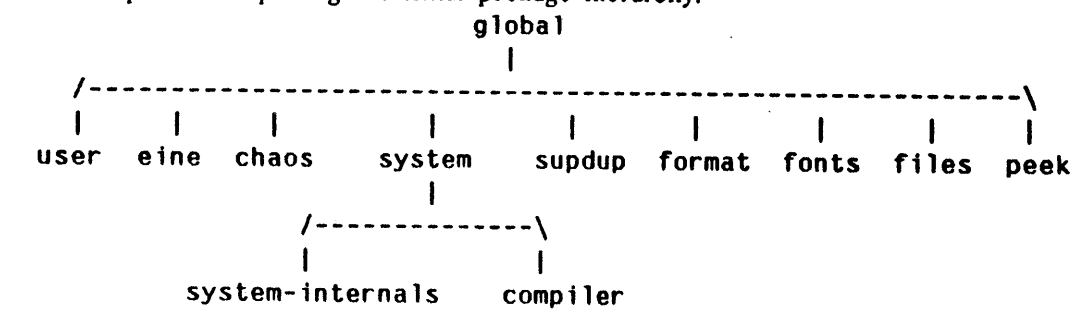
Packages which are used for special sorts of data:

fonts Contains the names of all fonts.

files Contains the file-symbols of all files. Many properties are kept on these symbols to remember information about files which are in use.

format Contains the keywords for **format**, as well as the code.

Here is a picture depicting the inital package hierarchy:



19.13 Multiple Instantiations of a Program

This isn't finished yet, which is why we don't say how to do any of this.

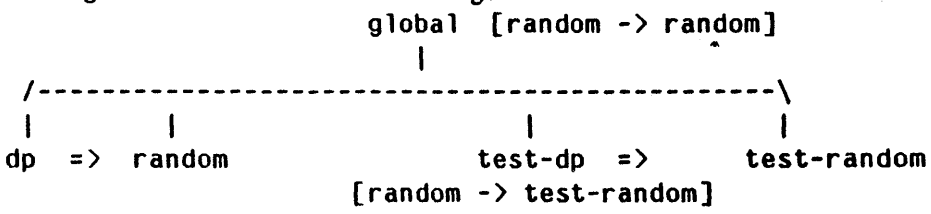
Suppose a maintainer of EINE (the Lisp Machine editor) has made some changes to EINE, and would like to debug them. He has a problem: if he reads in the new version, which presumably may be full of bugs, then he will not be able to do any editing! This would be annoying, since the editor is very useful.

We would like both the regular and the experimental versions of the editor to *both* be loaded into the Lisp world. In order for two definitions of each editor function to coexist, we need to load the new version into a separate package, which must have a different name (not named "**eine**", like the package the original editor is in). If the test version's package is called "**test-eine**", then the user can try it by calling **(test-eine:ed)**, and edit it using **(ed)**.

However, there is a problem to be faced. The editor redefines a few entry-point functions (**ed**, **edprop**, etc) which reside in **global**. If the test editor redefined them, the whole point of the separate package would be lost. So, the **test-eine** package must **shadow** all the symbols which the regular **eine** package **externs**.

Further complications are needed to make it possible to test one program using another instead of by hand. Suppose that there is a program named **random** residing in its own package, and containing a function named **number**. Suppose that we have a debugged program **dp** (Dissociated Press) which uses **random:number**. And now, we have written a new version of **random** and want to test it using **dp**, without installing it and breaking system tools which use **random**. What we want to do is to load in a test version of **random**, **test-random**, and also a **test-dp** which will refer to it, to test it with.

This can be done if we can make the **test-dp** package take references to **random** as references to the **test-random** package. All this takes is an entry on **test-dp**'s refname-alist, associating the name "**random**" with the **test-random** package. Then, when **random:number** is seen in the course of reading in the old **dp** program into **test-dp**, **test-random:number** will actually be used. Note that normally **test-dp** wouldn't have an entry on its own refname-alist for "**random**"; it would inherit the association from **global**. We are actually "shadowing" in **test-dp** the definition of "**random**" as a package refname which is present in **global**. Here is what we will get.



("=>" indicates who calls whom; "->" indicates a refname).

So far, every package has had its own name as a refname for itself. A test package, however, shouldn't have its actual name as a refname for itself, but the name its program expects: "**random**", not "**test-random**". This is necessary to handle test packages with

subpackages right, together with shadowing. In fact every package has a "program name" as well as a "name". For ordinary packages, they are the same, but for a test package, the program name is identical to that of the original package.

Suppose we have the Macsyma program with all of its sub-packages as described above. Further assume that the **input** sub-program's author has his own symbol named **simp**, and he calls **macsyma:simp** in various places to get the one in the **macsyma** package. Now, say someone wants to load an experimental **macsyma** into the machine: he would name the new obarray **test-macsyma** or something. In order to assure that the reference to **macsyma:simp** is properly resolved, the refname-alist of **test-macsyma** must contain **test-macsyma** under the name **macsyma**. This, finally, is the reason why each package has a reference to itself on its refname-alist.

20. Files

This chapter explains how the Lisp Machine system interacts with files and the file system. It explains how to keep your programs in files and how to get them into the Lisp environment, how they relate to packages, how they are divided into sections, and how they are seen by EINE (the editor).

Eventually, Lisp Machines will be able to support their own file sytems, or use a special purpose "File Computer" over the Chaosnet. At the moment, the prototype Lisp Machine uses the A.I. PDP-10 file system. To allow it to access the PDP-10 (which is not yet attached to the Chaosnet), a special program must be run on the PDP-10, which is invoked by typing **:lmio;file** to DDT.

A *pathname* or *filename* is a string of characters which identifies a file in the file system. On the existing file system, a pathname looks like

"*device*: *directory*; *fn1 fn2*"

It is assumed that the reader of this document is familiar with the meanings of these pathnames, and the use of ">" as the *fn2* in a pathname. Unlike Maclisp, Lisp Machine functions usually take filenames as a character string, rather then as a list. Most functions understand pathnames in which some components are not specified. For example, in the string **"lispm;qmod"**, the *device* and *fn2* are not specified.

20.1 Functions for Loading Programs

20.1.1 Functions for Loading Single Files

load *pathname* &optional *pkg*

This function loads the file *pathname* into the Lisp environment. If the file is a QFASL file, it calls **fasload**; otherwise it calls **readfile**. *pkg* should be a package or the name of a package, and if it is given it is used as the current package when the file is read in. Usually it is not given; when it is not supplied explicitly, **load** tries to figure out what package to use by calling **pkg-find-file-package**. If the FN2 is not specified in *pathname*, **load** first tries appending the *fn2* **"qfasl"**, and then tries the *fn2* ">" if the **"qfasl"** file is not found.

readfile *pathname*

readfile sequentially reads and evaluates forms from the file *pathname*, in the current package.

fasload *pathname*

fasload reads in and processes a QFASL file, in the current package. That is, it defines functions and performs other actions as directed by the specifications inserted in the file by the compiler.

20.1.2 Loading and Compiling Whole Packages

Because each package has a file-alist, it is possible to request that the files of a package be compiled or loaded, as needed. This is done with the **pkg-load** function, which takes as arguments a package and a list of keywords (or one keyword) specifying the precise nature of the operation. For example, (**pkg-load "eine" ':compile**) would recompile and reload the files of the **eine** package, such as require it.

pkg-load *package* &optional *keywords*

This function loads and/or compiles the files of a package. *package* may be a package or a package name; *keywords* should be one of the keyword symbols below or a list of keywords. The keywords control what **pkg-load** does.

The keywords defined include:

:confirm	Ask for confirmation before doing it (this is the default);
:noconfirm	Don't ask for confirmation
:compile	Compile files before loading;
:nocompile	Do not compile (this is the default);
:load	Load files (the default);
:noload	Don't load (but compile, if that was specified);
:selective	Ask about each file;
:complete	Don't ask about each file (the default);
:reload	Compile or load even files which appear not to need it;
:noreload	Only process files which have newer versions on disk (the default);
:recursive	Also process packages this one refers to;
:defs	Process only DEFS files.

See also **recompile-world** (page 262).

21. Processes

Processes are used to implement multi-processing. Several computations can be executed "concurrently" by placing each in a separate process. A computation in a process may also *wait* for something to happen, during which time it does not execute at all.

A *process* is a Lisp structure with the following components:

process-name
The name of the process, as a string. This string is only used for the process's printed representation, and for programs to print out; it can be anything reasonably mnemonic.

process-stack-group
The stack group currently associated with this process. When the process is run, this stack group will be resumed. See below.

process-wait-function
A function, applied to the argument list in the process's **process-wait-argument-list** to determine whether the process is runnable. The function should return **nil** if the process is not ready to run.

process-wait-argument-list
The arguments to which the **process-wait-function** is applied.

process-whostate
The reason this process is waiting, as a string. This is only used for display by the who-line or various programs; it can be anything reasonably mnemonic.

process-job The job associated with this process, or **nil** if the process is not associated with any job. See the chapter on jobs (page 199).

process-initial-stack-group
The function **process-preset** (see page 197) will make the **process-stack-group** be this stack group.

At any time there is a set of *active processes*. Each active process is either trying 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 cycles through the active processes, determining for each process whether it is ready to be run, or whether it is waiting. The scheduler determines whether a process is ready to run 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. If the process is ready to run, the scheduler resumes the **process-stack-group** of the process. For example, if a process were waiting for input from the keyboard, its wait-function might be **kbd-char-available**, which returns non-**nil** if there is a character available from the keyboard. Since **kbd-char-available** takes no arguments, the wait-argument-list of the process would be **nil**.

When a process's wait-function returns non-**nil**, the scheduler will resume its stack group and let it proceed. The process is now the *current process*, that is, the one process that is running on the machine. The scheduler sets the variable **current-process** to it. It will remain the current process and continue to run until either it decides to wait, or a *sequence break* occurs. A process can wait for some condition to become true by calling **process-wait** (see page 197), which will set up its wait-function and wait-argument-list accordingly, and resume the scheduler stack group. 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. In either case, the scheduler will continue cycling through the active processes. This way, each process that is ready to run will get its share of time in which to execute.

Note: Sequence breaks are not yet implemented, and so the scheduler only regains control when the running process calls **process-wait**. Any process that simply computes for a long time without waiting will keep all of the other processes from running. In the future, seqence breaks will happen periodically and at interesting times when some process's wait condition may have become true.

21.1 Functions for Manipulating Processes

process-create *name job* &rest *options*

process-create creates and returns a new process. *name* may be any string, *job* is usually **t**, and there are usually no options. The options are used in creating the stack group which executes on behalf of this process.

The fields of the new process are set up as follows:

process-name
name, which should be a string.

process-job *job*. If *job* is **t**, the current job is used instead. Otherwise *job* should be **nil** (meaning that the process is not associated with any job), or a job.

process-stack-group
A newly created stack group. The *options* argument to **process-create** are the options passed to **make-stack-group** (see page 107) used when creating this stack group.

process-initial-stack-group
The same as the **process-stack-group**.

The rest of the fields are set to **nil**; the process should not be enabled until **process-preset** (see below) is called.

process-preset *process initial-function* &rest *options*

process-preset initializes the state of a process. First, it restores the **process-stack-group** from the **process-initial-stack-group**. Then it presets the stack group, passing the **initial-function** and **options** arguments to **stack-group-preset** (see page 108). Finally it sets the **process-wait-function** and **process-argument-list** to return **t**, so that the process will be ready to run. The process is now ready to be enabled (see **process-enable** below).

process-kill *process*

This deactivates *process* if it is active, and dissociates it from its associated job (if any).

process-enable *process*

Enable **process**. If *process* has no associated job, or if its job if process-enabled, *process* is activated.

process-disable *process*

Disable *process*. If it was active, deactivate it.

process-wait *whostate function* &rest *arguments*

process-wait sets the current-process's **process-whostate**, **process-wait-function**, and **process-wait-argument-list** from its three arguments, which makes the current process wait 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 bindings in effect when **process-wait** was called will *not* be in effect when *function* is applied. Be careful when using any free references in *function*.

Example:

```
;; This won't work.
((lambda (until)
    (process-wait "sleep" '(lambda () (> (time) until))))
 500)

;; This is the right way to do it.
(process-wait "sleep" '(lambda (until) (> (time) until)) 500)
```

When running the **process-wait-function**, the scheduler sets the variables **current-process** and **current-job** to the process being considered and its job, so the **process-wait-function** can use them; for example:

```
;; Wait until I get the keyboard.
(process-wait "kbd" '(lambda () (eq kbd-job current-job)))
```

process-sleep *interval*
This simply waits for *interval* sixtieths of a second, and then returns. It uses **process-wait**.

process-allow-schedule
This function simply waits for a condition which is always true; all other processes will get a chance to run before the current process runs again.

21.2 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 should unlock the lock.

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 a time.

process-lock *locative*
This is used to seize the lock which *locative* points to. If necessary, **process-lock** will wait until the lock becomes free. When **process-lock** returns, the lock has been seized.

process-unlock *locative*
This is used to unlock the lock which *locative* points to. If the lock is free or was locked by some other process, an error is signaled. Otherwise the lock is unlocked.

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.

process-lock and **process-unlock** are written by use of a sub-primitive function called **%store-conditional** (see page 115), which is sometimes useful in its own right.

22. TVOBs and Jobs

[The subject of this chapter is currently being redesigned. The contents of this chapter will be completely changed in the next edition of this manual.]

22.1 Introduction to the Concepts of This Chapter

TVOBs (TV OBjects) represent permission to use the TV screen. The TVOB mechanism is provided to allow the TV to be shared between all of the activities the user may be conducting, without those activities getting in each other's way.

A *job* is a collection of processes and TVOBs, grouped together for the user's convenience. The processes can be started and stopped together, and the TVOBs can be put on or taken off the TV together.

22.2 TVOBs

A *TVOB* (TV OBject) is a Lisp structure with the following components:

tvob-name This is the name of the TVOB, as a string. It is only used for the TVOB's printed representation, and can be anything reasonably mnemonic.

tvob-x1 The first (leftmost) column of the TVOB's area of the screen.

tvob-y1 The first (highest) line of the TVOB's area of the screen. The **tvob-x1** and **tvob-y1** are the co-ordinates of the upper-left-hand corner of the TVOB's rectangular area.

tvob-x2 The first (leftmost) column to the right of the TVOB's area of the screen.

tvob-y2 The first (highest) line below the TVOB's area of the screen. The **tvob-x2** and **tvob-y2** are the co-ordinates of the lower-right-hand corner of the TVOB's rectangular area, with 1 added to each. Thus the height of the tvob is the difference between its **tvob-y2** and **tvob-y1**, and the width is the difference between the **tvob-x2** and **tvob-x1**.

tvob-handler A function, described below.

tvob-info This field may contain anything at all; it is meant to be used by the **tvob-handler** function.

tvob-job The job associated with this TVOB, or **nil** if there is no associated job.

tvob-priority
Either **t** or **nil**. If it is **t**, the functions which allocate area on the screen (**tvob-create** and **tvob-create-expandable**, see page 206) will be reluctant to allocate over this TVOB's area of the screen.

tvob-screen The screen on which this TVOB is displayed. See page 210.

tvob-status This field is provided for the convenience of **tvob-handler** functions. It contains one of the following symbols:

:selected	The TVOB is the selected TVOB. Only one TVOB will have this **tvob-status**.
:exposed	The TVOB is not selected, but is on **exposed-tvobs**. This means that this TVOB is not covered by any other TVOB; its screen area is fully exposed.
nil	The TVOB is not on **exposed-tvobs**.

tvob-clobbered-p
This field is provided for the convenience of **tvob-handler** functions. It is **t** if the TVOB has been sent a **:clobber** or **:set-edges** command more recently than an **:update** command; otherwise it is **nil**.

tvob-mouse-handler
A function to call when the mouse enters this TVOB's screen region. This allows the TVOB to take over control of the mouse. This field is **nil** if this TVOB does not do anything special with the mouse.

tvob-mouse-action
See **mouse-default-handler** (<not-yet-written>).

tvob-plist A disembodied property list. Use, for example, **(get (locf (tvob-plist tvob)) 'mumble)**.

It is often useful to divide the TV screen up into several parts, and do different things in each part. Sometimes one program wants to split up the screen, as Eine does; sometimes the user wants to run several programs at once, and each program wants some space on the screen. At any time, there is a set of *active TVOBs* (TV objects) which are sharing the screen. Each TVOB has a rectangular piece of the screen on which it does its displaying; it is not allowed to go outside its area.

It is possible for two active TVOB's regions of the screen to overlap. When this happens, only one of them is *exposed* (fully visible); the other is partially or fully buried. There is a subset of the active TVOBs called the *exposed* TVOBs; no two exposed TVOBs overlap. The TVOBs act as if they were a bunch of rectangular sheets of paper on a desktop; some are up at the top, and others are partially buried. Various programs can "pull" a non-exposed TVOB up to the top, making it exposed and making some other TVOB(s) non-exposed. Several functions of the job system, explained below, keep track of and change which TVOBs are active, and which are exposed.

The job system also keeps track of one TVOB called the *selected TVOB*. Conceptually, the selected TVOB is the one in which the user is interested at the moment, and it is usually the one that is responding to the keyboard. For example, when Eine is being used, the TVOB of the window in which the user is editing is the selected TVOB. The selected TVOB is always exposed. A TVOB's being selected, exposed but not selected, or not exposed at all is called the TVOB's *status*.

Usually a program will want certain actions to be taken when the status of a TVOB changes. When the TVOB associated with an Eine window becomes exposed, Eine generally wants to redisplay the window, and when the TVOB is selected, Eine starts blinking the window's blinkers and makes its buffer be the current buffer. In order to let a user program know that the status of a TVOB has changed, so that it can do these things, there is a function called the *handler* associated with every TVOB. When the status of the TVOB changes, its handler is applied to three arguments: the TVOB itself, a keyword symbol indicating what kind of change of status is occurring, and a list of other information whose meaning is dependant on the value of the second argument. The applications of this function can be thought of as a "command" being sent to the TVOB. For example, when a TVOB becomes selected, it is "sent a command" telling it so; that is, the handler is applied to the TVOB, the keyword **:select**, and **nil**.

Here is a list of all of the keyword symbols (i.e. all of the kinds of commands) that are used. In addition to status changes, requests for the TVOB to update and relocate itself cause the handler to be invoked.

:expose The TVOB is being made exposed. This command is only sent when the TVOB is active and not exposed.

:deexpose The TVOB is no longer exposed. This command is only sent when the TVOB is active and exposed.

:select The TVOB is now selected. This command is only sent when the TVOB is exposed and not selected.

:deselect The TVOB is no longer selected. This command is only sent when the TVOB is selected.

:clobber This command means that for some reason, the TVOB's area of the screen has been altered; future **:update** and **:clean** commands should not assume that the screen is as it was left. Most TVOBs will ignore this message, since the information is saved in the **tvob-clobbered-p** element of the TVOB (see below). This command is only sent when the TVOB is exposed.

:update This command is only sent when the TVOB is exposed. The TVOB should assure that its area of the screen contains whatever it is supposed to contain. Just what **:update** means depends on the program. Some programs can remember the contents of what is on the TVOB and can refresh it at will; others do not remember the contents, and cannot reconstruct them.

The former kind, upon receiving the **:update** command, should update the TVOB. If the TVOB has not been clobbered, the handler can assume that whatever it last put there is still there, and it may be able to avoid redisplaying. The **tvob-clobbered-p** field of the TVOB is set to **t** by the job system after a **:clobber** or **:set-edges** command is sent, and to **nil** after an **:update** or **:clean** command is sent. The handler can determine whether or not its area of the screen has been clobbered by simply looking at the **tvob-clobbered-p**.

The latter kind of TVOB cannot update, and should ignore the **:update** command entirely.

:clean **:clean** is like **:update** except that TVOBs of the kind that cannot refresh themselves should clear their areas instead of doing nothing. Like **:update**, this command is only sent when the TVOB is exposed. **:clean** is sent to all exposed TVOBs when the user requests that the screen be cleaned up; for instance when the FORM key is pressed in most programs.

:set-edges The TVOB should change its area of the screen. This command takes four arguments: the new left edge, top edge, bottom edge, and right edge, in raster units. The first two are inclusive, and the other two are exclusive. The elements of the TVOB structure that hold the screen area (**tvob-xl** etc.) will be updated automatically; the handler need not change them itself. The handler should update any other associated information; for example, if the TVOB has an associated "piece of paper", it should call **tv-redefine-pc-ppr** (see page 233).

The **tvob-info** component of the TVOB is provided to give the handler somewhere to put its internal state. It is usually some kind of structure, depending on what program created the TVOB. For example, it might be a piece of paper (see page 215).

22.3 Jobs

A *job* is a Lisp structure with the following components:

job-name The name of the job, as a string. This is only used for the printed representation of the job or for display by programs, and may be anything reasonably mnemonic.

job-tvobs A list of all TVOBs associated with this job.

job-processes
A list of all processes associated with this job.

job-enabled-tvobs
A list of this job's enabled tvobs. Each TVOB on this list is also on the **job-tvobs** list. The order of the enabled tvobs list is "highest" first; this list is sometimes passed to **tvob-setup**.

job-enabled-processes
A list of this job's enabled processes. This list is a subset of the **job-processes** list.

job-tvob-enabled-p
If this is non-**nil**, this job is *tvob-enabled*; its enabled TVOBs are active.

job-process-enabled-p
If this is non-**nil**, this job is *process-enabled*; its enabled processes are active.

job-who-line-process
Whenever this job becomes the **kbd-job**, the process in this component becomes the **tv-who-line-process** (The process whose **process-whostate** is displayed in the who-line) (see page 228).

job-tvob-selector
nil, or a function which is called by **tvob-setup** (see page 207) when this job is current and a new **selected-tvob** is needed. The function takes no arguments and doesn't return anything in particular. It isn't required to do anything, but it normally should call **tvob-select** with an appropriate TVOB.

job-forced-input
If non-**nil**, a character or a string which is forced input for this job. This is a character or characters which are pretending to come from the keyboard but really originated from another process or the mouse. See the function **force-kbd-input** (page 235).

job-forced-input-index
Index into **job-forced-input** when it is a string.

At any time, the user of the Lisp Machine may be conducting several different activities. For example, he may want to temporarily stop editing in order to send some mail; he might want to start up a file transfer, and while waiting for it to finish, continue editing.

Each such activity, in general, will want some processes to do computation, and some pieces of the screen (TVOBs) on which to do output. When the user is not concerned with some activity, he may want its processes to stop, and/or its TVOBs to stop displaying. In order to make it easy to deactivate a set of processes and TVOBs, such a set may be grouped together as a *job*.

Every job has a set of processes and TVOBs; these sets are represented by the lists in the **job-processes** and **job-tvobs** of the job. Of each set, there is a subset that is *enabled*; these are the **job-enabled-processes** and **job-enabled-tvobs** of the job. A process's being *enabled* means that whenever the job is told that it may run, that process will be made *active*. The same is true for TVOBs. When the job is told that it may run its enabled processes, it is said to be *process-enabled*; when it is told that it may display its enabled TVOBs, it is said to be *tvob-enabled*. A job can control which of its processes and TVOBs are enabled by means of the functions **process-enable**, **process-disable**, **tvob-enable**, and **tvob-disable**, which are described below.

At any time there is one job which is said to "own the keyboard"; this job is the value of the variable **kbd-job**. When a function calls any of the keyboard input functions (such as **kbd-tyi**), the function will wait until the current job is the **kbd-job** before returning. The reason for this is that while the TV can be split up into areas so that several programs can type on it at once, there is no similar way to split up the keyboard; if several jobs want keyboard input, one of them will get what the user types, and the rest will wait until they become the **kbd-job**.

When Lisp is initialized, one job is created and given the keyboard. The job is given a single, active TVOB, the size of the screen, and a single, active process. It is both process-enabled and TVOB-enabled, so the process and TVOB are both active.

22.4 Controlling Jobs

It should be made easy for the user to control which jobs are process-enabled and which are TVOB-enabled. Unfortunately, the commands to allow easy control of these parameters have not yet been fully developed. This section describes what has been implemented so far, but this will probably change.

The following two simple functions control whether a job is process-enabled or TVOB-enabled.

job-set-process-state *job state*

If *state* is non-**nil**, *job* is made process-enabled; otherwise, it is made process-disabled.

job-set-tvob-state *job state*

If *state* is non-**nil**, *job* is made TVOB-enabled; otherwise, it is made TVOB-disabled. Since this function can change the set of active TVOBs, the caller should follow with a call to **tvob-setup** (see page 207).

There is one job designated as the top-level job, from which other jobs are selected. This job is the value of the variable **si:top-job**. When Lisp is initialized, **si:top-job** is set to the initial job, and usually it is never set again. If this job wants to let some other job run, it uses the function **job-select**, which may be called directly by the user, or by a program's "command interface" function. (The functions **ed**, **edval**, and **edprop** serve this purpose for Eine, and the function **supdup** for the Supdup program.)

job-select *job*

job-select should be called from the top-level job to give the keyboard to *job*. The top-level job is made process-disabled and TVOB-disabled, and *job* is made process-enabled and TVOB-enabled, and is given the keyboard (made to be the **kbd-job**).

When the keyboard belongs to some job other than the top-level job, the "CALL" key is interpreted specially to mean "Return the keyboard to the top-level job." If the user types a "CALL", the current **kbd-job** will be made process-disabled and TVOB-disabled, the top-level job will be made process-enabled and TVOB-enabled, and the top-level job will be given the keyboard. The Control and Meta keys can be used with CALL: Control prevents the current **kbd-job** from being made TVOB-disabled, and Meta stops it from being process-disabled.

22.5 Functions for Manipulating TVOBs.

The following four functions are all used to create TVOBs; they differ primarily in the way the caller specifies the TVOB's area of the screen. To fully specify the area, use **tvob-create-absolute**. If you just want an area of a certain size, but don't care where the area is, use **tvob-create**. If you need at least a certain size but would accept a larger size if the space is available, use **tvob-create-expandable**. If you have a pc ppr (piece of paper) and want to make a TVOB for it, use **tvob-create-for-pc-ppr**.

tvob-create-absolute *x1 y1 x2 y2* &rest *options*

tvob-create-absolute creates and returns a new TVOB. Its fields are set up as follows:

tvob-handler The value of the **:handler** option.

tvob-info The value of the **:info** option.

tvob-job The value of the **:job** option. If it is **t**, the current job is used instead; this is the default. Otherwise it should be **nil** (meaning that the TVOB is not associated with any job), or a job.

tvob-priority
The value of the **:priority** option.

tvob-x1 *x1*.

tvob-y1 *y1*.

tvob-x2 *x2*.

tvob-y2 *y2*.

tvob-screen The value of the **:screen** option, which should be a screen. It defaults to the value of **tv-default-screen**.

tvob-status The value of the **:status** option.

tvob-clobbered-p
nil.

tvob-mouse-handler
The value of the **:mouse-handler** option. If the value is **t**, use the default mouse handler.

The options to **tvob-create-absolute** are:

:handler The **tvob-handler** function.

:info The value to go in the **tvob-info** field.

:job The job with which the TVOB will be associated. **t** means the current job and **nil** means no job. The default is the current job.

:mouse-handler
The **tvob-mouse-handler** function. **nil** means this TVOB doesn't

do anything special with the mouse, and **t** means the **mouse-default-handler** should be used.

:name The print-name of the TVOB.

:priority **t** to make this TVOB more tenacious of its place on the screen.

:screen The screen on which the TVOB will appear. The default is **tv-default-screen**.

tvob-create *x y* &rest *options*

This allocates an area of the screen of width *x* and height *y*, and creates and returns a TVOB with that area. The options are the same as for **tvob-create-absolute**. The screen area of the TVOB will be within the rectangular boundaries described by the **screen-x1**, **screen-y1**, **screen-x2**, and **screen-y2** of the screen on which the TVOB is created. **tvob-create** tries to choose an area that will overlap the fewest interesting TVOBs. Specifically, it tries to stay out of the area used by the exposed TVOB of the highest priority, then that of the exposed TVOB of the second-highest priority, etc. The way priority works is that the exposed tvobs are divided into two groups: those with **tvob-priority** of **t**, and those with **tvob-priority** of **nil**; the former all have higher priority than the latter. Within these two groups, the TVOBs are ordered by their ordering in the list **active-tvobs**. The priority is remembered by the ordering of the list **exposed-tvobs**, of which the first element is the TVOB of *lowest* priority, and the last is the TVOB of *highest* priority.

tvob-create-expandable *min-x min-y* &optional *max-x max-y* &rest *options*

This first finds a *min-x* by *min-y* area of the screen, the same way **tvob-create** does. Then it tries to make that area larger, up to *max-x* by *max-y*, without overlapping any other exposed TVOBs. Otherwise it is like **tvob-create**. *max-x* defaults to the size of the area in which automatic allocation takes place: the difference between the **screen-x2** and **screen-x1** of the screen. *max-y* defaults similarly.

tvob-create-for-pc-ppr *pc-ppr* &rest *options*

If you want to use a pc ppr, you need an associated TVOB in order to get permission for your pc ppr to use the screen. This function takes a pc ppr and creates a TVOB, whose area of the screen is that of the pc ppr. The **tvob-info** will be the pc ppr, the **tvob-handler** a function called **si:pc-ppr-tvob-handler** (which does the right thing for pieces of paper which don't remember what they are displaying and hence cannot **:update**), and the screen used will be the **pc-ppr-screen** of the pc ppr. If you give the **:handler** option, though, it will override the **si:pc-ppr-tvob-handler**. The rest of the options are the same as in **tvob-create-absolute**.

tvob-kill *tvob*

This deactivates *tvob* if it is active, and dissociates it from its associated job (if any).

tvob-enable *tvob*

Enable *tvob*. If *tvob* has no associated job, or if its job is tvob-enabled, activate *tvob*. After making some calls to **tvob-enable** and **tvob-disable**, the caller should call **tvob-setup** (see page 207).

tvob-disable *tvob*

Disable *tvob*. If it was active, deactivate it. After making some calls to **tvob-enable** and **tvob-disable**, the caller should call **tvob-setup** (see page 207).

tvob-setup *no-reselection* &rest *tvobs*

This is the function in charge of keeping the state of the screen and the internal database consistent when tvobs are activated, deactivated, moved, etc. After a program makes some calls to **tvob-enable** and **tvob-disable**, it may have changed the set of *active* TVOBs, and it should call **tvob-setup** to make sure that the set of *exposed* TVOBs is recomputed, and that the right messages are sent to all TVOBs. (If **tvob-enable** etc. did that themselves, then unneccesary redisplay and computation would be unavoidable.) The **job-set-tvob-state** function can also change the set of active TVOBs, and it too should be followed by a call to **tvob-setup**.

tvob-setup looks at its argument, *tvobs*, and at the list of active TVOBs and figures out which TVOBs should be exposed.

First, **tvob-setup** examines the elements of *tvobs*, all of which should be active, and rearranges the order of **active-tvobs**. The TVOBs in *tvobs* are moved to the front of **active-tvobs**, and placed in the order they were given to **tvob-setup**. The first TVOB in *tvobs* is guaranteed to be first in **active-tvobs**. The remaining active TVOBs are moved to the end of **active-tvobs**. Their relative order is not changed. The **job-enabled-tvobs** lists of the TVOB's jobs are similarly reordered.

Next, **tvob-setup** figures out the new subset of the active TVOBs that should be exposed, by walking down the **active-tvobs** list and taking every TVOB that doesn't overlap with some TVOB already on the new **exposed-tvobs** list. Since the new list starts out as **nil**, the first element of **active-tvobs**, which was the second argument to **tvob-setup**, will always be exposed. The exposed list is kept in reverse priority order, as explained under **tvob-create** (see page 206).

Having determined the new set of exposed tvobs, **tvob-setup** sends out **:deselect**, **:deexpose**, and **:expose** commands as needed. It only sends **:deselect** if the selected tvob would no longer be exposed; when it does this, it also sets **selected-tvob** to **nil**. At this point, **exposed-tvobs** is set to its new value. **tvob-select** then sends **:clobber** and **:update** commands to all of the new exposed TVOBs.

Finally, if there is no **selected-tvob**, and *no-reselection* is **nil**, **tvob-setup** tries to choose a new **selected-tvob** by calling the **job-tvob-selector** function of the **kbd-job** (if there is a **kbd-job** and it has a **job-tvob-selector**).

tvob-select *tvob*

This makes *tvob* be the **selected-tvob**. It makes sure that *tvob*'s job is TVOB-enabled, and that *tvob* is exposed. Then it deselects the current **selected-tvob** (if any) and selects *tvob*.

tvob-update

If the variable **tvob-complete-redisplay** is non-**nil**, set it to **nil** and call **tvob-complete-redisplay**. Otherwise send an **:update** message to all exposed TVOBs.

tvob-complete-redisplay *Variable*

Used as a flag by **tvob-update** (see above): if non-**nil**, **tvob-update** should do a **tvob-complete-redisplay**.

tvob-complete-redisplay

Clears the screen, outlines the screen area of partially-exposed enabled TVOBs, and sends **:clobber** and **:update** to all exposed TVOBs.

tvob-clean

This "cleans up" the screen. It sends a **:clean** message to all exposed TVOBs, and clears portions of the screen not occupied by exposed TVOBs.

tvob-command *command tvob* &rest *arguments*

Sends *command* to *tvob*, with the given *arguments*. *command* should be one of the symbols mentioned above (**:set-edges**, **:clobber**, etc.). After sending the command, **tvob-command** updates the *tvob*'s **tvob-status**, **tvob-clobbered-p**, or the screen area (**tvob-x1** et. al.) as appropriate.

In order to preserve consistency, only the **tvob-setup** and **tvob-select** functions should send any of the commands **:select**, **:deselect**, **:expose**, and **:deexpose**; you should never send these yourself.

tvob-under-point *x y* &optional *screen*

Returns the TVOB under the point (*x*,*y*) on *screen*, or **nil** if there is none. If there are several TVOBs at that point, the "top-most" one, i.e. the one which is actually visible, is returned. *screen* defaults to **tv-default-screen**.

22.6 Functions for Manipulating Jobs.

job-create *name*

Creates and returns a job, whose name is *name*. The job is created with no processes and no TVOBs, and its initial **job-process-enabled-p** and **job-tvob-enabled-p** are both **nil**. **job-create** also puts the job on **job-list**.

job-kill *job*

Deactivates and kills all of *job*'s processes and TVOBs, and removes *job* from the **job-list**.

job-list *Variable*

A list of all jobs. See **job-create** and **job-kill**.

job-reset-processes *job*

Disables all of *job*'s enabled processes, and unwinds those processes's stack groups.

job-select *job*

This is meant to be called from the top-level job, which should have the keyboard at the time. It disables the **kbd-job**'s processes and TVOBs, enables those of *job*, and gives *job* the keyboard.

job-return

This is meant to be called from jobs other than the top-level job. It disables the current job's processes and TVOBs, enables those of the top-level job, and gives the top-level job the keyboard. [The job calling it had better have the keyboard.]

23. The TV Display

The principal output device of the Lisp Machine is the TV display system. It is used in conjunction with the keyboard as an interactive terminal, and it can output printed text or graphics. This chapter describes the Lisp functions used to manipulate the TV.

23.1 The Hardware

The Lisp machine display system is a raster-scan, bit-map system. This means that the screen is divided up rectangularly into points. The video signal that enters the TV comes from a memory which has one bit for every point on the screen. This memory is directly accessible to the program, allowing extremely flexible graphics.

The coordinate system normally used has the origin (0,0) at the top left corner of the screen. *X* increases to the right, and *Y* increases downward.

There are currently two TV controllers in use. The 16-bit controller, which is going away, generates industry-standard composite video, allowing a screen size of 454. lines from top to bottom with 576. points on each line. The newer, 32-bit controller, provides various options. With the CPT monitor it generates a black-and-white display of 896. lines with 768. points on each line. Other monitors can also be supported.

One thing to be aware of is that the same fonts cannot be used with both controllers, because the 16-bit controller has its bits reversed.

It is possible to have a display in which there is more than one bit per visible point, allowing gray-scale or color. The set of all bits which contribute to a single point on the screen is called a *pixel*. (The point on the screen itself is also sometimes called a pixel.) Some of the software operates in terms of pixels. Pixels are implemented in an entirely different way in the two controllers. This document doesn't really discuss them yet.

Because of all these options, the Lisp machine system includes ***screen objects***. A screen object contains all the attributes of a particular TV controller and display monitor.

23.2 Screens

There is a type of Lisp object called a *screen*, which is the internal representation for a physical display with someone looking at it. Both microcode and Lisp functions look at screen objects. A screen is a structure which contains the following fields:

screen-name An arbitrary character string which appears in the printed representation of the screen-object.

screen-height
The total height of the screen in bits (raster lines, pixels).

screen-width The total width of the screen in bits (pixels).

screen-x1, screen-x2, screen-y1, screen-y2
The coordinates of a rectangle which is the portion of the screen in which allocation of tvobs may occur. Usually this is the whole screen, but if there is a who-line it is excluded. There could be other reserved areas of the screen.

screen-plane-mask
0 if this screen is on a 32-bit controller. If it is on a 16-bit controller, one of the bits in this mask is on corresponding to the memory "plane" which contains this screen. (For instance, for plane 0 the value of this field would be 1.) Having more than one bit on in this mask is not really supported.

screen-bits-per-pixel
The number of bits in a pixel.

screen-attributes
This is a list of keywords for special features of this screen.

:sideways	The monitor is standing on its left side. The TV routines know how to draw characters on such a screen, given a rotated font, so that the text comes out in the normal orientation.
:color	Has color (not yet implemented).
:gray	Has gray-scale.

screen-font-alist
An a-list that associates from font names to font objects. This is not really used yet.

screen-default-font
The font to be used by default on this screen.

screen-buffer
The address in virtual memory of the video buffer for this screen.

screen-locations-per-line
The number of locations (containing 16 or 32 bits depending on the controller) of the video buffer for a scan line.

screen-buffer-pixel-array
A two-dimensional array of positive integers, which are pixel values. The first subscript is the X coordinate and the second subscript is the Y coordinate.

screen-buffer-halfword-array
A one-dimensional array of 16-bit words of video buffer. This is provided to allow direct manipulation of the video buffer, bypassing the usual microcode primitives. Note that on a 16-bit controller, the

bits in these words are reversed.

tv-default-screen *Variable*

The value of **tv-default-screen** is the "normal" screen where text display happens. Various functions that take a screen as an optional argument default to this.

tv-define-screen *name* &rest *options*

Creates and returns a screen whose name is *name* (a string) and whose attributes are controlled by the options. These attributes has better correspond to an existing hardware screen. *options* is alternating keywords and arguments to those keywords. The following keywords are accepted:

:plane-mask The value of the **screen-plane-mask** field. Defaults to 1. 0 for screens on the 32-bit TV controller.

:height The value of the **screen-height** field. Defaults to 454.

:width The value of the **screen-width** field. Defaults to 576.

:x1 The value of the **screen-x1** field. Defaults to 0.

:y1 The value of the **screen-y1** field. Defaults to 0.

:x2 The value of the **screen-x2** field. Defaults to the width.

:y2 The value of the **screen-y2** field. Defaults to the height, unless the **:who-line-p** option is specified, in which case one line of space is left at the bottom of the screen for the who-line.

:who-line-p **t** to leave space for a who-line, **nil** to make the entire screen available for TVOB allocation. Defaults to **t**.

:buffer A fixnum which is the address of the video buffer containing the bits for this screen. Defaults to the address of the 16-bit TV buffer.

:locations-per-line
The value of the **screen-locations-per-line** field. Defaults from the width, the bits per pixel, and the controller type.

:bits-per-pixel
The value of the **screen-bits-per-pixel** field. Defaults to 1.

:attributes The value of the **screen-attributes** field. Deafults to **nil**.

:font-alist The value of the **screen-font-alist** field. Defaults to **nil**.

:default-font The value of the **screen-default-font** field. Defaults according to the type of controller used.

23.3 Simple Bit Manipulation

Some arrays of numbers exist which allow access to the TV memory. These are regular Lisp arrays and all array operations work on them, but they are set up so that their data storage is actually in the TV memory. These arrays are normally found in fields of a screen object.

screen-buffer-pixel-array is a two-dimensional array. Array element (*x*,*y*) corresponds to the point whose coordinates are *x* and *y*: if the array element is 0, the point is illuminated, and if the element is 1, the point is dark. (The opposite is true when the TV is in reverse-video mode; see below).

The elements of this array are single bits in the usual case, but they can be small positive numbers in the case of gray-scale or color screens.

In the case of a 16-bit TV, this array accesses whichever plane is currently selected.

screen-buffer-halfword-array is a one-dimensional array of 16-bit elements, whose exact interpretation depends on the type of TV screen. Certain programs use this to access the TV buffer memory.

It is possible to do anything to a TV screen, albeit slowly, using the above two arrays. However, for efficiency several microcode primitives are provided which perform certain common operations at much higher speed, typically close to the maximum speed of the memory. Most programs use these microcode primitives or the higher-level functions built on them rather than accessing the TV buffer memory directly. The remainder of this chapter describes these facilities.

23.4 Fonts

A font is a set of related characters. It is represented by an array (of type **art-1b**) which contains the bit patterns used to actually draw the characters. The leader of that array contains other required information such as character widths, height, bookkeeping information, etc.

There is a microcode entry for drawing characters, which understands the structure of fonts. It exists so as to make character drawing as fast as possible. User functions do not call the microcode entry directly, as it is rather kludgey, and handles only the easy cases. Instead the TV routines do all the necessary calls.

A font usually contains 128 characters. The widths may be variable, but the height is always fixed (characters need not actually have ink all the way from the top to the bottom of the height, but the distance between lines is fixed for each font). There are special provisions for fixed-width fonts to save space and time. There is a thing called the baseline, which is a certain vertical position in each character. For example, the baseline touches the bottom of the legs of a capital A, and passes through the stem of a lower-case p. When several fonts are used together, all the baselines are made to line up.

The way characters are drawn is a little strange (it is done this way for speed). There is a thing called a *raster element*, which is a row of 1-bits and 0-bits. A character is drawn by taking a column of raster elements, (making a rectangle) and OR'ing this into the bit-map memory. A raster element can be at most 16 bits wide for hardware reasons, so for large characters it may take several side-by-side columns to draw the character. The font is stored with several raster elements packed into each 32-bit word. The width of a raster element is chosen to give maximum packing, and depends on the font. The reason for the existence of raster elements is to decrease the number of memory cycles by processing several bits at a time.

The structure of the array leader of a font is defined by **defstruct** macros. Here we list the element names and what they are for. This structure is not guaranteed not to be changed in the future, however the macros are automatically made available to user programs.

font-name A symbol, in the **fonts** package, whose value is this font. This symbol also appears in the printed representation.

font-char-height
Height of the characters in this font (with a VSP of 0, this is how far apart the lines would be.)

font-char-width
Width of the characters if this is a fixed-width font, i.e. how far apart successive characters are drawn. Otherwise contains the width of "space".

font-raster-height
Number of raster lines of "ink" in a character (often the same as **font-char-height**).

font-raster-width
Width of a raster element.

font-rasters-per-word
Number of elements packed per word (used when accessing the font.)

font-words-per-char
Number of words needed to hold one column of elements.

font-baseline Number of raster lines down from the top of the character cell of the position to align.

font-char-width-table
nil for fixed width fonts. Otherwise, contains the 128-long array of character widths.

font-left-kern-table
nil for non-kerned fonts. Otherwise, contains the 128-long array of left-kerns. This is the amount (positive or negative) to back up the X position before drawing the character.

font-indexing-table
nil for narrow fonts which only take one column of raster elements to draw.

Otherwise, contains a 129-long array which determines what columns of the font to draw for that character as follows: for character *i*, draw columns *indexingtable*(*i*) through *indexingtable*(*i*+1)-1 inclusive. Note that 2 of the above 3 arrays only contain small positive numbers, so they are usually of type **art-16b** or **art-8b** to save space.

font-next-plane
nil usually. For multi-plane fonts, contains the font for the next higher plane. This field is obsolete and no longer supported.

font-blinker-width
Default width for blinkers.

font-blinker-height
Default height for blinkers.

The data part of a font array contains an integral number of words per character (per column in the case of wide characters that need an indexing table). Each word contains an integral number of raster elements, left adjusted and processed from left to right. All 32 bits of each element in this array are used. For easiest processing by Lisp programs, it should be of **art-1b** array type.

The exact format of the data part of a font array depends on whether the font is intended to be used with a 16-bit TV controller or a 32-bit controller. In the 32-bit case, the bits are displayed from right to left. The maximum width of a raster element is 32. bits; use of the **font-indexing-table** is required if characters are wider than this. If there is more than one raster element per word, the elements are displayed from right to left. In the 16-bit case, the bits and raster elements are displayed from left to right, and the maximum width of a raster element is 16. bits.

23.5 TVOBs

[Here explain what TVOBs are, how they differ from pieces of paper, what you use them for, and point to JOBSYS chapter.] Until this is written, see page 199.

23.6 Pieces of Paper

A *piece of paper* is something on which you draw characters. It is displayed on a certain rectangular portion of a screen. It remembers what fonts to use, where to display the next character, how to arrange margins and spacing, and what to do when certain special conditions arise. It optionally displays a blinking cursor (or several of them).

All character-drawing in the Lisp Machine system is accomplished with pieces of paper. One thing to note is that pieces of paper do not remember the characters you draw on them, except by making dots on the TV screen. This means that if one piece of paper overlays another, or if the screen is cleared, the contents of the first is lost. A higher-level facility (e.g. editor buffers) must be used if the characters are to be remembered. The

abbreviation "pc ppr" is often used for "piece of paper".

A piece of paper is represented as an ordinary array whose elements are named by the following accessor macros. These are automatically available to the user, but should not normally be used as they are not guaranteed to remain unchanged, and often contain internal values which are made into more palatable form by the interface functions. All screen coordinates in this structure are absolute screen coordinates; the user interface functions convert these into coordinates which are relative to the margins of the piece of paper.

pc-ppr-name An arbitrary string which appears in the printed representation.

pc-ppr-screen
The screen-object representing the screen on which this pc ppr displays.

pc-ppr-top The raster line number of the topmost screen line in this pc ppr.

pc-ppr-top-margin
The raster line number of the topmost screen line used to draw characters. The difference between **pc-ppr-top-margin** and **pc-ppr-top** is the size of the top margin.

pc-ppr-bottom
The raster line number of the screen line just below this pc ppr.

pc-ppr-bottom-margin
The raster line number of the screen line just below the bottommost point on which a character can be drawn. The difference between **pc-ppr-bottom** and **pc-ppr-bottom-margin** is the size of the bottom margin.

pc-ppr-bottom-limit
The lowest raster line to which the cursor may be positioned. This is a suitable value to prevent excursion below the bottom margin.

pc-ppr-left The bit number of the leftmost bit in the pc ppr's screen area.

pc-ppr-left-margin
The bit number of the leftmost bit used to draw characters. The difference between **pc-ppr-left-margin** and **pc-ppr-left** is the left margin.

pc-ppr-right The bit number of the bit just to the right of the pc ppr's screen area.

pc-ppr-right-margin
The bit number of the bit just to the right of the portion of the pc ppr in which characters may be drawn. The difference between **pc-ppr-right** and **pc-ppr-right-margin** is the right margin.

pc-ppr-right-limit
The rightmost bit position to which the cursor may be positioned. This is set to a suitable value to prevent excursion past the right margin.

pc-ppr-current-x
The X position of the left edge of the next character to be drawn, i.e. the

X coordinate of the cursor position.

pc-ppr-current-y
The Y position of the top edge of the next character to be drawn, i.e. the Y coordinate of the cursor position.

pc-ppr-flags A fixnum containing various bit flags, as follows:

pc-ppr-sideways-p
0 normally. 1 if the pc ppr is on a sideways screen, so the X and Y coordinates should be interchanged before calling the microcode.

pc-ppr-exceptions
Non-zero if any special conditions which prevent typeout are active. The conditions are:

pc-ppr-end-line-flag
1 if **pc-ppr-current-x** is greater than **pc-ppr-right-limit**. The default response to this is to advance to the next line.

pc-ppr-end-page-flag
1 if **pc-ppr-current-y** is greater than **pc-ppr-bottom-limit**. The default response to this is to return to the top of the pc ppr.

pc-ppr-more-flag
1 if "more-processing" must happen before the next character can be output. The default response to this is to display "**MORE**" and await keyboard input.

pc-ppr-output-hold-flag
1 if some higher-level function has decided that output is not to be allowed on this pc ppr. For example, its region of the screen might be in use for something else. When this is seen a function specified when the pc ppr was created is called.

pc-ppr-more-vpos
Y passing here triggers "more processing" by setting **pc-ppr-more-flag**. Add 100000 to this field to delay until after screen wraparound. Store **nil** here to inhibit more processing.

pc-ppr-baseline
The number of raster lines from the top of the character cell (**pc-ppr-current-y**) to the baseline.

pc-ppr-font-map
An array of fonts. Normally a font-change command specifies a code number, which is looked up in this array to find what font to actually use. Font 0 is the "principal" font. The array is usually 26. long.

pc-ppr-current-font
The font which is currently selected.

pc-ppr-baseline-adj
Y offset for current font to align baseline. This is the difference between the **pc-ppr-baseline** and the font's baseline.

pc-ppr-line-height
The number of raster lines per character line.

pc-ppr-char-width
A character width which is just used for old-style space/backspace/tab operations and for blinkers.

pc-ppr-char-aluf
ALU function for drawing characters. The default is the value of **tv-alu-ior**.

pc-ppr-erase-aluf
ALU function for erasing characters/lines/whole pc ppr. The default is the value of **tv-alu-andca**.

pc-ppr-blinker-list
(Possibly null) list of blinkers on this pc ppr.

pc-ppr-end-line-fcn
Function called when typeout is attempted with **pc-ppr-end-line-flag** set. The default is to wrap around to the next line.

pc-ppr-end-screen-fcn
Function called when typeout is attempted with **pc-ppr-end-page-flag** set. The default is to wrap around to the top margin.

pc-ppr-output-hold-fcn
Function called when typeout is attempted with **pc-ppr-output-hold-flag** set. The default is to wait for the flag to be cleared by some other process.

pc-ppr-more-fcn
Function called when typeout is attempted with **pc-ppr-more-flag** set. The default is to type **MORE** and await typein.

23.6.1 Simple Typeout

tv-tyo *pc-ppr char*
Draws a printing character, or executes a special format character. The character is drawn at the current cursor position, in the current font, and the cursor position is shifted to the right by the width of the character. The following format effectors are recognized:

200 Null. Nothing happens.

210 Backspace. The cursor is moved left the width of a space. At the beginning of a line it sticks.

211 Tab. The cursor is moved right to the next multiple of 8 times the width of a space.

215 Carriage return. The cursor is advanced to the beginning of the next line, and that line is erased. More-processing and screen wrap-around may be triggered.

240-247
Font change. The low 3 bits are the font number.

Other non-printing characters are displayed as their name enclosed in a box. These displays are quite wide and currently don't bother to respect the right margin.

tv-beep
This function is used to attract the user's attention. It flashes the screen and beeps the beeper. Doesn't really have that much to do with TVs.

tv-beep *Variable*
If the value of **tv-beep** is non-**nil**, the **tv-beep** function doesn't flash the screen, it only sounds a beep. The initial value is **nil**.

si:tv-move-bitpos *pc-ppr delta-x delta-y*
Move current X, current Y on piece of paper, keeping inside boundaries. This function is called from many others. It is the central place to keep track of edges, automatic wrap-around, **MORE** processing, etc. It will set the **pc-ppr-exceptions** flags as necessary.

si:tv-exception *pc-ppr*
This function is called by various TV functions when they encounter a **pc-ppr-exceptions** flag which they care about (for example, **tv-crlf** does not care about **pc-ppr-end-line-flag**). The appropriate function (stored in the pc ppr) is called. It is up to that function to correct the condition and clear the exception flag.

If you want to supply your own exception-handling function for a piece of paper, you would be well-advised to read the corresponding system default function first. They need to do non-obvious things in some cases.

si:tv-end-line-default *pc-ppr*
This is the default end-of-line function, called if an attempt is made to display a character when the cursor is off the end of a line. It essentially just does a crlf.

si:tv-end-screen-default *pc-ppr*
This is the default end-of-screen function, called when an attempt is made to display a character when the cursor is off the bottom of the pc ppr. It wraps around to the top of the pc ppr. Note that more-processing is separate from and unrelated to end-of-screen processing.

si:tv-more-default *pc-ppr*
This is the default more processor. It types out **MORE**, waits for input, and decides where the next "more" should happen.

tv-note-input
The purpose of **tv-note-input** is to prevent "more"s from happening during normal interactive usage, since typeout is frequently pausing for user input anyway, and presumably the user is keeping up in his reading. This function is called by the keyboard handler when a process hangs waiting for input. **tv-note-input** arranges (on each active pc ppr) for a more not to happen until the current line is reached again; except, if this line is far from the bottom, we prefer to more at the bottom before wrapping around. This makes moreing usually happen at the bottom.

23.6.2 Cursor Motion

Note that the "cursor" is the x,y position where the top-left corner of the next character printed will be placed. (This is not strictly true because there is base-line adjustment and kerning.) The cursor doesn't necessarily have a corresponding blinker; this is under the control of the user program.

Many of these functions are not used by real Lisp Machine code, but are present for completeness and to aid compatibility with ITS I/O. On the other hand, some are heavily used.

tv-home *pc-ppr*
Home up to the top-left corner. Usually you then want to do a **tv-clear-eol**.

tv-home-down *pc-ppr*
Home down the cursor to the bottom-left corner (the beginning of the last line in the pc ppr).

tv-crlf *pc-ppr*
Advance to the beginning of the next line, and erase its previous contents.

tv-space *pc-ppr*
Space forward.

tv-backspace *pc-ppr*
Space backward. Not too useful with variable-width fonts.

tv-tab *pc-ppr*
Tab. Spaces forward to the next multiple of 8 times the width of space.

tv-set-font *pc-ppr font*

This is the common internal routine for changing what font a piece of paper is to print with. It does some bookkeeping, such as adjusting the baseline. It is OK to set the font to one which is not in the font map, however this won't change the line-spacing, which is initially set up according to the tallest font in the font map.

tv-set-cursorpos *pc-ppr x y*

Sets the "cursor" position of the piece of paper in raster units (*not* character units). *x* and *y* are relative to the margins of the pc ppr.

tv-read-cursorpos *pc-ppr*

Returns two values, the X and Y coordinates of the cursor. These are relative to the margins of the pc ppr.

23.6.3 Erasing, etc.

tv-clear-char *pc-ppr*

Clear the current character position. In a variable-width font, the width of space is used, which isn't likely to be the right thing.

tv-clear-eol *pc-ppr*

Clear from current position to end of line.

tv-clear-eof *pc-ppr*

Clear from current position to end of piece of paper.

tv-clear-pc-ppr *pc-ppr*

Clear whole piece of paper.

tv-clear-pc-ppr-except-margins *pc-ppr*

Clear all of *pc-ppr* except the margins, which are unaffected. This is useful if the margins contain decorative graphics such as outlines.

tv-clear-screen &optional *screen*

Clears the entire screen, and tells the who-line it has been clobbered. *screen* defaults to **tv-default-screen**.

tv-delete-char *pc-ppr* &optional *(char-count 1)*

Deletes the specified number of character positions immediately to the right of the cursor, on the current line. The remainder of the line slides to the left, and blank space slides in from the right margin.

tv-insert-char *pc-ppr* &optional *(char-count 1)*
Inserts the specified number of blank character positions immediately to the right of the cursor, on the current line. The remainder of the line slides to the right, and anything that goes off the right margin is lost.

tv-delete-line *pc-ppr* &optional *(line-count 1)*
Deletes the specified number of lines immediately at and below the cursor. The remaining lines of the piece of paper slide up, and blank spaces slides in from the bottom margin.

tv-insert-line *pc-ppr* &optional *(line-count 1)*
Inserts the specified number of blank lines at the cursor. The remaining lines of the piece of paper slide down, and anything that goes off the bottom margin is lost.

tv-black-on-white &optional *screen*
Makes the hardware present the screen as black characters on a white background. (Presently, the *screen* argument can also be a plane-mask.)

tv-white-on-black &optional *screen*
Makes the hardware present the screen as white characters on a black background. (Presently, the *screen* argument can also be a plane-mask.)

tv-complement-bow-mode &optional *screen*
Makes the hardware present the screen in the reverse of its current mode. (Presently, the *screen* argument can also be a plane-mask.)

tv-white-on-black-state &optional *screen*
Returns **:white** if the screen is currently presented as white-on-black, or **:black** if it is currently presented as black-on-white. The *screen* argument can also be a plane-mask. If more than one bit is on in the plane-mask, and not all the planes are in the same state, **:both** is returned.

23.6.4 String Typeout

tv-string-out *pc-ppr string* &optional *(start 0) end*
Print a string onto a piece of paper. Optional starting and ending indices may be supplied; if unsupplied, the whole string is printed. This is basically just iterated **tv-tyo**, except in the case of simple fonts it runs much faster by removing a lot of overhead from the inner loop.

tv-line-out *pc-ppr string* &optional *(start 0) end*
This variant of **tv-string-out** is used by the editor's display routines to output one line. The argument is a string of either 8-bit or 16-bit characters (usually this is an EINE "line", but the leader is not touched except for the fill pointer.) The high 8 bits (**%%ch-font**) of each character are the index into the font map for the font in which that character is to be displayed. 8-bit chars use font 0. There are optional starting and ending indices; if these are omitted the whole string is specified. If

during printing the cursor runs off the end of the line, typeout stops and the index of the next character to be output is returned. At this point, the **pc-ppr-end-line-flag** is 1 and the cursor is off the end of the line. If the whole string is successfully output, **nil** is returned, and the pc ppr is pointing somewhere in the middle of the line.

tv-string-length *pc-ppr string* &optional *(start 0) end stop-x*

Compute the display-length of a string, which is the sum of the widths of the printing characters in it. Newline characters are ignored. Tab characters act as if the string starts at the left margin. *pc-ppr* is used mainly for its font map. *start* and *end* allow you to process a substring. *stop-x*, if non-**nil**, is a tv-length at which to stop. The index in the string of the character after the one which exceeded the *stop-x* is returned as the second value.

The first returned value is the *x*-position reached, i.e. the tv-length of the string. The second returned value is the next index in the string, which is *end* if *stop-x* was not supplied.

Contrast **tv-compute-motion**, which does a two-dimensional computation taking line-length into account.

tv-compute-motion *pc-ppr x y string* &optional *(start 0) end (cr-at-end-p nil) (stop-x 0) stop-y*

Compute the motion that would be caused by outputting a string. This is used by the editor to aid in planning its display, to compute indentations with variable width fonts, to position the cursor on the current character, etc. Note that this does not use the "case shift" flavor of font hacking. Instead, it uses the 16-bit-character flavor that the editor uses. This means that if you give it an ordinary 8-bit string it will be assumed to be all in font 0.

The arguments are: the piece of paper, the X and Y position to start at (**nil**s here use the current position of the pc ppr), the string, and optionally the starting and ending indices, a flag saying to fake a crlf at the end of the string, and two additional arguments which are the X and Y to stop at; if not given these default to the end of the screen. Returns 3 values: final-X, final-Y, and an indication of how far down the string it got. This is **nil** if the whole string (including the fake carriage return, if any) was processed without reaching the stopping point, or the index of the next character to be processed when the stopping point was reached, or **t** if the stopping point was reached after the fake carriage return.

tv-char-width *pc-ppr char*

Returns the width of the character, if displayed in the font current in the pc-ppr. The width of backspace is negative, the width of tab depends on the pc ppr's cursor position, and the width of carriage return is zero.

23.6.5 More Processing

More processing is a flow control mechanism for output to the user. Lisp machine more processing is similar to more processing in ITS. The problem that more processing solves is that displayed output tends to appear faster than the user can read it. The solution is to stop just before output which has not been read yet is wiped out, and display "**MORE**". The user then reads the whole screen and hits space to allow the machine to continue output. More processing normally occurs one line above where the cursor was when the machine last waited for user input; however, it tries to do an extra **MORE** at the bottom of the pc ppr, so as to get into a phase where the **MORE** always appears at the bottom, which is more aesthetic.

23.6.6 ALU Functions

Some TV operations take an argument called an *ALU Function*, which specifies how data being stored into the TV memory is to be combined with data already present. The ALU function is OR'ed directly into a microinstruction, so specifying a value other than one of those listed below may produce unexpected disasters. The following special variables have numeric values which are useful ALU functions.

tv-alu-ior *Variable*
Inclusive-OR. Storing a 1 turns on the corresponding bit, otherwise the bit in TV memory is left unchanged.

tv-alu-xor *Variable*
Exclusive-OR. Storing a 1 complements the corresponding bit, otherwise the bit in TV memory is left unchanged.

tv-alu-andca *Variable*
AND-with-complement. Storing a 1 turns off the corresponding bit, otherwise the bit in TV memory is left unchanged.

tv-alu-seta *Variable*
Bits are simply stored, replacing the previous contents. With most functions, this is not useful since it clobbers unrelated bits in the same word as the bits being operated on. However, it is useful for bitblt.

23.6.7 Blinkers

A *blinker* is an attention-getting mark on the screen. Often, but not always, it will blink. The most common type is a character-sized rectangle which blinks twice a second, but several other types exist, and it is easy for the user to define new ones. Often a piece of paper will have an associated blinker which shows where the next character will be drawn. A blinker can be on top of a character, and the character will still be visible. This done by XORing the blinker into the TV memory. Synchronization between pieces of paper and blinkers is provided so that when characters are being drawn on the screen, blinkers are turned off to prevent the picture from being messed up. (This is called "opening" a piece of paper, and should be invisible to the user.)

A blinker is an array, described as follows:

tv-blinker-x-pos
: X position of the left edge of the blinker. **nil** if the blinker should follow the **tv-blinker-pc-ppr**'s current X and Y.

tv-blinker-y-pos
: Y position of the top edge of the blinker.

tv-blinker-pc-ppr
: Pc ppr the blinker is associated with. **nil** for a ***roving blinker***, which can go anywhere.

tv-blinker-screen
: The screen on which the blinker is displayed.

tv-blinker-visibility
: **nil** invisible, **t** visible, **blink** blinking.

tv-blinker-half-period
: Time interval in 60ths of a second between changes of the blinker.

tv-blinker-phase
: **nil** means not visible, anything else means visible in some form. A complementing blinker has only two phases, **nil** and **t**, but provision is made for blinkers which go through an elaborate sequence of states.

tv-blinker-time-until-blink
: Time interval in 60ths of a second until the next change. The scheduler decrements this 60 times a second if the **tv-blinker-visibility** is **blink**. If it reaches zero, the blinker is blinked. If this field is **nil**, the blinker is not to be looked at by the scheduler.

tv-blinker-function
: The function to call to blink the blinker. The next two fields are for its use. The arguments to the function are the blinker, an operation code, the **tv-blinker-x-pos**, and the **tv-blinker-y-pos**. The operation codes are **nil** to make the blinker invisible, **t** to make it visible, and **blink** to blink it. When this function is called, interrupts have been disallowed and the proper screen

has been selected. For additional conventions, read the function **tv-blink**.

tv-blinker-width
Width in bits of area to complement if **tv-rectangular-blinker**. For other blinker types, miscellaneous data.

tv-blinker-height
Height in raster lines of area to complement if **tv-rectangular-blinker**. For other blinker types, miscellaneous data.

tv-blinker-sideways-p
t => interchange X and Y before calling microcode.

tv-set-blinker-cursorpos *blinker x y*
Set the cursor position of a blinker. If *blinker* is a roving blinker, *x* and *y* are absolute coordinates. Otherwise, they are relative to the margins of *blinker*'s piece of paper. If this blinker was following the pc ppr's cursor, it won't any more.

tv-read-blinker-cursorpos *blinker*
Read the cursor position of a blinker, returning two values, X and Y. If the blinker is not roving, these are relative to the margins of its piece of paper.

tv-set-blinker-visibility *blinker type*
Carefully alters the visibility of a blinker. *type* may be **nil** (off), **t** (on), or **blink**.

tv-set-blinker-function *blinker function* &optional *arg1 arg2*
Carefully alters the function which implements a blinker. *arg1* and *arg2*, if supplied, change **tv-blinker-height** and **tv-blinker-width**, which are really just general arguments to the function.

tv-set-blinker-size *blinker width height*
Carefully changes the size of a blinker, consulting the function which implements it if that function has a **tv-set-blinker-size-function** property.

23.7 Graphics

tv-draw-line *x1 y1 x2 y2 alu screen*
Draws a straight line between the points (*x1*,*y1*) and (*x2*,*y2*), merging the line into the existing contents of the screen with the specified alu function. This is a fast micro-coded function.

bitblt *alu width height from-array from-x from-y to-array to-x to-y*
This function moves a portion of one two-dimensional numeric array into a portion of another, merging them under the control of a specified alu function. It has several applications, including shifting portions of the TV screen around (use the **screen-buffer-pixel-array**), saving and restoring portions of the TV screen, writing half-tone and stipple patterns into the TV screen, and general array-moving.

bitblt operates on a rectangular region of *to-array* which starts at the coordinates (*to-x*,*to-y*) and has extent (**abs** *width*) in the *X* direction and (**abs** *height*) in the *Y* direction. An error occurs if this region does not fit within the bounds of *to-array*. Note that the coordinates and the *height* and *width* are in terms of array elements, not bits, although the actual operation is done bitwise. *from-array* needn't be as big as the specified region; conceptually, **bitblt** replicates *from-array* a sufficiently-large number of times in both *X* and *Y*, then picks out a rectangular region containing the same number of bits as the destination region, starting at the coordinates (*from-x*,*from-y*). **bitblt** combines these two regions under control of *alu*. The "A" operand is the *from-array*, thus an alu function of **tv-alu-seta** copies the *from-array*, ignoring the previous contents of the selected region of the *to-array*.

The specified *X* and *Y* coordinates are always the upper-left corner (minimum coordinate values) of the selected region.

bitblt normally works in a left-to-right and top-to-bottom order, that is with increasing coordinate values. When using overlapping *from* and *to* arrays, for instance when shifting a portion of the TV screen slightly, it may be necessary to work in one of the other three possible orders. This is done using the sign of the *width* and *height* arguments. If *width* is negative, decreasing *X* coordinates are used, and if *height* is negative, decreasing *Y* coordinates are used.

For the sake of efficiency, **bitblt** requires that the *from-array* and *to-array* have word-aligned rows. This means that the first dimension of these arrays must be a multiple of 32. divided by the number of bits per array-element. All TV screen arrays are forced by hardware to satisfy this criterion anyway.

23.8 The Who Line

The *who line* is a line at the bottom of the screen which contains information on what the program is currently doing. The who line has its own pc ppr and is updated whenever the software goes into an I/O wait. In addition, there are two short line segments (called *run lights*) at the bottom of the screen which are controlled by the microcode and by the scheduler. The one on the right lights up when the machine is running (not waiting not paging), and the one on the left lights up when the disk is running (paging).

tv-who-line-update &optional *state*

This function updates all fields of the who-line which have changed. It is called from various functions which change the "state of the machine" as perceived by the user. The optional argument, *state*, is a string to be displayed in the state field. If *state* is not specified, the value of **tv-who-line-run-state** is used, which is usually "RUN".

tv-who-line-list *Variable*

The value of **tv-who-line-list** is a list of who-line fields. Each field is a list; the first four elements of the list constitute a structure containing the following components:

tv-who-line-item-function
A function to call, given the field as its argument. The function is supposed to update the field of the who-line if it has changed. The list elements of the field after the first four are for the use of this function.

tv-who-line-item-state
If **nil**, the who-line has been clobbered (e.g. by clearing of the screen) and the field must be updated. Otherwise, this is used by the function in an unspecified way to remember its previous state.

tv-who-line-item-left
The bit position of the left edge of the portion of the who-line containing this field.

tv-who-line-item-right
The bit position (+1) of the right edge of the portion of the who-line containing this field.

The initial **tv-who-line-list** is set up to display the time, the name of the person logged-on to the machine, the current package, the "state" of a certain selected process, and name and position of the current input file.

tv-who-line-prepare-field *field*

This is called by **tv-who-line-item-functions** in preparation for redisplay of a who-line field. The portion of the screen on which the field displays is erased and the **tv-who-line-pc-ppr**'s cursor is set to the beginning of the field.

tv-who-line-string *field*

This is a useful function to put into a who-line field. It displays the string which is the value of the symbol which is the fifth element of the field, if it is not **eq** to the string previously displayed.

tv-who-line-pc-ppr *Variable*

The value of **tv-who-line-pc-ppr** is a piece of paper which is used to display the characters in the who-line.

tv-who-line-stream *Variable*

The value of **tv-who-line-stream** is a stream whose output displays on the **tv-who-line-pc-ppr**.

tv-who-line-process *Variable*

The value of **tv-who-line-process** is the process whose state is to be displayed in the who-line. **process-wait** calls **tv-who-line-update** if this is the current process. **tv-who-line-process** is normally the main process of the job which owns the keyboard.

tv-who-line-run-state *Variable*

Normally the string "RUN". This is what appears in the wholine when the machine isn't waiting for anything.

tv-who-line-run-light-loc *Variable*

Unibus address of the TV memory location used for the run-light.

tv-who-line-state *Variable*

This is a special variable which exists inside of **tv-who-line-update.**

23.9 Microcode Routines

tv-select-screen *screen*

This microcode primitive selects a screen for use by the **tv-draw-char** and **tv-erase** functions. It sets up microcode variables and hardware registers. Note that this state is not preserved through process switching, so this primitive should only be called with **inhibit-scheduling-flag** bound to **t**, which is normally desired for other reasons anyway.

tv-select-screen should also be used before referencing the TV arrays, such as the **screen-buffer-pixel-array**, if a 16-bit TV controller is being used.

tv-draw-char *font-array char-code x-bit-pos y-bit-pos alu-func*

The *x-bit-pos* and *y-bit-pos* are of the top left corner of the character. (0,0) is the top left corner of the screen. **tv-draw-char** extracts the raster elements for one character (or one column of a wide character) and displays them at the indicated address in the currently-selected plane, using the indicated ALU function to combine them with the bits already there. Note that this function does not know anything about pieces of paper; no pc ppr handling is in microcode.

tv-erase *width height x-bit-pos y-bit-pos alu-func*

This function is in microcode. *width* and *height* are in bits, and should be fixnums. A rectangle of the indicated size, of all 1s, is created and merged into the rectangle of TV memory in the currently-selected plane whose top left corner is at (*x-bit-pos*,*y-bit-pos*), using the specified *alu-func*. Usually the ANDCA function is used for erasing, but XOR is used for the blinking cursor etc. Note that *width* and *height* must be greater than zero.

tv-draw-line *x0 y0 x1 y1 alu-func screen*

This function is in microcode. A straight line is drawn from the point (*x0,y0*) to the point (*x1,y1*). These points had better not lie outside the screen. The bits that form the line are merged into the screen with the specified alu function. **tv-select-screen** is applied to *screen* before the line is drawn.

23.10 Opening a Piece of Paper

Before a piece of paper can be manipulated, any blinkers which may intercept it must be turned off (i.e. their **tv-blinker-phase** must be **nil**). The operation of assuring this is called *opening* the piece of paper. Similarly, before a blinker's location, size, shape, visibility, or other attributes can be changed, it must be opened, that is made to have no visible effect on the screen.

Once a blinker has been opened, we must make sure that the clock function, which implements the blinking, does not come in and turn the blinker back on. This is done in the simplest possible fashion, by binding the **inhibit-scheduling-flag** non-**nil**, which causes the microcode not to switch to another process. [In the present system processes are never interrupted, not even by the clock, and this variable is ignored.] This also prevents any other process from coming in and messing up the piece of paper by trying to type on it at the same time.

Once we are done with a blinker or piece of paper, and don't need to have it opened any more, we want the blinkers to reappear. It looks best if a blinker reappears right away, rather than at the next time it would have blinked. However, for efficiency we don't want to disappear and reappear the blinker every time a TV operation is performed. Rather, if a program is doing several TV operations right in a row, the first one will turn off the blinkers, the succeeding ones will notice that the blinkers are already off, and then soon after the sequence is completed the blinker will come back on. This is implemented by having the next clock interrupt after we get out of the TV code turn the blinker on.

tv-prepare-pc-ppr *Macro*

The form (**tv-prepare-pc-ppr** (*pc-ppr*) *form1 form2* ...) opens the piece of paper which is the value of the variable *pc-ppr* and evaluates the forms with it open. This macro contains all the knowledge of how to open a pc ppr, including disabling interrupts, finding and opening the blinkers, and selecting the proper screen.

tv-open-blinker *blinker*

The specified blinker is temporarily turned off; the next clock interrupt when **inhibit-scheduling-flag** is **nil** will turn it back on.

tv-open-screen

Opens all the visible blinkers, preparatory to arbitrary munging of the screen, for instance picture drawing.

tv-blink *blinker type*

The function to blink a blinker. *type* is one of the symbols **nil** (off), **t** (on), or **blink**. **tv-blink** checks *type*, selects the proper screen, digs up the blinker position out of the pc ppr if necessary, and calls the blinker's function to do the actual display.

tv-rectangular-blinker *blinker type x y*

A **tv-blinker-function** function for rectangular blinkers (the default). Ignores *type*, just complements.

tv-hollow-rectangular-blinker *blinker type x y*

Function for hollow rectangles.

tv-character-blinker *blinker type x y*

Function for blinkers defined by a character. Arg1 ("height") is the font, and arg2 ("width") is the character in the font. The character is XORed in and out as the blinker blinks.

23.11 Creating Pieces of Paper and Blinkers

tv-define-pc-ppr *name font-map* &rest *options*

This function creates a returns a piece of paper. Keyword arguments allow the user to specify some of the many attributes of the piece of paper and leave the remainder to default. *name* is just a string which is remembered in the pc-ppr and appears in its printed representation. *font-map* may be either a list or an array of fonts; or it may be **nil**, which causes the font map to be taken from the screen's default. The remaining arguments are alternating keywords (which should be quoted) and values for those keywords. For example,

```
(setq foo (tv-define-pc-ppr "foo" (list fonts:tvfont)
                            ':top 300
                            ':bottom 400))
```

Valid option keywords are:

:screen The screen on which the piece of paper is to display. The default is **tv-default-screen**.

:top Raster line number of highest line in the pc ppr. Defaults to **screen-y1** of the specified screen, the top.

:bottom Raster line number + 1 of lowest line in the pc ppr. Defaults to **screen-y2** of the specified screen, just above the who line (if there is one) at the bottom of the screen.

:left Raster point number of left edge of pc ppr. Defaults to **screen-x1** of the specified screen, the left edge.

:right ter point number + 1 of right edge of the pc ppr. Defaults to screen-x2 of the specified screen, the right edge.

:blinker-p **t** if this pc ppr should have a blinker on its cursor, **nil** if the cursor should be invisible. Default is **t**.

:activate-p **t** if this pc ppr should be initially active. Active means that its blinkers can blink. The default is **t**.

:reverse-video-p
t if this pc ppr should be in the inverse of the normal black-on-white mode. This works by changing **pc-ppr-char-aluf** and **pc-ppr-erase-aluf**. Default is **nil**.

:more-p **t** if this pc ppr should have more processing. Default is **t**.

:vsp Number of raster lines between character lines. This is added to the maximum height of the fonts in the font map to get the height of a line in this pc ppr. The default is 2.

:left-margin Amount of unused space at the left edge of the pc ppr. The default is 0.

:top-margin Amount of unused space at the top. The default is 0.

:right-margin Amount of unused space at the right. The default is 0.

:bottom-margin
Amount of unused space at the bottom. The default is 0.

:end-line-fcn A function which is invoked when typeout reaches the end of a line. The default is one which wraps around to the next line.

:end-screen-fcn
A function which is invoked when typeout reaches the bottom of the pc ppr. The default is one which wraps around to the top.

:output-hold-fcn
A function which is invoked when typeout encounters the output-hold flag. The default is one which waits for some other process to clear the flag.

:more-fcn A function which is invoked when more processing is necessary. The default is one which types **MORE** and waits for the user to hit a character, then ignores that character and continues typing.

:blink-fcn The function to implement the blinker if **:blinker-p** is not turned off. The default is **tv-rectangular-blinker**.

:sideways-p **t** means the monitor is standing on its left side instead of its bottom; change things around appropriately. The default comes from the specified screen.

:integral-p **t** means that the piece of paper should be forced to be an integral number of lines high; it will be made slightly smaller than the specified size if necessary. The default is **nil**.

:font-map Set the font-map. This is intended to replace the passing in of the font-map as the second argument.

tv-define-blinker *pc-ppr* &rest *options*

Define a blinker on a piece of paper. The options are similar in syntax to those in **tv-define-pc-ppr**. Valid options are:

:height Number of raster lines high. The default comes from the first font in the pc ppr's font map.

:width Number of raster points wide. The default comes from the first font in the pc ppr's font map.

:function The function to implement the blinker. The default is **tv-rectangular-blinker**.

:arg1 Another name for **:width**. Use this with **:functions** which don't interpret their first "argument" as a width.

:arg2 Another name for **:height**. Use this with **:functions** which don't interpret their second "argument" as a height.

:visibility Initial visibility, **t**, **nil**, or **blink**. Default is **blink**.

:follow-p **t** if this blinker should follow that pc ppr's cursor. Default is **nil**.

:roving-p **t** if this blinker is not confined to a single piece of paper. In this case the pc ppr argument is ignored and should be **nil**. Defau **nil**.

:activate-p **t** if this blinker should be initially active. The default is **nil**.

:half-period Number of 60ths of a second between changes in the blinker. Default is 15.

:screen The screen on which the blinker should appear. The default is to take it from the pc ppr, or from **tv-default-screen** in the case of a roving blinker.

:sideways-p **t** to make the blinker be rotated 90 degrees. Default is to take it from the pc ppr.

You may give **nil** as a pc-ppr, in which case you must specify **:width** and **:height** (or **:arg1** and **:arg2**) since they will default to **nil**. You should give **nil** as pc-ppr if and only if you specify **:roving-p**, probably, since **:roving-p** means this blinker is not on a pc ppr.

tv-redefine-pc-ppr *pc-ppr* &rest &eval *options*
Redefine some of the parameters of a pc ppr. The allowed options are **:top**, **:bottom**, **:left**, **:right**, **:top-margin**, **:bottom-margin**, **:left-margin**, **:right-margin**, **:vsp**, **:integral-p**, **:more-p**, **:screen**, and **:fonts**. **:fonts** allows you to change the font map, which can change the line height. The size of the blinker will not be changed, but perhaps it should be.

tv-deactivate-pc-ppr *pc-ppr*
Cause a piece of paper's blinkers to stop blinking. It is illegal to type out on a pc ppr which is deactivated.

tv-activate-pc-ppr *pc-ppr*
Cause blinkers to blink again.

tv-deactivate-pc-ppr-but-show-blinkers *pc-ppr*
Cause all blinkers on this piece of paper to be stuck in the blunk (t) state. I.e. mark place but don't flash. Deactivates so that they won't flash. Typing out on this piece of paper will cause blinkers to start blinking again.

tv-return-pc-ppr *pc-ppr*
return-array all of a piece of paper.

tv-make-stream *pc-ppr*
Returns a stream which accepts output and displays it on *pc-ppr*, and reads input from the keyboard, echoing it on *pc-ppr*.

23.12 The Keyboard

Keyboard input can be done either by reading from the **standard-input** stream, which is preferred, or by calling these keyboard routines directly.

The characters read by the functions below are in the Lisp Machine character set, with extra bits to incicate the Control and Meta keys. Also, the characters may come from the *forced-input* mechanism (see page 235), and may be from the mouse. The byte fields which make up the fixnums returned by these functions have names beginning with **"%%kbd-"**, and are explained on page 152.

The special characters Break, Call, and Escape are normally intercepted by the keyboard routines. Break causes the process which reads it to enter a **break** loop (see page 266). Call returns control to the top-level job, or enters a **break** loop if control is already in the top-level job. Control and Meta modifiers cause additional effects. See page 204 for details. Escape is a prefix for various commands, as in ITS. Commands consist of Escape, an optional numeric argument (in octal), and a letter, and do not echo. The commands that currently exist are:

<esc>C Complement TV black-on-white mode.

<esc>*n*C	Complement black-on-white mode of plane *n*.
<esc>*n*S	Select video-switch input *n*.
<esc>M	Complement more-processing enable.
<esc>0M	Turn off more-processing.
<esc>1M	Turn on more-processing.

kbd-tyi &optional *(whostate* "TYI"*)*

This is the main routine for reading from the keyboard. The optional argument is what to display as the program state in the who line (usually just "TYI") while awaiting typein. The value returned is a number which consists of a Lisp machine character code, augmented with bits for the control and meta keys. The character is not echoed.

kbd-tyi-no-hang

Returns **nil** if no character has been typed, or the character code as **kbd-tyi** would return it.

kbd-char-available

Returns non-**nil** if there is a character waiting to be read; otherwise returns **nil**. It does not read the character out. This function can be used with **process-wait**.

kbd-super-image-p *Variable*

If the value of **kbd-super-image-p** is non-**nil**, checking for the special characters Break, Call, and Escape is disabled. Note that you cannot lambda-bind this variable, because it is looked at in different stack-groups.

kbd-simulated-clock-fcn-list *Variable*

List of functions to be called every 60th of a second (while the machine is waiting for typein.) This is used to implement blinkers. [This variable should be renamed and moved to the scheduler section.]

force-kbd-input *job input*

This is used to make a job think it has keyboard input that was not actually typed by the user. The menu system, for example, uses this. *job* is the job to receive the input. *input* is either a fixnum, representing a single character, or an array of characters (which may or may not be a string). **force-kbd-input** waits until previous forced input has been read, then gives the new forced input to the job.

23.13 Internal Special Variables

tv-blinker-list *Variable*
This is a list of all blinkers which are visible (blinking or solidly on). It is used by the **tv-blinker-clock** routine and by **tv-open-screen.**

tv-roving-blinker-list *Variable*
This is a list of peculiar blinkers which don't stay on any single piece of paper. Whenever any piece of paper is opened, in addition to that piece of paper's own blinkers, all of the roving blinkers will be temporarily turned off. Only the visible ones are on this list. This is primarily for the mouse's blinker.

tv-pc-ppr-list *Variable*
This is a list of all the pieces of paper. Currently for no particular reason.

tv-white-on-black-state *Variable*
[??]

tv-beep-duration *Variable*
Controls beeping.

tv-beep-wavelength *Variable*
Controls beeping.

tv-more-processing-global-enable *Variable*
This flag controls whether "**MORE**"s can happen. Complemented by <esc>M. The initial value is **t.**

23.14 Font Utility Routines

[Are these the latest word? I suspect not.]

tv-get-font-pixel *font char row col*
Returns a number which is the pixel value of the specified point in the specified character in the specified font. This is 0 or 1 for normal fonts, or a gray-level value for multi-plane fonts. The value returned is zero if you address outside of the character raster.

tv-store-font-pixel *pixel font char row col*
This is similar to the above, but stores. It is an error to store outside of the character raster.

tv-make-sideways-font *font*

Returns a new font which is the same, except turned on its side in such a way that it works on pieces of paper created with the **sideways-p t** option.

tv-make-dbl-hor-font *font*

Returns a new font with alternating bits split into two planes in such a way that it will work with doubled horizontal resolution (producing squished characters if the original font had a normal aspect ratio.)

tv-make-gray-font *font1* &optional *(x-ratio 2) (y-ratio 2) (n-planes 2)*

Returns a new font which is the original font with areas *x-ratio* wide and *y-ratio* high converted into single points with an appropriate gray level value. *n-planes* determines the number of gray levels available.

23.15 The Font Compiler

The Font Compiler is a lisp program which runs on the pdp10. It converts fonts represented as AST files into QFASL files which can be loaded into the Lisp machine. When a font is loaded, a symbol in the **fonts** package is **setq**'ed to the representation of that font.

To run the font compiler, incant

```
:lispm1;qcmp
(fasload (lmio)fcmp)
(crunit dsk lmfont)  ;Or whatever directory you keep fonts on
(fcmp-1 'input 'output 'fontname screen-type)
```

input is the first-name of the AST file containing the font to be processed. *output* is the first-name of the QFASL file to be produced. *fontname* is the name of the symbol in the **fonts** package whose value will be the font. This symbol will also appear in the **font-name** field of the font and in the printed representation of the font. *screen-type* is **t** if the font is to be used with the 32-bit TV controller, or **nil** if the font is to be used with the 16-bit controller.

[Here insert a catalog of fonts when things settle down a little more.]

24. Errors and Debugging

The first section 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. For a new user of the Lisp Machine, the second section is probably much more useful; you may want to skip the first.

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 *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.

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.

24.1 The Error System

24.1.1 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 Lisp Machine Lisp there is the concept of a *condition*. Every condition has a name, which is a symbol. When an unusual situation occurs, some condition is *signalled*, and a *handler* for that condition is invoked.

When a condition is signalled, the system (essentially) searches up the stack of nested function invocations looking for a handler established to handle that condition. The handler is a function which gets called to deal with the condition. The condition mechanism itself is just a convenient way for finding an appropriate handler function given the name of an exceptional situation. On top of this is built the error-condition system, which defines what arguments are passed to a handler function and what is done with the values returned by a handler function. Almost all current use of the condition mechanism is for errors, but the user may find other uses for the underlying mechanism.

The search for an appropriate handler is done by the function **signal**:

signal *condition-name* &rest *args*

signal searches through all currently-established condition handlers, starting with the most recent. If it finds one that will handle the condition *condition-name*, then it calls that handler with a first argument of *condition-name*, and with *args* as the rest of the arguments. If the first value returned by the handler is **nil**, **signal** will continue searching for another handler; otherwise, it will return the first two values returned by the handler. If **signal** doesn't find any handler that returns a non-**nil** value, it will return **nil**.

Condition handlers are established through the **condition-bind** special form:

condition-bind *Special Form*

The **condition-bind** special form is used for establishing handlers for conditions. It looks like:

```
(condition-bind ((cond-1 hand-1)
                 (cond-2 hand-2)
                 ...)
    body)
```

Each *cond-n* is either the name of a condition, or a list of names of conditions, or **nil**. If it is **nil**, a handler is set up for *all* conditions (this does not mean that the handler really has to handle all conditions, but it will be offered the chance to do so, and can return **nil** for conditions which it is not interested in). Each *hand-n* is a form which is evaluated to produce a handler function. The handlers are established sequentially such that the *cond-1* handler would be looked at first.

Example:

```
(condition-bind ((:wrong-type-argument 'my-wta-handler)
                 ((lossage-1 lossage-2) lossage-handler))
    (princ "Hello there.")
    (= t 69))
```

This first sets up the function **my-wta-handler** to handle the **:wrong-type-argument** condition. Then, it sets up the binding of the symbol **lossage-handler** to handle both the **lossage-1** and **lossage-2** conditions. With these handlers set up, it prints out a message and then runs headlong into a wrong-type-argument error by calling the function = with an argument which is not a number. The condition handler **my-wta-handler** will be given a chance to handle the error. **condition-bind** makes use of ordinary variable binding, so that if the **condition-bind** form is thrown through, the handlers will be disestablished. This also means that condition handlers are established only within the current stack-group.

24.1.2 Error Conditions

The use of the condition mechanism by the error system defines an additional protocol for what arguments are passed to error-condition handlers and what values they may return.

There are basically four possible responses to an error: *proceeding*, *restarting*, *throwing*, or entering the *debugger*. The default action, taken if no handler exists or deigns to handle the error (returns non-**nil**), is to enter the debugger. A handler may give up on the execution that produced the error by throwing (see ***throw**, page 33). *Proceeding* means to repair the error and continue execution. The exact meaning of this depends on the particular error, but it generally takes the form of supplying a replacement for an unacceptable argument to some function, and retrying the invocation of that function. *Restarting* means throwing to a special standard catch-tag, **error-restart**. Handlers cause proceeding and restarting by returning certain special values, described below.

Each error condition is signalled with some parameters, the meanings of which depend on the condition. For example, the condition **:unbound-variable**, which means that something tried to find the value of a symbol which was unbound, is signalled with one parameter, the unbound symbol. It is always all right to signal an error condition with extra parameters.

An error condition handler is applied to several arguments. The first argument is the name of the condition that was signalled (a symbol). This allows the same function to handle several different conditions, which is useful if the handling of those conditions is very similar. (The first argument is also the name of the condition for non-error conditions.) The second argument is a **format** control string (see the description of **format**, on page 85). The third argument is **t** if the error is *proceedable*; otherwise it is **nil**. The fourth argument is **t** if the error is *restartable*; otherwise it is **nil**. The fifth argument is the name of the function that signalled the error, or **nil** if the signaller can't figure out the correct name to pass. The rest of the arguments are the parameters with which the condition was signalled. If the **format** control string is used with these parameters, a readable English message should be produced. Since more information than just the parameters might be needed to print a reasonable message, the program signalling the condition is free to pass any extra parameters it wants to, after the parameters which the condition is *defined* to take. This means that every handler must expect to be called with an arbitrarily high number of arguments, so every handler should have a **&rest** argument (see page 7).

An error condition handler may return any of several values. If it returns **nil**, then it is stating that it does not wish to handle the condition after all; the process of signalling will continue looking for a prior handler (established farther down on the stack) as if the handler which returned **nil** had not existed at all. (This is also true for non-error conditions.) If the handler does wish to handle the condition, it can try to proceed from the error if it is proceedable, or restart from it if it is restartable, or it can throw to a catch tag. Proceeding and restarting are done by returning two values. To proceed, return the symbol **return** as the first value, and the value to be returned by the function **cerror** as the second. To restart, return the symbol **error-restart** as the first value, and the value

to be thrown to the tag **error-restart** as the second. The condition handler must **not** return any other sort of values. However, it can legitimately throw to any tag instead of returning at all. If a handler tries to proceed an unproceedable error or restart an unrestartable one, an error is signalled.

Note that if the handler returns **nil**, it is not said to have handled the error; rather, it has decided not to handle it, but to "continue to signal" it so that someone else may handle it. If an error is signalled and none of the handlers for the condition decide to handle it, the debugger is entered.

Here is an example of an excessively simple handler for the **:wrong-type-argument** condition.

```
;;; This function handles the :wrong-type-argument condition,
;;; which takes two defined parameters: a symbol indicating
;;; the correct type, and the bad value.
(defun sample-wta-handler (condition control-string proceedable-flag
                           restartable-flag function
                           correct-type bad-value &rest rest)
 (prog ()
   (format error-output "~%There was an error in ~S~%" function)
   (lexpr-funcall (function format)
                  control-string correct-type bad-value rest)
   (cond ((and proceedable-flag
               (yes-or-no-p query-io "Do you want use nil instead?"))
          (return 'return nil))
         (t (return nil)))))  ;don't handle
```

24.1.3 Signalling Errors

Some error conditions are signalled by the Lisp system when it detects that something has gone wrong. Lisp programs can also signal errors, by using any of the functions **ferror**, **cerror**, or **error**. **ferror** is the most commonly used of these. **cerror** is used if the signaller of the error wishes to make the error be *proceedable* or *restartable*, or both. **error** is provided for Maclisp compatibility.

A **ferror** or **cerror** that doesn't have any particular condition to signal should use **nil** as the condition name. The only kind of handler that will be invoked by the signaller in this case is the kind that handles *all* conditions, such as is set up by

```
(condition-bind ((nil something) ...) ...)
```

In practice, the **nil** condition is used a great deal.

ferror *condition-name control-string* &rest *params*

ferror signals the condition *condition-name*. Any handler(s) invoked will be passed *condition-name* and *control-string* as their first and second arguments, **nil** and **nil** for the third and fourth arguments (i.e. the error will be neither proceedable nor restartable), the name of the function that called **ferror** for the fifth argument, and *params* as the rest of their arguments.

Note that *condition-name* can be **nil**, in which case no handler will probably be found and the debugger will be entered.

Examples:

```
(cond ((> sz 60)
       (ferror nil
               "The size, ~S, was greater then the maximum"
               sz))
      (t (foo sz)))

(defun func (a b)
  (cond ((and (> a 3) (not (symbolp b)))
         (ferror ':wrong-type-argument
                 "The name, ~1G~S, must be a symbol"
                 'symbolp
                 b))
        (t (func-internal a b))))
```

cerror *proceedable-flag restartable-flag condition-name control-string* &rest *params*

cerror is just like **ferror** (see page 242) except that the handler is passed *proceedable-flag* and *restartable-flag* as its third and fourth arguments. If **cerror** is called with a non-**nil** *proceedable-flag*, the caller should be prepared to accept the returned value of **cerror** and use it to restart the error. Similarly, if he passes **cerror** a non-**nil** *restartable-flag*, he should be sure that there is a ***catch** above him for the tag **error-restart**.

Note: Many programs that want to signal restartable errors will want to use the **error-restart** special form; see page 243.

Example:

```
(do ()
    ((symbolp a))
  ; Do this stuff until a becomes a symbol.
  (setq a (cerror t nil ':wrong-type-argument
                  "The argument ~2G~A was ~1G~S, which is not ~3G~A"
                  'symbolp a 'a "a symbol")))
```

Note: the form in this example is so useful that there is a standard special form to do it, called **check-arg** (see page 244).

error *message* &optional *object interrupt*

error is provided for Maclisp compatibility. In Maclisp, the functionality of **error** is, essentially, that *message* gets printed, preceeded by *object* if present, and that *interrupt*, if present, is a user interrupt channel to be invoked.

In order to fit this definition into the Lisp Machine way of handling errors, **error** is defined to be:

```
(cerror (not (null interrupt))
        nil
        (or (get interrupt 'si:condition-name)
            interrupt)
        (cond ((missing object) ;If no object given
               "~*~a")
              (t "~s ~a"))
        object
        message)
```

Here is what that means in English: first of all, the condition to be signalled is **nil** if *interrupt* is **nil**. If there is some condition whose meaning is close to that of one of the Maclisp user interrupt channels, the name of that channel has an **si:condition-name** property, and the value of that property is the name of the condition to signal. Otherwise, *interrupt* is the name of the condition to signal; probably there will be no handler and the debugger will be entered.

If *interrupt* is specified, the error will be proceedable. The error will not be restartable. The **format** control string and the arguments are chosen so that the right error message gets printed, and the handler is passed everything there is to pass.

error-restart *Macro*

error-restart is useful for denoting a section of a program that can be restarted if certain errors occur during its execution. An **error-restart** form looks like:

```
(error-restart
  form-1
  form-2
  ...)
```

The forms of the body are evaluated sequentially. If an error occurs within the evaluation of the body and is restarted (by a condition handler or the debugger), the evaluation resumes at the beginning of the **error-restart**'s body.

Example:

```
(error-restart
  (setq a (* b d))
  (cond ((> a maxtemp)
         (cerror nil t 'overheat
                 "The frammistat will overheat by ~D. degrees!"
                 (- a maxtemp))))
  (setq q (cons a a)))
```

If the **cerror** happens, and the handler invoked (or the debugger) restarts the error, then evaluation will continue with the **(setq a (* b d))**, and the condition **(> a maxtemp)** will get checked again.

error-restart is implemented as a macro that expands into:

```
(prog ()
 loop (*catch 'error-restart
              (return (progn
                        form-1
                        form-2
                        ...)))
      (go loop))
```

check-arg *Macro*

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

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

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

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

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

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

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

proceed.

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

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

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

If this error got to the debugger, the message

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

would be printed.

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

check-arg uses *predicate* to determine whether the value of the variable is of the correct type. If it is not, **check-arg** signals the **:wrong-type-argument** condition, with four parameters. First, *type-symbol* if it was supplied, or else *predicate* if it was atomic, or else **nil**. Second, the bad value. Third, the name of the argument (*var-name*). Fourth, a string describing the proper type (*description*). If the error is proceeded, the variable is set to the value returned, and **check-arg** starts over, checking the type again. Note that only the first two of these parameters are defined for the **:wrong-type-argument** condition, and so **:wrong-type-argument** handlers should only depend on the meaning of these two.

24.1.4 Standard Condition Names

Some condition names are used by the kernel Lisp system, and are documented below; since they are of global interest, they are on the keyword package. Programs outside the kernel system are free to define their own condition names; it is intended that the description of a function include a description of any conditions that it may signal, so that people writing programs that call that function may handle the condition if they desire. When you decide what package your condition names should be in, you should apply the same criteria you would apply for determining which package a function name should be in; if a program defines its own condition names, they should *not* be on the keyword package. For example, the condition names **chaos:bad-packet-format** and **arpa:bad-packet-format** should be distinct. For further discussion, see page 176.

The following table lists all standard conditions and the parameters they take; more will be added in the future. These are all error-conditions, so in addition to the condition name and the parameters, the handler receives the other arguments described above.

:wrong-type-argument *type-name value*
value is the offending argument, and *type-name* is a symbol for what type is required. Often, *type-name* is a predicate which returns non-**nil** if applied to an acceptable value. If the error is proceeded, the value returned by the handler should be a new value for the argument to be used instead of the one which was of the wrong type.

:inconsistent-arguments *list-of-inconsistent-argument-values*
These arguments were inconsistent with each other, but the fault does not belong to any particular one of them. This is a catch-all, and it would be good to identify subcases in which a more specific categorization can be made. If the error is proceeded, the value returned by the handler will be returned by the function whose arguments were inconsistent.

:wrong-number-of-arguments *function number-of-args-supplied list-of-args-supplied*
function was invoked with the wrong number of arguments. The elements of *list-of-args-supplied* have already been evaluated. If the error is proceeded, the value returned should be a value to be returned by *function*.

:invalid-function *function-name*
The name had a function definition but it was no good for calling. You can proceed, supplying a value to return as the value of the call to the function.

:invalid-form *form*
The so-called *form* was not a meaningful form for **eval**. Probably it was of a bad data type. If the error is proceeded, the value returned should be a new form; **eval** will use it instead.

:undefined-function *function-name*
The symbol *function-name* was not defined as a function. If the error is proceeded, then the symbol will be defined to the function returned, and that function will be used to continue execution.

:unbound-variable *variable-name*
The symbol *variable-name* had no value. If the error is proceeded, then the symbol will be set to the value returned by the handler, and that value will be used to continue execution.

Currently, errors detected by microcode do not signal conditions. Generally this means that errors in interpreted code signal conditions and some errors in compiled code do not. This will be corrected some time in the future.

24.1.5 Errset

As in Maclisp, there is an *errset* facility which allows a very simple form of error handling. If an error occurs inside an errset, and no condition handler handles it, i.e. the debugger would be entered, control is returned (*thrown*) to the errset. The errset can control whether or not the debugger's error message is printed.

A problem with **errset** is that it is *too* powerful; it will apply to any unhandled error at all. If you are writing code that anticipates some specific error, you should find out what condition that error signals and set up a handler. If you use **errset** and some unanticipated error crops up, you may not be told—this can cause very strange bugs.

errset *Special Form*

The special form (**errset** *form flag*) 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 *Variable*

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

err *Special Form*

This is for Maclisp compatibility.

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

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

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

24.2 The Debugger

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

The user interface described herein is not thought too well of, and we hope to redesign it sometime soon.

24.2.1 Entering the Debugger

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

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

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

Here is an example of an error from Lisp code:

```
>>ERROR: The argument X was 1, which is not a symbol,
While in the function { FERROR ← } FOO ← *EVAL
```

Here the first line contains the English description of the error message, and the second line contains the abbreviated backtrace. The backtrace indicates that the function which actually entered the error handler was **ferror**, but that function is enclosed in braces because it is not very important; the useful information here is that the function **foo** is what called **ferror** and thus signalled the error.

There is not any good way to manually get into the debugger; the interface will someday be fixed so that you can enter it at any time if you want to use its facilities to examine the state of the Lisp environment and so on. In the meantime, just type an unbound symbol at Lisp top level.

24.2.2 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, and then explains them in approximate order of usefulness. A summary is provided at the end of the listing.

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

```
→
```

At this point, you may either type in a Lisp expression, or type a command (a Control or Meta character is interpreted as a command, whereas a normal character is interpreted as the first character of an expression). If you type a Lisp expression, it will be interpreted as a Lisp form, and will be evaluated in the context of the function which got the error. (That is, all bindings which were in effect at the time of the error will be in effect when your form is evaluated.) The result of the evaluation will be printed, and the debugger will

prompt again with an arrow. If, during the typing of the form, you change your mind and want to get back to the debugger's command level, type a Control-Z; the debugger will respond with an arrow prompt. In fact, at any time that typein is expected from you, you may type a Control-Z to flush what you are doing and get back to command level. This **read-eval-print** loop maintains the values of +, *, and - just as the top-level one does.

Various debugger commands ask for Lisp objects, such as an object to return, or the name of a catch-tag. Whenever it tries to get a Lisp object from you, it expects you to type in a *form*; it will evaluate what you type in. This provides greater generality, since there are objects to which you might want to refer that cannot be typed in (such as arrays). If the form you type is non-trivial (not just a constant form), the debugger will show you the result of the evaluation, and ask you if it is what you intended. It expects a Y or N answer (see the function **y-or-n-p**, page 263), and if you answer negatively it will ask you for another form. To quit out of the command, just type Control-Z.

24.2.3 Debugger Commands

All debugger commands are single characters, usually with the Control or Meta bits. The single most useful command is Control-Z, which exits from the debugger and throws back to the Lisp top level loop. ITS users should note that Control-Z is not Call. Often you are not interested in using the debugger at all and just want to get back to Lisp top level; so you can do this in one character. This is similar to Control-G in Maclisp.

Self-documentation is provided by the Help (top-H) or "?" command, which types out some documentation on the debugger commands.

Often you want to try to continue from the error. To do this, use the Control-C command. The exact way Control-C works depends on the kind of error that happened. For some errors, there is no standard way to continue at all, and Control-C will just tell you this and return to the debugger's command level. For the very common "unbound symbol" error, it will get a Lisp object from you, which it will store back into the symbol. Then it will continue as if the symbol had been bound to that object in the first place. For unbound-variable or undefined-function errors, you can also just type Lisp forms to set the variable or define the function, and then type Control-C; it will proceed without asking anything.

Several commands are provided to allow you to examine the Lisp control stack (regular pdl), which keeps a record of all functions which are 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, and also the saved address at which the function was executing (in case you want to look at

the code generated by the compiler); for the example above it might look like:

```
FOO: (P.C. = 23)
   Arg 0 (X): 13
   Arg 1 (Y): 1

BAR: (P.C. = 120)
   Arg 0 (ADDEND): 13
```

and so on. This means that **foo** was executing at instruction 23, and 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.

The debugger knows about a "current stack frame", and there are several commands which use it. The initially "current" stack frame is the one which signalled the error; either the one which got the microcode error, or the one which called **ferror** or **error**.

The command Control-L (or Form) clears the screen, retypes the error message that was initially printed when the debugger was entered, and then prints out a description of the current frame, in the format used by Meta-B. The Control-N command moves "down" to the "next" frame (that is, it changes the current frame to be the frame which called it), and prints out the frame in this same format. Control-P moves "up" to the "previous" frame (the one which this one called), and prints out the frame in the same format. Meta-< moves to the top of the stack, and Meta-> to the bottom; both print out the new current frame. Control-S asks you for a string, and searches the stack for a frame whose executing function's name contains that string. That frame becomes current and is printed out. These commands are easy to remember since they are analogous to editor commands.

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. Meta-N moves to the next frame and prints it out in full-screen format, and Meta-P moves to the previous frame and prints it out in full-screen format. Meta-S is like Control-S but does a full-screen display.

Control-A prints out the argument list for the function of the current frame, as would be returned by the function **arglist** (see page 61). Control-R is used to return a value from the current frame; the frame that called that frame continues running as if the function of the current frame had returned. This command prompts you for a form, which it will evaluate; it returns the resulting value, possibly after confirming it with you. Meta-R is used to return multiple values from the current frame, but it is not currently implemented. The Control-T command does a *throw* to a given tag with a given value; you are prompted for the tag and the value.

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

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 **rplaca** or **rplacd** on the locative). Control-Meta-L is similar, but refers to the *n*th local variable of the frame.

24.2.4 Summary of Commands

Control-A	Print argument list of function in current frame.
Control-Meta-A	Examine or change the *n*th argument of the current frame.
Control-B	Print brief backtrace.
Meta-B	Print longer backtrace.
Control-C	Attempt to continue.
Meta-C	Attempt to restart.
Control-G	Quit to command level.
Control-L	Redisplay error message and current frame.
Meta-L	Full-screen typeout of current frame.
Control-N	Move to next frame. With argument, move down *n* frames.
Meta-N	Move to next frame with full-screen typeout. With argument, move down *n* frames.
Control-P	Move to previous frame. With argument, move up *n* frames.
Meta-P	Move to previous frame with full-screen typeout. With argument, move up *n* frames.
Control-R	Return a value from the current frame.
Meta-R	Return several values from the current frame. (doesn't work)
Control-S	Search for a frame containing a specified function.
Meta-S	Same as control-S but does a full display.
Control-T	Throw a value to a tag.
Control-Z	Throw back to Lisp top level.
? or Help	Print a help message.
Meta-<	Go to top of stack.
Meta->	Go to bottom of stack.
Form	Same as Control-L.

Line Move to next frame. With argument, move down *n* frames. Same as Control-N.

Return Move to previous frame. With argument, move up *n* frames. Same as control-P.

24.2.5 Miscellany

Sometimes, e.g. when the debugger is running, microcode trapping is "disabled": any attempt by the microcode to trap will cause the machine to halt.

trapping-enabled-p
This predicate returns **t** if trapping is enabled; otherwise it returns **nil**.

enable-trapping &optional *(arg* **1***)*
If *arg* is **1**, trapping is enabled. If it is **0**, trapping is disabled.

24.3 Trace

The *trace* facility allows the user to trace some functions. When a function is traced, certain special actions will be taken when it is called, and when it returns. The function **trace** allows the user to specify this.

The trace facility is closely compatible with Maclisp. Although the functions of the trace system which are presented here are really functions, they are implemented as special forms because that is the way Maclisp did it.

trace *Special Form*
A **trace** form looks like:

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

A *spec* may be either a symbol, which is interpreted as a function name, or a list of the form (*function-name option-1 option-2* ...). If *spec* is a symbol, it is the same as giving the function name with no options. Some options take "arguments", which should be given immediately following the option name.

The following 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**.

: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.

:step Causes the function to be single-stepped whenever it is called. See the documentation on the step facility below.

: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.

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

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

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

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

would cause a break in **foo** if and only if the first argument to **foo** is **nil**. **arglist** should have a colon, but it is omitted because this is the name of a system function and therefore global.

Similarly, the variable **si:fnvalues** will be a list of the resulting values of the traced function. For obvious reasons, this should only be used with the **exit** option.

The trace specifications may be "factored." For example,

```
(trace ((foo bar) wherein baz value))
```

is equivalent to

```
(trace (foo wherein baz value) (bar wherein baz value))
```

This is not yet supported.

All output printed by trace can be ground into an indented, readable format, by simply setting the variable **sprinter** to **t**. Setting **sprinter** to **nil** changes the output back to use the ordinary print function, which is faster and uses less storage but is less readable for large list structures. This is not yet supported.

trace returns as its value a list of names of all functions traced; for any functions traced with the **wherein** option, say **(trace (foo wherein bar))**, instead of putting just **foo** in the list it puts in a 3-list **(foo wherein bar)**.

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

It is possible to call **trace** with no arguments. **(trace)** evaluates to a list of all the functions currently being traced.

untrace *Special Form*

> **untrace** is used to undo the effects of **trace** and restore functions to their normal, untraced state. The argument to **untrace** for a given function should be what **trace** returned for it; i.e. if **trace** returned **foo**, use **(untrace foo)**; if **trace** returned **(foo wherein bar)** use **(untrace (foo wherein bar))**. **untrace** will take multiple specifications, e.g. **(untrace foo quux (bar wherein baz) fuphoo)**. Calling **untrace** with no arguments will untrace all functions currently being traced.

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

trace-compile-flag *Variable*

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

24.4 The Stepper

The Step facility provides 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.

24.4.1 How to Get Into the Stepper.

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

step *form*

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

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

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

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

The other way to get into the stepper is to use the **step** option of **trace** (see page 252). 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.

24.4.2 How to Use 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.

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

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

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

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

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

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

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

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

Control-E (Editor)
: Editor escape (enter the Eine editor).

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

  step-form which is the current form.

  step-values which is the list of returned values.

  step-value which is the first returned value.

  If you change the values of these variables, it will work.

Control-L
: Clear the screen and redisplay the last 10. pending forms (forms which 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.

24.5 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, and so the speed of general execution when the MAR is enabled is not significantly slowed down. However, the speed of accessing pages of memory containing the locations being checked is slowed down, since every reference involves a microcode trap.

These are the functions that control the MAR:

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

The **set-mar** function clears any previous setting of the MAR, and sets the MAR on *n-words* words, starting at *location*. *location* may be any object. *n-words* currently defaults to 1, but eventually it will 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 on the value of a variable, use

```
(set-mar (value-cell-location symbol) :write)
```

clear-mar

This turns off the MAR. Restarting 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.

si:%mar-low *Variable*
si:%mar-high *Variable*

These two fixnums are the inclusive boundaries of the area of memory monitored by the MAR. The values of these variables live inside the microcode.

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 considerably 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. Currently it is not possible to proceed from a MAR trrap, because some machine

state is lost. Eventually, most MAR traps will be continuable.

25. Utility Programs

ed &optional *x*

ed is the main function for getting into the editor, Eine. Eine is not yet documented in this manual, however the commands are very similar to Emacs.

(**ed**) or (**ed nil**) simply enters Eine, leaving you in the same buffer as the last time Eine was running.

(**ed t**) puts you in a fresh buffer with a generated name (like BUFFER-4).

(**ed 'foo**) tries hard to edit the definition of the **foo** function. If there was a buffer named FOO already, it selects it. If **foo** is defined as an interpreted function (or if it was compiled on the Lisp machine and the compiler saved the interpreted function) then that function is **grindef**'ed into a new buffer called FOO. If **foo** is not defined but has a value, it will edit that value in a buffer called FOO-VALUE. Otherwise it will create a buffer called FOO and put in "(**defun foo** (" so that you can type in the definition.

If you call **ed** on a list, it will **grindef** that list into a new buffer.

If you call **ed** on a buffer, or a string or symbol which is the name of a buffer, it will edit that buffer.

edval *sym*

Enters Eine, selecting a buffer called *sym*-VALUE. If that buffer did not previously exist, a **setq** of *sym* to its current value is **grindef**'ed into the buffer.

edprop *sym prop*

Enters Eine, selecting a buffer called *sym-prop*-PROPERTY. If that buffer did not previously exist, a **putprop** of *sym* to its current *prop*-property is **grindef**'ed into the buffer.

peek &optional *character*

peek is similar to the ITS program of the same name. It displays various information about the system, periodically updating it. Like ITS PEEK, it has several modes, which are entered by typing a single key which is the name of the mode. The initial mode is selected by the argument, *character*. If no argument is given, **peek** starts out in "N" mode.

The currently implemented modes are:

N (for Normal)

Display all active processes, showing their names and whostates (see page 195).

M (for Memory)

Display the amount of room left in all areas (this is the same as

(room t) (see page 262).

K (for Chaosnet)
: Display various information about all open Chaosnet connections (see <not-yet-written>).

?
: Give self-documentation.

B (for Back)
: Go back to the previous mode.

Q (for Quit)
: Exit from **peek**.

Space
: Update the display immediately.

At the top of the screen, **peek** displays the version number of the microcode, the time (as returned by **(time)**), and the amount of room left in the **working-storage-area** and **macro-compiled-program** areas.

supdup &optional *host window-size*

host may be a string or symbol, which will be taken as a host name, or a number, which will be taken as a host number. If no *host* is given, MIT-MC 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 system.

window-size should be a fixnum; it defaults to **3**. Its value will be used as the window size of the Chaos net connection.

To give commands to **supdup**, type a Break followed by one character. The commands are as closely compatible with ITS as possible. The characters currently implemented are:

Call
: Enter a breakpoint.

C (for Change)
: Change the escape character (normally Break) to something else.

Q (for Quit)
: Close the connection and return.

L (for Logout)
: Tell the foreign host to try to log out your process, then close the connection and return.

Help or ?
: Document these commands.

Rubout
: Do nothing (useful if you accidentally type Break).

dribble-start *filename*

dribble-start opens *filename* as a "dribble file" (also known as a "wallpaper file"). It rebinds **standard-input** and **standard-output** so that all of the terminal interaction is directed to the file as well as the terminal.

Currently, there can only be one output file open at a time; thus, while you are dribbling, you can't write files.

dribble-end

This closes the file opened by **dribble-start** and resets the I/O streams.

25.1 Useful Commands

who-calls *x* &optional *package*
who-uses *x* &optional *package*

x must be a symbol. **who-calls** tries to find all of the compiled functions in the Lisp world which call *x* as a function, use *x* as a variable, or use *x* as a constant. (It won't find things that use constants which contain *x*, such as a list one of whose elements is *x*; it will only find it if *x* itself is used as a constant.) It tries to find all of the compiled code objects by searching all of the function cells of all of the symbols on *package* and *package*'s decendants. *package* defaults to the **global** package, and so normally all packages are checked.

If **who-calls** encounters an interpreted function definition, it simply tells you if *x* appears anywhere in the interpreted code.

who-uses is currently the same thing as **who-calls**.

The symbol **unbound-function** is treated specially by **who-calls**. (**who-calls 'unbound-function**) will search the compiled code objects for any calls through a symbol which is not currently defined as a function. This is useful for finding errors.

apropos *string* &optional *package*

(**apropos** *string*) tries to find all symbols whose print-names contain *string* 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, it tells you so, and prints the function (if any). It finds the symbols on *package* and *package*'s decendants. *package* defaults to the **global** package, so normally all packages are searched.

where-is *pname* &optional *package*

Prints the names of all packages which contain a symbol with the print-name *pname*. *pname* gets upper-cased. The package *package* and all its sub-packages are searched; *package* defaults to the **global** package, which causes all packages to be searched.

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, flonums, packages, stack groups, closures, and FEFs, and prints out the attributes of each in human-readable form. Sometimes it will describe something which it finds inside something else; such recursive descriptions are indented appropriately. For instance, **describe** of a symbol will tell you about the symbol's value, its definition, and each of its properties. **describe** of a flonum (regular or small) will show you its internal representation in a way which is useful for tracking down roundoff errors and the like.

If *x* is a named-structure, **describe** handles it specially. To understand this, you should read the section on named structures (see page 91). First it gets the named-structure symbol, and sees whether its function knows about the **:describe** operation. If the operation is known, it applies the function to two arguments: the symbol **:describe**, and the named-structure itself. Otherwise, it looks on the named-structure symbol for information which 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-file *filename*

This prints what the system knows about the file *filename*. It tells you what package it is in and what version of it is loaded.

describe-package *package-name*

(**describe-package** *package-name*) is equivalent to (**describe** (**pkg-find-package** *package-name*)); that is, it describes the package whose name is *package-name*.

describe-area *area*

area may be the name or the number of an area. Various attributes of the area are printed.

room &optional *arg*

room tells 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 words used in the area, the size of the area, and the percentage of words which are used in the area.

If *arg* is not given, the value of **room** should be a list of area numbers and/or area names; **room** describes those areas. If *arg* is a fixnum, **room** describes that area. If *arg* is **t**, **room** describes all areas.

room *Variable*

The value of **room** is a list of area names and/or area numbers, denoting the areas which the function **room** will describe if given no arguments.* Its initial value is:

```
(working-storage-area macro-compiled-program)
```

set-memory-size *n-words*

set-memory-size tells the virtual memory system to only use *n-words* words of main memory for paging. Of course, *n-words* may not exceed the amount of main memory on the machine.

recompile-world &rest *keywords*

recompile-world is a rather ad-hoc tool for recompiling all of the Lisp Machine system packages. It works by calling the **pkg-load** facility (see page 194). It will find all files that need recompiling from any of the packages:

```
system-internals format compiler
chaos supdup peek eine
```

keywords is a list of keywords; usually it is empty. The useful keywords are:

load After compiling, load in any files which are not loaded.

noconfirm Don't ask for confirmation for each package.

selective Ask for confirmation for each file.

Any of the other keywords accepted by **pkg-load** will also work.

qld &optional *restart-p*

qld is used to generate a new Lisp Machine system after the cold-load is loaded in. If you don't know how to use this, you don't need it. If **restart-p** is non-**nil**, then it ignores that it has done anything, and starts from scratch.

disassemble *function*

function should be a FEF, or a symbol which is defined as a FEF. This prints out a human-readable version of the macro-instructions in *function*. The macro-code instruction set is explained on <not-yet-written>.

print-disk-label

Tells you what is on the disk.

set-current-band *band*

Sets which "band" (saved virtual memory image) is to be loaded when the machine is started. Use with caution!

set-current-microload *band*

Sets which microload (MCR1 or MCR2) is to be loaded when the machine is started. Use with caution!

25.2 Querying the User

y-or-n-p &optional *stream message*

This is used for asking the user a question whose answer is either "Yes" or "No". It types out *message* (if any) and reads in one character from the keyboard. If the character is Y, T, or space, it returns **t**. If the character is N or rubout, it returns **nil**. Otherwise it prints out *message* (if any), followed by "(Y or N)", to *stream* and tries again. *stream* defaults to **standard-output**.

y-or-n-p should only be used for questions which 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 *stream message*

This is used for asking the user a question whose answer is either "Yes" or "No". It types out *message* (if any) and reads in a line from the keyboard. If the line is the string "Yes", it returns **t**. If the line is "No", it returns **nil**. (Case is ignored, as are leading and trailing spaces and tabs.) Otherwise it prints out *message* (if any), followed by 'Please type either "Yes" or "No"' to *stream* and tries again. *stream* defaults to **standard-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.

setup-keyboard-dispatch-table *array lists*

Several programs on the Lisp Machine accept characters from the keyboard and take a different action on each key. This function helps the programmer initialize a dispatch table for the keyboard.

array should be a two-dimensional array with dimensions **(4 220)**. The first subscript to the array represents the Control and Meta keys; the Control key is the low order bit and the Meta key is the high order bit. The second subscript is the character typed, with the Control and Meta bits stripped off. In other words, the first subscript is the **%%kbd-control-meta** part of the character and the second is the **%%kbd-char** part.

lists should be a list of four lists. Each of these lists describes a row of the table. The elements of one of these lists are called *items*, and may be any of a number of things. If the item is not a list, it is simply stored into the next location of the row. If the item is a list, then its first element is inspected. If the first element is the symbol **:repeat**, then the second element should be a fixnum *n*, and the third element is stored into the next *n* locations of the row. If the first element is **:repeat-eval**, it is treated as is **:repeat** except that the third element of the item is a form which is evaluated before being put into the array. The form may take advantage of the symbol **si:rpcnt**, which is set to **0** the first time the form is evaluated, and is increased by one every subsequent time. If the first element of an item is **:eval**, then the second element is evaluated, and the result is stored into the next location of the row. Otherwise, the item itself is stored in the next location of the row. Altogether exactly all 220 locations of the row must be filled in, or else an error will be signalled.

25.3 Stuff That Doesn't Fit Anywhere Else

time

(**time**) returns a number which increases by 1 every 1/60 of a second, and wraps around at some point (currently after 18 bits' worth). The most important thing about **time** is that it is completely incompatible with Maclisp; this will get changed.

defun-compatibility *x*

This function is used by **defun** and the compiler to convert Maclisp-style **defun**s to Lisp Machine definitions. *x* should be the cdr of a (**defun** ...) form. **defun-compatibility** will return a corresponding (**defun** ...) or (**macro** ...) form, in the usual Lisp Machine format.

set-error-mode &optional *(car-sym-mode* **1***) (cdr-sym-mode* **1***)*
(car-num-mode **0***) (cdr-num-mode* **0***)*

set-error-mode sets the four "error mode" variables. See the documentation of **car** and **cdr** (page 38) which explains what these mean.

print-error-mode &optional *mode stream*

This prints an English description of the error-mode number *mode* onto the output stream *stream*. *mode* defaults to the mode currently in effect, and *stream* defaults to **standard-output**.

***rset** *flag*

Sets the variable ***rset** to *flag*. Nothing looks at this variable; it is a vestigial crock left over from Maclisp.

disk-restore &optional *partition*

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 default load, which is the one the machine loads automatically when it is booted. The specified partition is copied into the paging area of the disk and then started. Lisp-machine disks currently contain seven partitions on which copies of virtual-memory may be saved for later execution in this way.

disk-restore asks the user for confirmation before doing it.

disk-save *partition*

partition may be the name or the number of a disk partition containing a virtual-memory load, or **nil**, meaning to use the default load, which is the one the machine loads automatically when it is booted. The current contents of virtual memory are copied from main memory and the paging area of the disk into the specified partition, and then restarted as if it had just been reloaded.

disk-save asks the user for confirmation before doing it.

25.4 Status and SStatus

The **status** and **sstatus** special forms exist for compatibility with Maclisp. Programs that wish to run in both Maclisp and Lisp Machine Lisp can use **status** to determine which of these they are running in. Also, **(sstatus feature ...)** can be used as it is in Maclisp.

status *Special Form*

(status features) returns a list of symbols indicating features of the Lisp environment. The complete list of all symbols which may appear on this list, and their meanings, is given in the Maclisp manual. The default list for the Lisp Machine is:

```
(sort fasload strings newio roman trace grindef grind lispm)
```

The value of this list will be kept up to date as features are added or removed from the Lisp Machine system. Most important is the symbol **lispm**, which is the last element of the list; this indicates that the program is executing on the Lisp Machine.

(status feature *symbol*) returns **t** if *symbol* is on the **(status features)** list, otherwise **nil**.

(status nofeature *symbol*) returns **t** if *symbol* is not on the **(status features)** list, otherwise **nil**.

(status status) returns a list of all **status** operations.

(status sstatus) returns a list of all **sstatus** operations.

sstatus *Special Form*

(sstatus feature *symbol*) adds *symbol* to the list of features.

(sstatus nofeature *symbol*) removes *symbol* from the list of features.

25.5 The Lisp Top Level

These functions constitute the Lisp top level, 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
This is the actual top level loop. It prints out "**105 FOOBAR**" and then goes into a loop reading a form from **standard-input**, evaluating it, and printing the result (with slashification) to **standard-output**. If several values are returned by the form all of them will be printed. Also the values of *, +, and - are maintained (see below).

break *Special Form*
break is used to enter a breakpoint loop, which is similar to a Lisp top level loop. (**break** *tag*) will always enter the loop; (**break** *tag conditional-form*) will evaluate *conditional-form* and only enter the break loop if it returns non-**nil**. If the break loop is entered, **break** prints out

```
;bkpt tag
```

and then enters a loop reading, evaluating, and printing forms. A difference between a break loop and the top level loop is that after reading a form, **break** checks for the following special cases: if the symbol **◊g** is typed, **break** throws back to the Lisp top level. If **◊p** is typed, **break** returns **nil**. If (**return** *form*) is typed, **break** evaluates *form* and returns the result.

\- *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*
While a form is being evaluated by a read-eval-print loop, * is bound 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.

lisp-initialization-list *Variable*
The value of **lisp-initialization-list** is a list of forms, which are sequentially evaluated by **lisp-reinitialize**.

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**.

25.6 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. The init file is named *user*; .LISPM (INIT) if you have a directory.

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 which is the value of **logout-list**. The functions **login-setq** and **login-eval** help 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 which are evaluated when a user logs out.

login *name*

If anyone is logged into the machine, **login** logs him out. (See **logout**.) Then **user-id** is set from *name*. Finally **login** attempts to find your INIT file. It first looks in "*user-id*; .LISPM (INIT)", then in "(INIT); *user-id* .LISPM", and finally in the default init file "(INIT); * .LISPM". When it finds one of these that exists, it loads it in. **login** returns **t**.

logout

First, **logout** evaluates the forms on **logout-list**. Then it tries to find a file to run, looking first in "*user-id*; .LSPM_ (INIT)", then in "(INIT); *user-id* .LSPM_", and finally in the default file "(INIT); * .LSPM_". If and when it finds one it these that exists, it loads it in. Then it sets **user-id** to an empty string and **logout-list** to **nil**, and returns **t**.

login-setq *Special Form*

login-setq is like **setq** except that it puts a **setq** form on **logout-list** to set the variables to their previous values.

login-eval *x*

login-eval is used for functions which are "meant to be called" from INIT files, such as **eine:ed-redefine-keys**, which conveniently return a form to undo what they did. **login-eval** adds the result of the form *x* to the **logout-list**.

Concept Index

Variable Index

Function Index

Table of Contents

1. Basic Window Features

This chapter describes the features provided by the window flavor, which therefore apply to most windows. Later chapters describe hairier features which are incorporated into some windows. Most of the features are explained with a brief description followed by an enumeration of the messages that you can send to invoke them, the instance-variables associated with the feature, and the related window creation options (if any).

Most of the messages described in this chapter are essential to the workings of the system, and should not have their primary methods redefined by the user. When this is not the case, the text will say so explicitly. In any case, it is all right to put daemons on any message.

1.1 Flavor Naming Conventions

The following conventions are followed for naming flavors of windows. In this section the word *frobboz* is used to stand for any feature, attribute, or class of windows that would appear in a flavor name (e.g. peek, lisp-listener, or delayed-redisplay-label). Naming conventions are different for *instantiatable* flavors (which are complete and can support instances of themselves) and *mixin* flavors (which are incomplete and only supply one particular aspect of behavior).

frobboz — An instantiatable flavor whose most distinguishing characteristic is that it is a "frobboz". frobboz is preferred to frobboz-window except when it is necessary to make a distinction.

frobboz-mixin — A flavor which provides the "frobboz" feature when mixed into other flavors. This generally has no explicit components, only included-flavors. Not instantiatable by itself.

basic-frobboz — The same as frobboz-mixin except that it alters the "essential character" of the window. It does not work to mix two "basic" flavors together unless they know about each other. In certain cases a basic-frobboz may be instantiatable without other flavors, while in other cases it is more like a mixin and not instantiatable.

essential-frobboz
essential-frobboz-mixin — Something which is needed in order to work. These are often but not always components of minimum-window. They are also usually internal things the user does not see.

minimum-frobboz — A simpler type of "frobboz" than frobboz or basic-frobboz, for the case where those latter flavors want to be built out of components.

window — The simplest type of window. Almost all user-defined windows should be built by adding mixins to the window flavor, although occasionally the need will

:edges-from *source*

Specifies that the window is to take its edges (position and size) from *source*, which can be one of:

a string — The inside-size of the window is made large enough to display the string, in font 0.

a list (*left top right bottom*) — Those edges, relative to the superior, are used.

:mouse — The user is asked to point the mouse to where the top-left and bottom-right corners of the window should go.

a window — That window's edges are copied.

:minimum-width *n-pixels*
:minimum-height *n-pixels*

In combination with the :edges-from ':mouse init option, these options specify the minimum size of the rectangle accepted from the user.

:left *x-position*
:x *x-position*
:top *y-position*
:y *y-position*
:right *right-edge-x-position*
:bottom *bottom-edge-y-position*
:edges *(left top right bottom)*
:width *outside-width-in-pixels*
:height *outside-height-in-pixels*

These set the position and size of the window relative to its superior. The default if the position is unspecified is *(0,0)*. The default if the size is unspecified is the inside-size of the superior. The right thing happens if you specify only some parts of this information.

:character-width *spec*

This is another way of specifying the width. *spec* is either a number of characters or a character string. The inside-width of the window is made to be wide enough to display those characters in font 0.

:character-height *spec*

This is another way of specifying the height. *spec* is either a count of lines or a character string containing a certain number of lines separated by carriage returns. The inside-height of the window is made to be that many lines.

bits-action controls what happens to the bits which compose this window's display. The allowed values are:

:restore — The bits are restored to what they used to be. That is, the window's bits are moved from where they were automatically saved, into the screen's physical bits.

:clean — The window is refreshed, that is, made blank for the most part.

:noop — Nothing is done with the bits. The window's bits become set to whatever was on that part of the screen previously.

bits-action defaults to :restore if that is possible (the window has a bit-save-array), otherwise to :clean.

:deexpose &optional *save-bits-p screen-bits-action remove-from-superior*

This is the opposite of :expose. This message is sent by the system when a window needs to be removed from the screen. It is usually a mistake to send it yourself; unless screen-management is delayed, the screen manager will immediately re-expose the window, since it is not covered by anything. See the :bury message.

If the window is currently selected, it is first deselected, since the selected window must always be exposed.

The allowed values for *save-bits-p* are:

:default — The bits are saved if a bit-save array exists, otherwise they are not. This is, of course, the default.

:force — If a bit-save array does not exist, one is created, and the bits are saved.

nil — The bits are not saved.

screen-bits-action controls what is done to the bits left behind on the screen. If this is a temporary window, the supplied value is ignored and the bits belonging to the windows underneath are restored from where they were temporarily saved. Otherwise the allowed values are:

:noop — Leave them there (however, the screen manager will come along and replace them with the bits of whatever window shows through). This is the default.

:clean — Clear them out.

remove-from-superior is for internal use, and should not be supplied.

:activate

Causes the window to be on its superior's list of inferiors. Active windows are visible to the rest of the system for automatic exposure and screen management. When a window is exposed, it will be automatically activated if it is not already.

:deactivate

Causes the window to be deexposed and removed from its superior's list of inferiors. The window system will not remember this window anywhere, so it can be garbage collected.

:kill

Causes the window to become deactivated, with the intent that it will never be used again. Distinct from :deactivate so that daemons can be placed on it to clean up things like processes.

:select &optional (*save-selected* t)

[The *save-selected* argument is obsolete and doesn't do anything now.]

Causes the window to become the *selected window*, which is the one that is allowed to :tyi from the keyboard. It and all its superiors are automatically activated and exposed if they were not already. The currently selected window is sent a :deselect message with argument nil, causing it to be remembered as the previously-selected window. The window's blinkers are made to blink if that is appropriate (see <not-yet-written>).

tv:selected-window *Variable*

This is nil or the currently-selected window.

:deselect &optional (*restore-selected* t)

Causes the window to no longer be the *selected window*. If *restore-selected* is t, then the window that was selected before this one is selected again, and this window is put on the end of the ring buffer of previously-selected windows. Otherwise this window is put on the front of the ring buffer and no window is selected. This window's blinkers are set to their deselected-blinker-visibility; typically they stop blinking.

You don't normally want to send this message yourself with an argument of nil, since the screen manager, if not delayed, will automatically select a window, which may be this one.

:mouse-select &optional (*save-selected* t)

This is used when you select a window by pointing to it with the mouse or by using the select operation in the system menu. It represents a "stronger" form of selection. The arguments are the same as for :select. Deexposes any temporary windows that lock the window, copies any keyboard typeahead into the currently selected window's io-buffer, and then sends a :select message to the window.

:save-bits
:set-save-bits *t-or-nil*
Get or set whether the window saves its bits when it is deexposed.

:set-superior *new-superior*
Causes the superior of the window to be *new-superior*. Use this with caution! It doesn't completely work for all cases.

:lisp-listener-p
Returns nil if the window is not a *lisp listener*, :idle if a *lisp listener* and not currently eval'ing a form, or :busy if a *lisp listener* but currently eval'ing a form.

The following instance variables are relevant to these issues and may be of interest to the user. There are of course quite a few related instance variables which are internal and not documented here [but perhaps they should be?]

tv:screen-array *Variable*
The two-dimensional array of bits on which a window's output is drawn. nil if the window is deexposed and has no bit array.

tv:bit-array *Variable*
The array in which the window saves its bits when not exposed, or nil if it does not do so. Several aspects of a window's behavior depend on whether or not this is null.

tv:superior *Variable*
The window within which this one appears. nil if this is top-level (typically a screen).

tv:restored-bits-p *Variable*
This is used for communication to the :after :refresh methods; if it is t the bits of the window have been restored from the saved copy, but if it is nil they need to be regenerated.

tv:name *Variable*
The name of the window. This string is the default thing displayed in the label and appears in the printed-representation of the window.

tv:process *Variable*
For a window that incorporates the process-mixin flavor, this is the process associated with the window, or nil.

[Should the locking and temporary stuff be documented here, or assumed to be internal?]

Here are some relevant window-creation options.

:inside-edges
Like :edges, but returns the edges excluding the margins.

:set-edges *new-left new-top new-right new-bottom* &optional *option*
Changes the size and position of a window as specified by the first four arguments. *option* may be :verify, in which case t is returned if the edges are acceptable, or nil if they are not, and nothing is actually changed. Sends the :verify-new-edges message in order to check the new edges. If not merely verifying, and not changing the size of the window, then the window is moved without any furthur message transmission. On the other hand, if the size is changing, then a :change-of-size-or-margins message is sent to the window.

:full-screen &optional *option*
Sends a :set-edges message to the window making it the full inside size of its superior. *option* is passed directly to the :set-edges message.

:set-size *new-width new-height* &optional *option*
Sends a :set-edges message to the window setting it to the specified size without moving its upper-left corner. *option* is passed directly to the :set-edges message.

:set-inside-size *new-width new-height* &optional *option*
Sends a :set-edges message to the window setting it to the specified size not including the margins. The upper-left corner does not move. *option* is passed directly to the :set-edges message.

:set-position *new-x new-y* &optional *option*
Sends a :set-edges message to the window setting its position to the specified place. *option* is passed directly to the :set-edges message.

:center-around *x y*
Positions the center of the window as close to *x* and *y* as is possible without hanging off the edge of the superior.

:change-of-size-or-margins &rest *options*
This message is sent by the system whenever the size of the inside part of a window, or anything about its margins, is changed. The primary method actually does the changes, moves the inside bits around as necessary, and blanks out the margins. You can define :after daemons for this message to do such things as modification of internal data structures that depend on the number of lines that fit in the window. Normal code should never redefine the primary method nor send the message directly. *options* is a list of alternating keywords and values specifying what is changing, similar to the options used when creating a window.

:string-out *string* &optional (*start* 0) (*end* nil)
Outputs *string*. *start* and *end* specify a substring of the string. More efficient than character by character output.

:line-out *string* &optional (*start* 0) (*end* nil)
Outputs *string* followed by a newline. *start* and *end* specify a substring of the string. More efficient than outputting the string character by character.

:clear-screen &optional *margins*
Erases the window and homes its cursor. If *margins* is nil (the default), the inside of the window is erased and the margins are left alone. If *margins* is t, the margins are also erased.

:clear-eof
Erases from the current position of the cursor to the end of the window.

:clear-eol
Erases from the current cursor position to the end of the line.

:clear-char
Erases the character position under the cursor. In case of multiple or variable width fonts, this may not be the actual width of the character there.

:home-cursor
Positions the cursor to the upper-leftmost character position in the window, inside the margins.

:read-cursorpos &optional (*units* ':pixel)
Returns the current cursor position as two values, *x* and *y*. The cursor position is relative to the upper-left-hand corner of the window inside the margins. The units of measurement may be specified as :pixel or :character.

:set-cursorpos *x y* &optional (*units* ':pixel)
Puts the cursor at the specified position. *units* the same as for :read-cursorpos.

:fresh-line
If the cursor is not at the beginning of the line, advances to the next line. In either case does a :clear-eol.

:draw-rectangle *width height x y alu*
Makes a rectangle of 1-bits of the specified dimensions, and merges it into the window at the specified *x,y* position using the *alu* function supplied. The position is relative to the *outside* of the window, unlike the position returned by :read-cursorpos. This is useful for erasing, darkening, and complementing rectangular areas of a window. The rectangle is clipped if it would lie outside of the window.

:more-exception
Called when more-vpos is reached. Prompts with **more** and waits for the user to type a character before continuing. Resets more-vpos.

:note-input-wait
Called when :tyi hangs waiting for input. Sets more-vpos appropriately.

:output-hold-exception
Called when output is attempted on the window but either the output hold flag is set or the window is locked by a temporary window. If the latter is true, then waits until the window is no longer locked. If the former, and the window is deexposed, then deexposed-typeout-action is inspected (see <not-yet-written>). Always returns with the window no longer output held.

:delete-line &optional *(n 1)*
Deletes lines at and below the current cursor position. *n* specifies the number of lines to delete. Lines below the deleted lines are shifted up, and blank lines are brought in at the bottom of the window.

:insert-line &optional *(n 1)*
Makes *n* blank lines at the cursor by shifting the lines at and below the cursor down. *n* lines at the bottom of the window are lost.

The following instance variables are relevant.

tv:cursor-x *Variable*
tv:cursor-y *Variable*
The position at which to put the next character. These are relative to the upper-left-hand corner of the window *outside* the margins, unlike the values returned by the :read-cursorpos message.

tv:more-vpos *Variable*
The Y position at which a **more** will happen, or 100000 plus the Y position if it is to be deferred until after the bottom of the window has been reached, or nil if there is no more-processing on this window.

tv:current-font *Variable*
The currently selected font for character display.

tv:font-map *Variable*
An array of fonts. The 0'th entry is the "standard" font.

:vsp *vsp*
Selects the number of blank raster lines between character lines. The default is 2. The line-height of a window is initialized from this and the height of the tallest font initially specified.

:more-p *t-or-nil*
Enables or disables more-processing. The default is t, but many flavors change the default to nil for their own purposes.

[Exactly what forms of typeout are controlled by these next two?]

:right-margin-character-flag *t-or-nil*
The default is nil. If t, if a line is longer than the width of the window, when it wraps around to the next line an "!" is put in the right margin.

:truncate-line-out-flag *t-or-nil*
The default is nil, but if t when a line is longer than the width of the window it is truncated.

1.6 Lower-level Display Primitives

[Here will be explained prepare-sheet, the microcode primitives, and maybe some or all of the sheet-*numble* functions. Somewhere we are going to need sections on blinkers and fonts, also. Maybe here is a good place. Or maybe all the displaying-in-a-window stuff should be moved out into its own chapter?]

1.7 Character Input

Note that these operations are a superset of the standard *stream* protocol. Thus a window may be used directly as a stream.

:tyi &optional *eof*
Returns the next input character. Hangs until a character is available. The *eof* argument is ignored since keyboards do not have end-of-file.

:tyi-no-hang
Returns the next input character if one is immediately available, else nil. Never hangs.

:untyi *ch*
Returns *ch* to the head of the input stream. It will be the next character read. Only one character may be untyi'ed at a time.

:mouse-buttons *buttons-down x y*

If a button is pushed with the mouse over an exposed window that has a :mouse-buttons method, that window receives this message. *Buttons-down* is a bit-mask of the buttons pushed. *X* and *y* are the coordinates of the mouse relative to this window. The message is sent at the time the mouse button is first depressed. Encoding of double-clicks or deferring of command execution until the mouse button is released, if desired, must be done by this handler. It is a system convention that clicking the right-hand mouse button twice nearly always gets you the system menu. It is also a system convention that clicking the left-hand button on an unexposed mouse-selectable window exposes and selects it.

:mouse-moves *x y*

When the mouse moves while over an exposed window that handles :mouse-moves, it receives such a message with the window-relative coordinates of the mouse as arguments. The mouse-blinker must be moved by this method.

:handle-mouse

Sent in the mouse process to the window when the mouse moves into the window's area of influence, should track the mouse and send :mouse-moves and :mouse-buttons as appropriate. Usually calls tv:mouse-default-handler.

:set-mouse-position *x y*

Sets the mouse position to *x*, *y*. Coordinates are relative to the window.

1.9 Notification

Notification means telling the user about an unexpected occurrence, such as an error in a background process, by printing a message in some suitable place. The system provides for such messages to come out either on the selected window, if it agrees to accept them (Lisp listeners do), or on a special window popped-up for that purpose.

:notify-stream &optional *window-of-interest*

Sent to the selected window in order to get a stream via which to notify the user.

Windows like Lisp Listeners simply hand back themselves and ignore *window-of-interest*. Windows which don't want to be corrupted by extraneous output, though, usually include the pop-up-notification-mixin flavor, which creates a pop-up window for use as the stream, and also tells it *window-of-interest*, which is the window that the output will be on behalf of. A pop-up-notification window arranges to select *window-of-interest* when it is selected (e.g. by clicking on it with the mouse). The pop-up-notification-mixin is included in the window flavor.

nil — No border here.

t — The default function with the default thickness.

a number — The default function with the specified thickness.

a symbol — That function with its default thickness.

a cons (*function* . *thickness*)
That function with that thickness.

a list (*function left top right bottom*)
That function in the specified rectangular area. This is the internal form that everything else turns into, but if you specify this from the outside only the width and height implied by those four numbers will be paid attention to; the position comes from the relationship with other parts of the margin system.

The default (and currently only) border function is tv:draw-rectangular-border. Its default width is 1.

:border-margin-width *n-pixels*
The width of the white space in the margins between the borders and the inside of the window. The default is 1. This doesn't do anything unless there are borders.

:label *spec*
Controls the label. The default is t, which makes the label display the window's name in the lower-left corner. Choices are:

nil — No label.

t — A label with all the default characteristics.

:top — Put it at the top of the window.

:bottom — Put it at the bottom of the window. This is usually the default.

a string — The label is this string instead of the window's name.

a font — Specifies what font to display the label in.

a list (*left top right bottom font string*)
Specifies all of the options. This is the internal form everything else is turned into. Negative numbers mean up from the bottom or left from the right. tv:compute-label-position is the function which understands this. Externally you can control only the height and whether it goes at the top of the bottom; the position is controlled by interaction with the rest of the margin system.

The following messages are relevant.

Function Index

Window Creation Options