symbolics™

Volume 3
Lisp Language

Volume 3. Lisp Language

#996030

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

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

Contents

Lisp Language

symbolics™

Volume 3A
Lisp Language

Volume 3A. Lisp Language

#996030

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

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

Contents

Lisp Language

symbolics™

Documentation Map

1 System Index

TOC
Table of Contents

INDEX
Index

RN
Release Notes/ Patch Notes

NEWS
Newsletters/ Bug Reports

2 System Fundamentals

NOTA
Notation Conventions

LMS
Lisp Machine Summary 3600 Edition

3600
Notes on the 3600 for LM-2 Users

INED
Using the Input Editor

MISCF
Miscellaneous Useful Functions

3 Lisp Language

PRIM
Primitive Object Types

EVAL
Evaluation

FLOW
Flow of Control

ARR
Arrays and Strings

FUNC
Functions

MAC
Macros

DEFS
Defstruct

FLAV
Objects, Message Passing, and Flavors

COND
Conditions

PKG
Packages

4 Program Development Tools

TOOLS
Program Development Tools and Techniques

HELP
Program Development Help Facilities

ZMACS
Zmacs Manual

DEBUG
Debugger

MAINT
Maintaining Large Systems

COMP
The Compiler

MISCT
Other Tools

5 User Interface Support

WINDOC
Using the Window System

WINDEX
Window System Program Examples

MENUS
Window System Choice Facilities

SCROLL
Scroll Windows

MISCUI
Miscellaneous Functions

6 Utilities and Applications

ZMAILT
Zmail Tutorial and Reference Manual

ZMAILC
Zmail Concepts and Techniques

FED
Font Editor

HARD
Hardcopy System

CONV
Converse

FSED
FSEdit

MISCU
Other Utilities and Applications

7 Networks and I/O

STR
Streams

FILE
Files

NETIO
Networks and Peripherals

PROT
Networks and Protocols

8 System Installation, Maintenance, Programming

SIG
Software Installation Guide

SITE
Site Operations

TAPE
Tape

STOR
Storage Management

PROC
Processes

INIT
Initializations

INT
Internals

Map to the New Documentation System

The documentation in this eight-volume set includes all previously published Lisp Machine documentation, reorganized by topics and intended use of the information. (In addition, some documents contain information that is new as of Release 5.0.) The most obvious aspects of the reorganization are:

- The *Lisp Machine Manual* has been taken apart, and its various chapters are now scattered throughout the new system.
- Release Notes and Patch Notes through Release 5.0, which had previously been bound separately, have been merged into their relevant sources.

Following is a mapping of old to new documents, listed in alphabetic order by old document title:

Old title	New title	Mnemonic	Volume
Chaosnet	*Networks and Peripherals*	NETIO	7
Chaosnet File Protocol	*Networks and Protocols*	PROT	7
Font Editor	*Font Editor*	FED	6
Front-End Processor	*Networks and Peripherals*	NETIO	7
Introduction to Using the Window System	*Using the Window System*	WINDOC	5
Lisp Machine Choice Facilities	*Window System Choice Facilities*	MENUS	5
Lisp Machine Manual	[See page LMM.]		
Lisp Machine Summary 3600 Edition	*Lisp Machine Summary 3600 Edition*	LMS	2
LM-2 Serial I/O	*Networks and Peripherals*	NETIO	7
LM-2 UNIBUS I/O	*Networks and Peripherals*	NETIO	7
Notes on the 3600 for LM-2 Users	*Notes on the 3600 for LM-2 Users*	3600	2
Operating the Lisp Machine	[Discontinued.]		
Program Development Help Facilities	*Program Development Help Facilities*	HELP	4

Old title	New title	Mnemonic	Volume
Program Development Tools and Techniques	*Program Development Tools and Techniques*	TOOLS	4
Release Notes for System 78	[Merged into related documents.]		
Release 4.0 Release Notes	[Merged into related documents.]		
Release 4.1 Patch Notes	[Merged into related documents.]		
Release 4.2 Patch Notes	[Merged into related documents.]		
Release 4.3 Patch Notes	[Merged into related documents.]		
Release 4.4 Patch Notes	[Merged into related documents.]		
Release 4.5 Patch Notes	[Merged into related documents.]		
Scroll Windows	*Scroll Windows*	SCROLL	5
Signalling and Handling Conditions	*Conditions*	COND	3
Software Installation Guide	*Software Installation Guide*	SIG	8
Symbolics File System	*Files*	FILE	7
System 210 Release Notes	[Merged into related documents.]		
Window System Program Examples	*Window System Program Examples*	WINDEX	5
Zmail Concepts and Techniques	*Zmail Concepts and Techniques*	ZMAILC	5
Zmail Tutorial and Reference Manual	*Zmail Tutorial and Reference Manual*	ZMAILT	5
Zmacs Manual	*Zmacs Manual*	ZMACS	4

Lisp Machine Manual

[Has been separated, by chapter, into the following documents:]

Old chapter title	Pages	New document title	Mnemonic	Volume
1. Introduction	1-6	*Notation Conventions*	NOTA	2
2. Primitive Object Types	7-12	*Primitive Object Types*	PRIM	3
3. Evaluation	13-32	*Evaluation*	EVAL	3
4. Flow of Control	33-51	*Flow of Control*	FLOW	3
5. Manipulating List Structure	52-85	*Primitive Object Types*	PRIM	3
6. Symbols	86-91	*Primitive Object Types*	PRIM	3
7. Numbers	92-106	*Primitive Object Types*	PRIM	3
8. Arrays	107-125	*Arrays and Strings*	ARR	3
9. Strings	126-135	*Arrays and Strings*	ARR	3
10. Functions	136-157	*Functions*	FUNC	3
11. Closures	158-162	*Functions*	FUNC	3
12. Stack Groups	163-169	*Internals*	INT	8
13. Locatives	170-171	*Primitive Object Types*	PRIM	3
14. Subprimitives	172-191	*Internals*	INT	8
15. Areas	192-196	*Storage Management*	STOR	8
16. The Compiler	197-207	*The Compiler*	COMP	4
17. Macros	208-232	*Macros*	MAC	3
18. The LOOP Iteration Macro	233-256	*Flow of Control*	FLOW	3

Old chapter title	Pages	New document title	Mnemonic	Volume
19. Defstruct	257-278	*Defstruct*	DEFS	3
20. Objects, Message Passing, and Flavors	279-313	*Objects, Message Passing, and Flavors*	FLAV	3
21. The I/O System				
21.1	314-318	*Streams*	STR	7
21.2	319-331	*Primitive Object Types*	PRIM	3
21.3-21.10	331-375	*Streams*	STR	7
22. Naming of Files	376-391	*Files*	FILE	7
23. Packages	392-405	*Packages*	PKG	3
24. Maintaining Large Systems				
24.1-24.7	406-421	*Maintaining Large Systems*	MAINT	4
24.8	422-427	*Site Operations*	SITE	8
25. Processes	428-439	*Processes*	PROC	8
26. Errors and Debugging				
26.1	440-450	*Conditions*	COND	3
26.2-26.8	450-468	*Debugger*	DEBUG	4
27. How to Read Assembly Language	469-486	*Internals*	INT	8
28. Querying the User	487-489	*Miscellaneous Functions*	MISCUI	5
29. Initializations	490-492	*Initializations*	INIT	8
30. Dates and Times	493-498	*Miscellaneous Functions*	MISCUI	5
31. Miscellaneous Useful Functions				
31.1-31.3	499-504	*Miscellaneous Useful Functions*	MISCF	2
31.4	505	*Storage Management*	STOR	8
31.5-31.7	506-508	*Miscellaneous Useful Functions*	MISCF	2

symbolics™

PRIM Primitive Object Types

Cambridge, Massachusetts

Primitive Object Types

 # 990053

March 1984

This document corresponds to Release 5.0.

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

No representation or affirmation of fact contained in this document should be construed as a warranty by Symbolics, and its contents are subject to change without notice. Symbolics, Inc. assumes no responsibility for any errors that might appear in this document.

Symbolics software described in this document is furnished only under license, and may be used only in accordance with the terms of such license. Title to, and ownership of, such software shall at all times remain in Symbolics, Inc. Nothing contained herein implies the granting of a license to make, use, or sell any Symbolics equipment or software.

Symbolics is a trademark of Symbolics, Inc., Cambridge, Massachusetts.

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

Table of Contents

1. Data Types

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

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**. This string serves as the *printed representation* of the symbol. See the section "What the Printer Produces". Each symbol has a *binding* (sometimes also called the "value"), which may be any Lisp object. It is also referred to as the "contents of the value cell", since internally every symbol has a cell called the *value cell* that holds the binding. It is accessed by the **symeval** function and updated by the **set** function. (That is, given a symbol, you use **symeval** to find out what its binding is, and use **set** to change its binding.) Each symbol has a *definition*, which may also be any Lisp object. It is also referred to as the "contents of the function cell", since internally every symbol has a cell called the *function cell* that holds the definition. The definition can be accessed by the **fsymeval** function and updated with **fset**. Usually the functions **fdefinition** and **fdefine** are employed. The property list is a list of an even number of elements; it can be accessed directly by **plist** and updated directly by **setplist**. Usually the functions **get**, **putprop**, and **remprop** are used. The property list is used to associate any number of additional attributes with a symbol — attributes not used frequently enough to deserve their own cells as the value and definition do. Symbols also have a package cell, which indicates which "package" of names the symbol belongs to. This is explained further in the section on packages and can be disregarded by the casual user. See the document *Packages*.

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

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

There are several kinds of numbers in Zetalisp. *Fixnums* represent integers in the range of -2^23 to 2^23-1. *Bignums* represent integers of arbitrary size, but they are more expensive to use than fixnums because they occupy storage and are slower. The system automatically converts between fixnums and bignums as required. *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 the section

"Numbers". Full details of these types and the conversions between them are discussed there.

The usual form of compiled, executable 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 and are usually found as the definitions of symbols. See the document *The Compiler*. The printed representation of a FEF includes its name, so that it can be identified.

Another Lisp object that represents executable code is a "microcode 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 any of these compiled code objects is to *apply* it to arguments. However, some functions are provided for examining such objects, for user convenience. See the function **arglist**. See the function **args-info**. See the function **describe**. See the function **disassemble**.

A *locative* is a kind of a pointer to a single memory cell anywhere in the system. See the section "Locatives". The contents of this cell can be accessed by **cdr** and updated by **rplacd**.

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

A *list* is not a primitive data type, but rather a data structure made up of conses and the symbol **nil**. See the section "Manipulating List Structure".

2. Predicates

A *predicate* is a function that tests for some condition involving its arguments and returns the symbol **t** if the condition is true, or the symbol **nil** if it is not true. Most of the following predicates are for testing what data type an object has; some other general-purpose predicates are also explained.

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

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

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

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

listp *arg* *Function*
listp returns **t** if its argument is a cons, otherwise **nil**. Note that this means **(listp nil)** is **nil** even though **nil** is the empty list. [This may be changed in the future.]

nlistp *arg* *Function*
nlistp returns **t** if its argument is anything besides a cons, otherwise **nil**. **nlistp** is identical to **atom**, and so **(nlistp nil)** returns **t**. [This may be changed in the future, if and when **listp** is changed.]

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

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

fixp *arg* *Function*
fixp returns **t** if its argument is a fixed-point number, that is, a fixnum or a bignum, otherwise **nil**.

floatp *arg* *Function*
floatp returns **t** if its argument is a floating-point number, that is, a flonum or a small flonum on the LM-2 or a single- or double-precision floating-point number on the 3600. Otherwise it returns **nil**.

fixnump *arg* *Function*

fixnump returns **t** if its argument is a fixnum, otherwise **nil**.

bigp *arg* *Function*

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

flonump *arg* *Function*

flonump returns **t** if *arg* is a (large) flonum, otherwise **nil**.

small-floatp *arg* *Function*

(LM-2 only) **small-floatp** returns **t** if *arg* is a small flonum, otherwise **nil**.

sys:single-float-p *arg* *Function*

(3600 only) Returns **t** if *arg* is a single-precision floating-point number, otherwise **nil**.

sys:double-float-p *arg* *Function*

(3600 only) Returns **t** if *arg* is a double-precision floating-point number, otherwise **nil**.

stringp *arg* *Function*

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

arrayp *arg* *Function*

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

functionp *arg* &optional *allow-special-forms* *Function*

functionp returns **t** if its argument is a function (essentially, something that is acceptable as the first argument to **apply**), otherwise it returns **nil**. In addition to interpreted, compiled, and microcoded functions, **functionp** is true of closures, select-methods, and symbols whose function definition is **functionp**. See the section "Other Kinds of Functions". **functionp** is not true of objects that can be called as functions but are not normally thought of as functions: arrays, stack groups, entities, and instances. If *allow-special-forms* is specified and non-**nil**, then **functionp** will be true of macros and special-form functions (those with quoted arguments). Normally **functionp** returns **nil** for these since they do not behave like functions. As a special case, **functionp** of a symbol whose function definition is an array returns **t**, because in this case the array is being used as a function rather than as an object.

subrp *arg* *Function*

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

closurep *arg* *Function*

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

entityp *arg* *Function*

(LM-2 only) **entityp** returns **t** if its argument is an entity, otherwise **nil**. See the section "Entities".

locativep *arg* *Function*

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

errorp *object* *Function*

errorp returns **t** if *object* is an error object, and **nil** otherwise. That is:

```
(errorp x) <=> (typep x 'error)
```

typep *arg* &optional *type* *Function*

typep is really two different functions. With one argument, **typep** is not really a predicate; it returns a symbol describing the type of its argument. With two arguments, **typep** is a predicate that returns **t** if *arg* is of type *type*, and **nil** otherwise. Note that an object can be "of" more than one type, since one type can be a subset of another.

The symbols that can be returned by **typep** of one argument are:

:symbol *arg* is a symbol.

:fixnum *arg* is a fixnum (not a bignum).

:bignum *arg* is a bignum.

:flonum (LM-2 only) *arg* is a flonum (not a small-flonum).

:small-flonum (LM-2 only) *arg* is a small flonum.

:single-float (3600 only) *arg* is a single-precision floating-point number.

:double-float (3600 only) *arg* is a double-precision floating-point number.

:list *arg* is a cons.

:locative *arg* is a locative pointer. See the section "Locatives".

:compiled-function
arg is the machine code for a compiled function (sometimes called a FEF).

:microcode-function
arg is a function written in microcode.

:closure *arg* is a closure. See the section "Closures".

:select-method *arg* is a select-method table. See the section "Other Kinds of Functions".

:stack-group *arg* is a stack-group. See the section "Stack Groups".

:string	*arg* is a string.
:array	*arg* is an array that is not a string.
:random	Returned for 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). The primitive type of the object could be array, instance, or entity. See the section "Named Structures". See the document *Objects, Message Passing, and Flavors.*

The *type* argument to **typep** of two arguments can be any of the above keyword symbols (except for **:random**), the name of a user-defined data type (either a named structure or a flavor), or one of the following additional symbols:

:atom	Any atom (as determined by the **atom** predicate).
:fix	Any kind of fixed-point number (fixnum or bignum).
:float	Any kind of floating-point number (flonum or small-flonum).
:number	Any kind of number.
:instance	An instance of any flavor. See the document *Objects, Message Passing, and Flavors.*
:entity	An entity. **typep** of one argument returns the name of the particular user-defined type of the entity, rather than **:entity**.
:null	**nil** is the only value that has this type.

See also **data-type**.

Note that **(typep nil) => :symbol**, and **(typep nil ':list) => nil**; the latter may be changed.

The following functions are some other general purpose predicates:

eq *x y* *Function*

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

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

Note that in Zetalisp equal fixnums are **eq**; this is not true in Maclisp. Equality does not imply **eq**ness for other types of numbers. To compare numbers, use =. See the section "Numeric Comparisons".

neq *x y* *Function*

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

eql *x y* *Function*

eql returns **t** if is arguments are **eq**, or if they are numbers of the same type with the same value, or (in Common Lisp) if they are character objects that represent the same character. The predicate = compares the values of two numbers even if the numbers are of different types. Use **equal** or **string-equal** to compare the characters of two strings.

Examples:

```
(eql 'a 'a) => t
(eql 3 3)   => t
(eql 3 3.0) => nil
(eql 3.0 3.0) => t
(eql #/a #/a) => t
(eql (cons 'a 'b) (cons 'a 'b)) => nil
(eql "foo" "FOO") => nil
```

The following expressions might return either **t** or **nil**:

```
(eql '(a . b) '(a . b))
(eql "foo" "foo")
```

In Zetalisp:

```
(eql 1.0s0 1.0d0) => nil
(eql 0.0 -0.0) => nil
```

equal *x y* *Function*

The **equal** predicate returns **t** if its arguments are similar (isomorphic) objects. See the function **eq**. Two numbers are **equal** if they have the same value and type (for example, a flonum is never **equal** to a fixnum, even if = is true of them). For conses, **equal** is defined recursively as the two **car**s being **equal** and the two **cdr**s being equal. Two strings are **equal** if they have the same length, and the characters composing them are the same. See the function **string-equal**. Alphabetic case is ignored. See the variable **alphabetic-case-affects-string-comparison**. All other objects are **equal** if and only if they are **eq**. Thus **equal** could have been defined by:

```
(defun equal (x y)
  (cond ((eq x y) t)
        ((neq (typep x) (typep y)) nil)
        ((numberp x) (= x y))
        ((stringp x) (string-equal x y))
        ((listp x) (and (equal (car x) (car y))
                        (equal (cdr x) (cdr y))))))
```

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

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

not *x* *Function*

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 should not make understanding of your program depend on this. 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.

See the function **null**.

null *x* *Function*

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 should not make understanding of your program depend on this. 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. Manipulating List Structure

This chapter discusses functions that manipulate conses, and higher-level structures made up of conses, such as lists and trees. It also discusses hash tables and resources, which are related facilities.

A cons is a primitive Lisp data object that is extremely simple: it knows about two other objects, called its car and its cdr.

A list is recursively defined to be the symbol **nil**, or a cons whose cdr is a list. A typical list is a chain of conses: the cdr of each is the next cons in the chain, and the cdr of the last one is the symbol **nil**. The cars of each of these conses are called the *elements* of the list. A list has one element for each cons; the empty list, **nil**, has no elements at all. Here are the printed representations of some typical lists:

```
(foo bar)                ;This list has two elements.
(a (b c d) e)            ;This list has three elements.
```

Note that the second list has three elements: **a**, **(b c d)**, and **e**. The symbols **b**, **c**, and **d** are *not* elements of the list itself. (They are elements of the list that is the second element of the original list.)

A "dotted list" is like a list except that the cdr of the last cons does not have to be **nil**. This name comes from the printed representation, which includes a "dot" character. Here is an example:

```
(a b . c)
```

This "dotted list" is made of two conses. The car of the first cons is the symbol **a**, and the cdr of the first cons is the second cons. The car of the second cons is the symbol **b**, and the cdr of the second cons is the symbol **c**.

A tree is any data structure made up of conses whose cars and cdrs are other conses. The following are all printed representations of trees:

```
(foo . bar)
((a . b) (c . d))
((a . b) (c d e f (g . 5) s) (7 . 4))
```

These definitions are not mutually exclusive. Consider a cons whose car is **a** and whose cdr is **(b (c d) e)**. Its printed representation is:

```
(a b (c d) e)
```

It can be thought of and treated as a cons, or as a list of four elements, or as a tree containing six conses. You can even think of it as a "dotted list" whose last cons just happens to have **nil** as a cdr. Thus, lists and "dotted lists" and trees are not fundamental data types; they are just ways of thinking about structures of conses.

A circular list is like a list except that the cdr of the last cons, instead of being **nil**, is the first cons of the list. This means that the conses are all hooked together in a

ring, with the cdr of each cons being the next cons in the ring. While these are perfectly good Lisp objects, and there are functions to deal with them, many other functions will have trouble with them. Functions that expect lists as their arguments often iterate down the chain of conses waiting to see a **nil**, and when handed a circular list this can cause them to compute forever. The printer is one of these functions; if you try to print a circular list the printer will never stop producing text. See the section "Output Functions". You must use circular lists carefully.

The Lisp Machine internally uses a storage scheme called "cdr coding" to represent conses. This scheme is intended to reduce the amount of storage used in lists. The use of cdr-coding is invisible to programs except in terms of storage efficiency; programs will work the same way whether or not lists are cdr-coded or not. Several of the functions below mention how they deal with cdr-coding. You can completely ignore all this if you want. However, if you are writing a program that allocates a lot of conses and you are concerned with storage efficiency, you may want to learn about the cdr-coded representation and how to control it. See the section "Cdr-coding".

3.1 Conses

car *x* *Function*

Returns the *car* of *x*. Example:

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

Officially **car** is applicable only to conses and locatives. However, as a matter of convenience, **car** of **nil** return **nil**.

cdr *x* *Function*

Returns the *cdr* of *x*. Example:

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

Officially **cdr** is applicable only to conses and locatives. However, as a matter of convenience, **cdr** of **nil** return **nil**.

caaaar *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

caaadr *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

caaar *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

caadar *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

caaddr *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

caadr *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

caar *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cadaar *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cadadr *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cadar *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

caddar *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cadddr *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

caddr *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cadr *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cdaaar *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cdaadr *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cdaar *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cdadar *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cdaddr *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cdadr *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cdar *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cddaar *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cddadr *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cddar *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cdddar *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cddddr *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cdddr *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cddr *x* *Function*

All 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**. See the function **car**. See the function **cdr**.

cons *x y* *Function*

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* *Function*

(ncons *x*) is the same as **(cons *x* nil)**. The name of the function is from "nil-cons".

xcons *x y* *Function*

xcons ("exchanged cons") is like **cons** except that the order of the arguments is reversed. Example:

```
(xcons 'a 'b) => (b . a)
```

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

This function creates a *cons* in a specific *area*. (Areas are an advanced feature of storage management.) See the section "Areas". 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* *Function*

(ncons-in-area *x area-number***)** = **(cons-in-area** *x* **nil** *area-number***)**

xcons-in-area *x y area-number* *Function*

(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. See the document *Macros*.

car-location *cons* *Function*

car-location returns a locative pointer to the cell containing the car of *cons*.

Note: there is no **cdr-location** function; it is difficult because of the cdr-coding scheme. See the section "Cdr-coding".

3.2 Lists

length *list* *Function*

length returns the length of *list*. The length of a list is the number of elements 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) ))
```

except that it is an error to take **length** of a non-**nil** atom.

first *list* *Function*

This function takes a list as an argument, and returns the first element of the list. **first** is identical to **car**. The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

second *list* *Function*

This function takes a list as an argument, and returns the second element of the list. **second** is identical to **cadr**. The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

third *list* *Function*

This function takes a list as an argument, and returns the third element of the list. The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

fourth *list* *Function*

This function takes a list as an argument, and returns the fourth element of the list. The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

fifth *list* *Function*

This function takes a list as an argument, and returns the fifth element of the list. The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

sixth *list* *Function*

This function takes a list as an argument, and returns the sixth element of the list. The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

seventh *list* *Function*

This function takes a list as an argument, and returns the seventh element of the list. The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

rest1 *list* *Function*

rest1 returns the rest of the elements of a list, starting with element 1 (counting the first element as the zeroth). Thus **rest1** is identical to **cdr**. The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

rest2 *list* *Function*

rest2 returns the rest of the elements of a list, starting with element 2

(counting the first element as the zeroth). Thus **rest2** is identical to **cddr**. The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

rest3 *list* *Function*

rest3 returns the rest of the elements of a list, starting with element 3 (counting the first element as the zeroth). The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

rest4 *list* *Function*

rest4 returns the rest of the elements of a list, starting with element 4 (counting the first element as the zeroth). The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

nth *n list* *Function*

(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
```

If *n* is greater than the length of the list, **nil** is returned.

Note: this is not the same as the Interlisp function called **nth**, which is similar to but not exactly the same as the Lisp Machine function **nthcdr**. Also, some people have used macros and functions called **nth** of their own in their Maclisp programs, which may not work the same way; be careful.

nth could have been defined by:

```
(defun nth (n list)
  (do ((i n (1- i))
       (l list (cdr l)))
      ((zerop i) (car l))))
```

nthcdr *n list* *Function*

(nthcdr *n list***)** cdrs *list n* times, and returns the result. Examples:

```
(nthcdr 0 '(a b c)) => (a b c)
(nthcdr 2 '(a b c)) => (c)
```

In other words, it returns the *n*th cdr of the list. If *n* is greater than the length of the list, **nil** is returned.

This is similar to Interlisp's function **nth**, except that the Interlisp function is one-based instead of zero-based; see the Interlisp manual for details. **nthcdr** could have been defined by:

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

last *list* *Function*

last returns the last cons of *list*. If *list* is **nil**, it returns **nil**. Note that **last** is unfortunately *not* analogous to **first** (**first** returns the first element of a list, but **last** does not return the last element of a list); this is a historical artifact. 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))) ))
```

list &rest *args* *Function*

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 (length args))))
      (do ((l list (cdr l))
           (a args (cdr a)))
          ((null a) list)
        (rplaca l (car a)))))
```

list* &rest *args* *Function*

list* is like **list** except that the last cons of the constructed list is "dotted". It must be given at least one argument. Example:

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

This is like

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

More examples:

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

list-in-area *area-number* &rest *args* *Function*

list-in-area is exactly the same as **list** except that it takes an extra argument, an area number, and creates the list in that area.

list*-in-area *area-number* &rest *args* *Function*

list*-in-area is exactly the same as **list*** except that it takes an extra argument, an area number, and creates the list in that area.

make-list *length* &rest *options* *Function*

This creates and returns a list containing *length* elements. *length* should be a fixnum. *options* are alternating keywords and values. The keywords may be either of the following:

:area The value specifies in which area the list should be created. See the section "Areas". It should be either an area number (a fixnum), or **nil** to mean the default area.

:initial-value
The elements of the list will all be this value. It defaults to **nil**.

make-list always creates a *cdr-coded* list. See the section "Cdr-coding". Examples:

```
(make-list 3) => (nil nil nil)
(make-list 4 ':initial-value 7) => (7 7 7 7)
```

When **make-list** was originally implemented, it took exactly two arguments: the area and the length. This obsolete form is still supported so that old programs will continue to work, but the new keyword-argument form is preferred.

circular-list &rest *args* *Function*

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. **circular-list** could have been defined by:

```
(defun circular-list (&rest elements)
  (setq elements (copylist* elements))
  (rplacd (last elements) elements)
  elements)
```

copylist *list* &optional *area force-dotted* *Function*

Returns a list that is **equal** to *list*, but not **eq**. **copylist** does not copy any elements of the list: only the conses of the list itself. The returned list is fully cdr-coded to minimize storage. See the section "Cdr-coding". If the list is "dotted", that is, **(cdr (last** *list*)) is a non-**nil** atom, this will be true of the returned list also. You may optionally specify the area in which to create the new copy.

copylist* *list* &optional *area* *Function*

This is the same as **copylist** except that the last cons of the resulting list is never cdr-coded. See the section "Cdr-coding". This makes for increased efficiency if you **nconc** something onto the list later.

copyalist *list* &optional *area* *Function*

copyalist is for copying association lists. See the section "Tables". The *list* is copied, as in **copylist**. In addition, each element of *list* that is a cons is replaced in the copy by a new cons with the same car and cdr. You may optionally specify the area in which to create the new copy.

copytree *tree* &optional *area* *Function*

copytree copies all the conses of a tree and makes a new tree with the same fringe. You may optionally specify the area in which to create the new copy.

reverse *list* *Function*

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. The list created by **reverse** is not cdr-coded. 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.
```

nreverse *list* *Function*

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

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

nreverse could have been defined by:

```
(defun nreverse  (x)
   (cond ((null x) nil)
         ((nreverse1 x nil))))

(defun nreverse1 (x y)          ; auxiliary function
   (cond ((null (cdr x)) (rplacd x y))
         ((nreverse1 (cdr x) (rplacd x y)))))
         ;; this last call depends on order of argument evaluation.
```

Currently, **nreverse** does something inefficient with cdr-coded lists, because it just uses **rplacd** in the straightforward way. See the section "Cdr-coding".

This may be fixed someday. In the meantime **reverse** might be preferable in some cases.

append &rest *lists* *Function*

The arguments to **append** are lists. The result is a list that is the concatenation of the arguments. The arguments are not changed (see **nconc**). Example:

```
(append '(a b c) '(d e f) nil '(g)) => (a b c d e f g)
```

append makes copies of the conses of all the lists it is given, except for the last one. So the new list will share the conses of the last argument to append, but all the other conses will be newly created. Only the lists are copied, not the elements of the lists.

A version of **append** that only accepts two arguments could have been defined by:

```
(defun append2 (x y)
   (cond ((null x) y)
         ((cons (car x) (append2 (cdr x) y)) )))
```

The generalization to any number of arguments could then be made (relying on **car** of **nil** being **nil**):

```
(defun append (&rest args)
  (if (< (length args) 2) (car args)
      (append2 (car args)
               (apply (function append) (cdr args)))))
```

These definitions do not express the full functionality of **append**; the real definition minimizes storage utilization by 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 is **nil** in which case *cdr-nil* is used. See the section "Cdr-coding".

To copy a list, use **copylist**; the old practice of using **append** to copy lists is unclear and obsolete.

nconc &rest *lists* *Function*

nconc takes lists as arguments. It returns a list that is the arguments concatenated together. The arguments are changed, rather than copied. See the function **append**. 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.
```

nreconc *x y* *Function*

(nreconc *x y***)** is exactly the same as **(nconc (nreverse** *x***)** *y***)** except that it is more efficient. Both *x* and *y* should be lists.

nreconc could have been defined by:

```
(defun nreconc (x y)
   (cond ((null x) y)
         ((nreverse1 x y)) ))
```

using the same **nreverse1** as above.

butlast *list* *Function*

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* *Function*

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* *Function*

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)
```

nleft *n list* &optional *tail* *Function*

Returns a "tail" of *list*, that is, one of the conses that makes up *list*, or **nil**. **(nleft** *n list***)** returns the last *n* elements of *list*. If *n* is too large, **nleft** will return *list*.

(nleft *n list tail***)** takes cdr of *list* enough times that taking *n* more cdrs would yield *tail*, and returns that. You can see that when *tail* is **nil** this is the same as the two-argument case. If *tail* is not **eq** to any tail of *list*, **nleft** will return **nil**.

ldiff *list sublist* *Function*

list should be a list, and *sublist* should be 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.

3.3 Alteration of List Structure

The functions **rplaca** and **rplacd** are used to make alterations in existing list structure, that is, to change the cars and cdrs of existing conses.

The structure is not copied but is physically altered; hence you should be cautious when using these functions, as strange side effects can occur if portions of list structure become shared unknown to you. The **nconc**, **nreverse**, **nreconc**, and **nbutlast** functions and the **delq** family have the same property.

rplaca *x y* *Function*

(rplaca *x y***)** changes the car of *x* to *y* and returns (the modified) *x*. *x* must be a cons or a locative. *y* may be any Lisp object. Example:

```
(setq g '(a b c))
(rplaca (cdr g) 'd) => (d c)
Now g => (a d c)
```

rplacd *x y* *Function*

(rplacd *x y***)** changes the cdr of *x* to *y* and returns (the modified) *x*. *x* must be a cons or a locative. *y* may be any Lisp object. Example:

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

subst *new old tree* *Function*

(subst *new old tree*) substitutes *new* for all occurrences of *old* in *tree*, and returns the modified copy of *tree*. The original *tree* is unchanged, as **subst** recursively copies all of *tree* replacing elements **equal** to *old* as it goes. Example:

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

subst could have been defined by:

```
(defun subst (new old tree)
    (cond ((equal tree old) new) ;if item equal to old, replace.
          ((atom tree) tree)     ;if no substructure, return arg.
          ((cons (subst new old (car tree))  ;otherwise recurse.
                 (subst new old (cdr tree))))))
```

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

To copy a tree, use **copytree**; the old practice of using **subst** to copy trees is unclear and obsolete.

Note: certain details of **subst** may be changed in the future. It may possibly be changed to use **eq** rather than **equal** for the comparison, and possibly may substitute only in cars, not in cdrs. This is still being discussed.

nsubst *new old tree* *Function*

nsubst is a destructive version of **subst**. The list structure of *tree* is altered by replacing each occurrence of *old* with *new*. **nsubst** could have been defined as

```
(defun nsubst (new old tree)
    (cond ((eq tree old) new)     ;if item eq to old, replace.
          ((atom tree) tree)      ;if no substructure, return arg.
          (t                      ;otherwise, recurse.
             (rplaca tree (nsubst new old (car tree)))
             (rplacd tree (nsubst new old (cdr tree)))
             tree)))
```

sublis *alist tree* *Function*

sublis makes substitutions for symbols in a tree. The first argument to **sublis** is an association list. See the section "Tables". The second argument is the tree in which substitutions are to be made. **sublis** looks at all symbols in the fringe of the tree; if a symbol appears in the association list, occurrences of it are replaced by the object with which it is associated. The argument is not modified; new conses are created where necessary and only where necessary, so the newly created tree shares as much of its substructure as possible with the old. For example, if no substitutions are made, the result is just the old tree. Example:

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

sublis could have been defined by:

```
(defun sublis (alist sexp)
  (cond ((symbolp sexp)
         (let ((tem (assq sexp alist)))
           (if tem (cdr tem) sexp)))
        ((listp sexp)
         (let ((car (sublis alist (car sexp)))
               (cdr (sublis alist (cdr sexp))))
           (if (and (eq (car sexp) car) (eq (cdr sexp) cdr))
               sexp
               (cons car cdr))))
        (t
         (sexp))))
```

nsublis *alist tree* *Function*

nsublis is like **sublis** but changes the original tree instead of creating new.

nsublis could have been defined by:

```
(defun nsublis (alist tree)
  (cond ((atom tree)
         (let ((tem (assq tree alist)))
           (if tem (cdr tem) tree)))
        (t (rplaca tree (nsublis alist (car tree)))
           (rplacd tree (nsublis alist (cdr tree)))
           tree)))
```

3.4 Cdr-coding

This section explains the internal data format used to store conses inside the Lisp Machine. It is only important to read this section if you require extra storage efficiency in your program.

The usual and obvious internal representation of conses in any implementation of Lisp is as a pair of pointers, contiguous in memory. If we call the amount of storage that it takes to store a Lisp pointer a "word", then conses normally occupy two words. One word (say it is the first) holds the car, and the other word (say it is the second) holds the cdr. To get the car or cdr of a list, you just reference this memory location, and to change the car or cdr, you just store into this memory location.

Very often, conses are used to store lists. If the above representation is used, a list of n elements requires two times n words of memory: n to hold the pointers to the elements of the list, and n to point to the next cons or to **nil**. To optimize this

particular case of using conses, the Lisp Machine uses a storage representation called "cdr coding" to store lists. The basic goal is to allow a list of *n* elements to be stored in only *n* locations, while allowing conses that are not parts of lists to be stored in the usual way.

The way it works is that there is an extra two-bit field in every word of memory, called the "cdr-code" field. There are three meaningful values that this field can have, which are called cdr-normal, cdr-next, and cdr-nil. The regular, noncompact way to store a cons is by two contiguous words, the first of which holds the car and the second of which holds the cdr. In this case, the cdr code of the first word is cdr-normal. (The cdr code of the second word does not matter; as we will see, it is never looked at.) The cons is represented by a pointer to the first of the two words. When a list of *n* elements is stored in the most compact way, pointers to the *n* elements occupy *n* contiguous memory locations. The cdr codes of all these locations are cdr-next, except the last location whose cdr code is cdr-nil. The list is represented as a pointer to the first of the *n* words.

Now, how are the basic operations on conses defined to work based on this data structure? Finding the car is easy: you just read the contents of the location addressed by the pointer. Finding the cdr is more complex. First you must read the contents of the location addressed by the pointer, and inspect the cdr-code you find there. If the code is cdr-normal, then you add one to the pointer, read the location it addresses, and return the contents of that location; that is, you read the second of the two words. If the code is cdr-next, you add one to the pointer, and simply return that pointer without doing any more reading; that is, you return a pointer to the next word in the *n*-word block. If the code is cdr-nil, you simply return **nil**.

If you examine these rules, you will find that they work fine even if you mix the two kinds of storage representation within the same list. There is no problem with doing that.

How about changing the structure? Like car, rplaca is very easy; you just store into the location addressed by the pointer. To do an rplacd you must read the location addressed by the pointer and examine the cdr code. If the code is cdr-normal, you just store into the location one greater than that addressed by the pointer; that is, you store into the second word of the two words. But if the cdr-code is cdr-next or cdr-nil, there is a problem: there is no memory cell that is storing the cdr of the cons. That is the cell that has been optimized out; it just does not exist.

This problem is dealt with by the use of "invisible pointers". An invisible pointer is a special kind of pointer, recognized by its data type (Lisp Machine pointers include a data type field as well as an address field). The way they work is that when the Lisp Machine reads a word from memory, if that word is an invisible pointer then it proceeds to read the word pointed to by the invisible pointer and use that word instead of the invisible pointer itself. Similarly, when it writes to a location, it first reads the location, and if it contains an invisible pointer then it writes to the

location addressed by the invisible pointer instead. (This is a somewhat simplified explanation; actually there are several kinds of invisible pointer that are interpreted in different ways at different times, used for things other than the cdr coding scheme.)

Here is how to do an rplacd when the cdr code is cdr-next or cdr-nil. Call the location addressed by the first argument to rplacd *l*. First, you allocate two contiguous words (in the same area that *l* points to). Then you store the old contents of *l* (the car of the cons) and the second argument to rplacd (the new cdr of the cons) into these two words. You set the cdr-code of the first of the two words to cdr-normal. Then you write an invisible pointer, pointing at the first of the two words, into location *l*. (It does not matter what the cdr-code of this word is, since the invisible pointer data type is checked first, as we will see.)

Now, whenever any operation is done to the cons (car, cdr, rplaca, or rplacd), the initial reading of the word pointed to by the Lisp pointer that represents the cons will find an invisible pointer in the addressed cell. When the invisible pointer is seen, the address it contains is used in place of the original address. So the newly allocated two-word cons will be used for any operation done on the original object.

Why is any of this important to users? In fact, it is all invisible to you; everything works the same way whether or not compact representation is used, from the point of view of the semantics of the language. That is, the only difference that any of this makes is in efficiency. The compact representation is more efficient in most cases. However, if the conses are going to get rplacd'ed, then invisible pointers will be created, extra memory will be allocated, and the compact representation will be seen to degrade storage efficiency rather than improve it. Also, accesses that go through invisible pointers are somewhat slower, since more memory references are needed. So if you care a lot about storage efficiency, you should be careful about which lists get stored in which representations.

You should try to use the normal representation for those data structures that will be subject to rplacding operations, including **nconc** and **nreverse**, and the compact representation for other structures. The functions **cons**, **xcons**, **ncons**, and their area variants make conses in the normal representation. The functions **list**, **list***, **list-in-area**, **make-list**, and **append** use the compact representation. The other list-creating functions, including **read**, currently make normal lists, although this might get changed. Some functions, such as **sort**, take special care to operate efficiently on compact lists (**sort** effectively treats them as arrays). **nreverse** is rather slow on compact lists, currently, since it simply uses **rplacd**, but this will be changed.

(copylist *x***)** is a suitable way to copy a list, converting it into compact form. See the function **copylist**.

3.5 Tables

Zetalisp includes functions that 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 can be 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. For example:

```
((tweety . bird) (sylvester . cat))
```

is an association list with two elements. Given a symbol representing the name of an animal, it can retrieve what kind of animal this is.

Structured records can be stored as association lists or as stereotyped cons-structures where each element of the structure has a certain car-cdr path associated with it. However, these are better implemented using structure macros. See the document *Defstruct*.

Simple list-structure is very convenient, but may not be efficient enough for large data bases because it takes a long time to search a long list. Zetalisp includes hash table facilities for more efficient but more complex tables, and a hashing function (**sxhash**) to aid you in constructing your own facilities. See the section "Hash Tables".

3.6 Lists as Tables

memq *item list* *Function*

(**memq** *item list*) returns **nil** if *item* is not one of the elements of *list*. Otherwise, it returns the sublist of *list* beginning with the first occurrence of *item*; that is, it returns the first cons of the list whose car is *item*. The comparison is made by **eq**. Because **memq** returns **nil** if it does not find anything, and something non-**nil** if it finds something, it is often used as a predicate. Examples:

```
(memq 'a '(1 2 3 4)) => nil
(memq 'a '(g (x a y) c a d e a f)) => (a d e a f)
```

Note that the value returned by **memq** is **eq** to the portion of the list beginning with **a**. Thus **rplaca** on the result of **memq** may be used, if you first check to make sure **memq** did not return **nil**. Example:

```
(let ((sublist (memq x z)))      ;search for x in the list z.
  (if (not (null sublist))       ;if it is found,
      (rplaca sublist y)))       ;replace it with y.
```

memq could have been defined by:

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

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

member *item list* *Function*

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)
        (t (member item (cdr list))) ))
```

mem *predicate item list* *Function*

mem is the same as **memq** except that it takes an extra argument that 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**. It can also be used with noncommutative predicates. The predicate is called with *item* as its first argument and the element of *list* as its second argument, so:

```
(mem #'< 4 list)
```

finds the first element in *list* for which **(< 4 *x*)** is true; that is, it finds the first element greater than **4**.

find-position-in-list *item list* *Function*

find-position-in-list looks down *list* for an element that is **eq** to *item*, like **memq**. However, it returns the numeric index in the list at which it found the first occurrence of *item*, or **nil** if it did not find it at all. This function is sort of the complement of **nth**; like **nth**, it is zero-based. See the function **nth**. 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* *Function*

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**.

tailp *sublist list* *Function*

Returns **t** if *sublist* is a sublist of *list* (that is, one of the conses that makes up *list*). Otherwise returns **nil**. Another way to look at this is that **tailp** returns **t** if **(nthcdr** *n list***)** is *sublist*, for some value of *n*. **tailp** could have been defined by:

```
(defun tailp (sublist list)
  (do ((list list (cdr list)))
      ((null list) nil)
    (if (eq sublist list)
        (return t))))
```

delq *item list* &optional *n* *Function*

(delq *item list***)** returns the *list* with all occurrences of *item* removed. **eq** is used for the comparison. The argument *list* is actually modified (**rplacd**ed) when instances of *item* are spliced out. **delq** should be used for value, not for effect. That is, use:

```
(setq a (delq 'b a))
```

rather than:

```
(delq 'b a)
```

These two are *not* equivalent when the first element of the value of **a** is **b**.

(delq *item list n***)** is like **(delq** *item list***)** except only the first *n* instances of *item* are deleted. *n* is allowed to be zero. If *n* is greater than or equal to the number of occurrences of *item* in the list, all occurrences of *item* in the list 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 -1))
  (cond ((or (atom list) (zerop n)) list)
        ((eq item (car list))
         (delq item (cdr list) (1- n)))
        (t (rplacd list (delq item (cdr list) n)))))
```

If the third argument (*n*) is not supplied, it defaults to **-1**, which is effectively infinity, since it can be decremented any number of times without reaching zero.

delete *item list* &optional *n* *Function*

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

del *predicate item list* &optional *n* *Function*

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

remq *item list* &optional *n* *Function*

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* *Function*

remove is the same as **remq** except that **equal** is used for the comparison instead of **eq**.

rem *predicate item list* &optional *n* *Function*

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

union &rest *lists* *Function*

Takes any number of lists that represent sets and creates and returns a new list that represents the union of all the sets it is given. **union** uses **eq** for its comparisons. You cannot change the function used for the comparison. **(union)** returns **nil**.

intersection &rest *lists* *Function*

Takes any number of lists that represent sets and creates and returns a new list that represents the intersection of all the sets it is given. **intersection** uses **eq** for its comparisons. You cannot change the function used for the comparison. **(intersection)** returns **nil**.

nunion &rest *lists* *Function*

Takes any number of lists that represent sets and returns a new list that represents the union of all the sets it is given, by destroying any of the lists passed as arguments and reusing the conses. **(nunion)** returns **nil**.

nintersection &rest *lists* *Function*

Takes any number of lists that represent sets and returns a new list that represents the intersection of all the sets it is given, by destroying any of the lists passed as arguments and reusing the conses. **(nintersection)** returns **nil**.

subset *predicate list* &rest *extra-lists* *Function*

predicate should be a function of one argument. A new list is made by applying *predicate* to all of the elements of *list* and removing the ones for which the predicate returns **nil**. One of this function's names (**rem-if-not**) means "remove if this condition is not true"; that is, it keeps the elements for which *predicate* is true. The other name (**subset**) refers to the function's action if *list* is considered to represent a mathematical set.

If *extra-lists* is present, each element of *extra-lists* (that is, each further argument to **subset**) is a list of objects to be passed to *predicate* as *predicate*'s second argument, third argument, and so on. The reason for this is that *predicate* might be a function of many arguments; *extra-lists* lets you control what values are passed as additional arguments to *predicate*. However, the list returned by **subset** is still a "subset" of those values that were passed as the *first* argument in the various calls to *predicate*.

rem-if-not *predicate list* &rest *extra-lists* *Function*

predicate should be a function of one argument. A new list is made by applying *predicate* to all of the elements of *list* and removing the ones for which the predicate returns **nil**. One of this function's names (**rem-if-not**) means "remove if this condition is not true"; that is, it keeps the elements for which *predicate* is true. The other name (**subset**) refers to the function's action if *list* is considered to represent a mathematical set.

If *extra-lists* is present, each element of *extra-lists* (that is, each further argument to **subset**) is a list of objects to be passed to *predicate* as *predicate*'s second argument, third argument, and so on. The reason for this is that *predicate* might be a function of many arguments; *extra-lists* lets you control what values are passed as additional arguments to *predicate*. However, the list returned by **subset** is still a "subset" of those values that were passed as the *first* argument in the various calls to *predicate*.

subset-not *predicate list* &rest *extra-lists* *Function*

predicate should be a function of one argument. A new list is made by applying *predicate* to all the elements of *list* and removing the ones for which the predicate returns non-**nil**. One of this function's names (**rem-if**) means "remove if this condition is true". The other name (**subset-not**) refers to the function's action if *list* is considered to represent a mathematical set. The meaning of *extra-lists* is the same as for **subset**.

rem-if *predicate list* &rest *extra-lists* *Function*

predicate should be a function of one argument. A new list is made by applying *predicate* to all the elements of *list* and removing the ones for which the predicate returns non-**nil**. One of this function's names (**rem-if**) means "remove if this condition is true". The other name (**subset-not**) refers to the function's action if *list* is considered to represent a mathematical set. The meaning of *extra-lists* is the same as for **subset**.

del-if *predicate list* *Function*

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

del-if-not *predicate list* *Function*

del-if-not is just like **rem-if-not** except that it modifies *list* rather than creating a new list.

every *list predicate* &optional *step-function* *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; **cddr** is a typical function to use here.

some *list predicate* &optional *step-function* *Function*

some returns a tail of *list* such that the car of the tail is the first element that the *predicate* returns non-**nil** when applied to, or **nil** if *predicate* returns **nil** for every element. If *step-function* is present, it replaces **cdr** as the function used to get to the next element of the list; **cddr** is a typical function to use here.

3.7 Association Lists

assq *item alist* *Function*

(**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)
```

You can **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))**. Since the cdr of **nil** is guaranteed to be **nil**, this yields **nil** if no pair is found (or if a pair is found whose cdr is **nil**.)

assq could have been defined by:

```
(defun assq (item list)
    (cond ((null list) nil)
          ((eq item (caar list)) (car list))
          ((assq item (cdr list))) ))
```

assoc *item alist* *Function*

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* *Function*

ass is the same as **assq** except that it takes an extra argument that should be a predicate of two arguments, which is used for the comparison instead of **eq**. **(ass 'eq a b)** is the same as **(assq a b)**. See the function **mem**. As with **mem**, you may use noncommutative predicates; the first argument to the predicate is *item* and the second is the key of the element of *alist*.

memass *predicate item alist* *Function*

memass searches *alist* just like **ass**, but returns the portion of the list beginning with the pair containing *item*, rather than the pair itself. **(car (memass** *x y z***)) = (ass** *x y z***)**. See the function **mem**. As with **mem**, you may use noncommutative predicates; the first argument to the predicate is *item* and the second is the key of the element of *alist*.

rassq *item alist* *Function*

rassq means "reverse assq". It is like **assq**, but it tries to find an element of *alist* whose *cdr* (not car) is *eq* to *item*. **rassq** could have been defined by:

```
(defun rassq (item in-list)
    (do l in-list (cdr l) (null l)
      (and (eq item (cdar l))
           (return (car l)))))
```

rassoc *item alist* *Function*

rassoc is to **rassq** as **assoc** is to **assq**. That is, it finds an element whose cdr is **equal** to *item*.

rass *predicate item alist* *Function*

rass is to **rassq** as **ass** is to **assq**. That is, it takes a predicate to be used instead of **eq**. See the function **mem**. As with **mem**, you may use

noncommutative predicates; the first argument to the predicate is *item* and the second is the cdr of the element of *alist*.

sassq *item alist function* *Function*

(sassq *item alist function***)** is like **(assq** *item alist***)** except that if *item* is not found in *alist*, instead of returning **nil**, **sassq** calls the function *function* with no arguments. **sassq** could have been defined by:

```
(defun sassq (item alist function)
    (or (assq item alist)
        (apply function nil)))
```

sassq and **sassoc** are of limited use. These are primarily leftovers from Lisp 1.5.

sassoc *item alist function* *Function*

(sassoc *item alist function***)** is like **(assoc** *item alist***)** except that if *item* is not found in *alist*, instead of returning **nil**, **sassoc** calls the function *function* with no arguments. **sassoc** could have been defined by:

```
(defun sassoc (item alist function)
    (or (assoc item alist)
        (apply function nil)))
```

pairlis *cars cdrs* *Function*

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))
```

3.8 Property Lists

Lisp has always had a kind of tabular data structure called a *property list* (plist for short). A property list contains zero or more entries; each entry associates from a keyword symbol (called the *indicator*) to a Lisp object (called the *value* or, sometimes, the *property*). There are no duplications among the indicators; a property list can only have one property at a time with a given name.

This is very similar to an association list. The difference is that a property list is an object with a unique identity; the operations for adding and removing property list entries are side-effecting operations that alter the property list rather than making a new one. An association list with no entries would be the empty list **()**, that is, the symbol **nil**. There is only one empty list, so all empty association lists are the same object. Each empty property list is a separate and distinct object.

The implementation of a property list is a memory cell containing a list with an even

number (possibly zero) of elements. Each pair of elements constitutes a *property*; the first of the pair is the indicator and the second is the value. The memory cell is there to give the property list a unique identity and to provide for side-effecting operations.

The term "property list" is sometimes incorrectly used to refer to the list of entries inside the property list, rather than the property list itself. This is regrettable and confusing.

How do we deal with "memory cells" in Lisp; that is, what kind of Lisp object is a property list? Rather than being a distinct primitive data type, a property list can exist in one of three forms:

1. A property list can be a cons whose cdr is the list of entries and whose car is not used and is therefore available to the user to store something.

2. The system associates a property list with every symbol. See the section "The Property List". A symbol can be used where a property list is expected; the property-list primitives will automatically find the symbol's property list and use it.

3. A property list can be a memory cell in the middle of some data structure, such as a list, an array, an instance, or a defstruct. An arbitrary memory cell of this kind is named by a locative. See the section "Locatives". Such locatives are typically created with the **locf** special form. See the macro **locf**.

Property lists of the first kind are called "disembodied" property lists because they are not associated with a symbol or other data structure. The way to create a disembodied property list is **(ncons nil)**, or **(ncons** *data***)** to store *data* in the car of the property list.

Here is an example of the list of entries inside the property list of a symbol named **b1** that is being used by a program that 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**. We say that "the value of **b1**'s **color** property is **blue**", or, informally, that "**b1**'s **color** property is **blue**." The program is probably representing the information that the block represented by **b1** is painted blue. Similarly, it is probably representing in the rest of the property list that block **b1** is on top of block **b6**, and that **b1** is associated with blocks **b2**, **b3**, and **b4**.

get *plist indicator* *Function*

get looks up *plist*'s *indicator* property. If it finds such a property, it returns the value; otherwise, it returns **nil**. If *plist* is a symbol, the symbol's associated property list is used. For example, if the property list of **foo** is **(baz 3)**, then:

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

getl *plist indicator-list* *Function*

getl is like **get**, except that the second argument is a list of indicators. **getl** searches down *plist* for any of the indicators in *indicator-list* until it finds a property whose indicator is one of the elements of *indicator-list*. If *plist* is a symbol, the symbol's associated property list is used. **getl** returns the portion of the list inside *plist* beginning with the first such property that it found. So the **car** of the returned list is 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**. For example, if the property list of **foo** were:

```
(bar (1 2 3) baz (3 2 1) color blue height six-two)
```

then:

```
(getl 'foo '(baz height))
  => (baz (3 2 1) color blue height six-two)
```

When more than one of the indicators in *indicator-list* is present in *plist*, which one **getl** returns depends on the order of the properties. This is the only thing that depends on that order. The order maintained by **putprop** and **defprop** is not defined (their behavior with respect to order is not guaranteed and may be changed without notice).

putprop *plist x indicator* *Function*

This gives *plist* an *indicator*-property of *x*. After this is done, **(get** *plist indicator***)** will return *x*. If *plist* is a symbol, the symbol's associated property list is used. Example:

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

defprop *symbol x indicator* *Special Form*

defprop is a form of **putprop** with "unevaluated arguments", which is sometimes more convenient for typing. Normally it does not make sense to use a property list rather than a symbol as the first (or *plist*) argument. Example:

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

is the same as:

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

remprop *plist indicator* *Function*

This removes *plist*'s *indicator* property, by splicing it out of the property list. It returns that portion of the list inside *plist* of which the former *indicator*-property was the **car**. **car** of what **remprop** returns is what **get** would have returned with the same arguments. If *plist* is a symbol, the symbol's associated property list is used. For example, if the property list of **foo** was:

```
(color blue height six-three near-to bar)
```

then:

```
(remprop 'foo 'height) => (six-three near-to bar)
```

and **foo**'s property list would be:

```
(color blue near-to bar)
```

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

There is a mixin flavor, called **si:property-list-mixin**, that provides messages that do things analogous to what the above functions do. [Currently, the above functions do not work on flavor instances, but this will be fixed.]

3.9 Hash Tables

A hash table is a Lisp object that works something like a property list. Each hash table has a set of *entries*, each of which associates a particular *key* with a particular *value*. The basic functions that deal with hash tables can create entries, delete entries, and find the value that is associated with a given key. Finding the value is very fast even if there are many entries, because hashing is used; this is an important advantage of hash tables over property lists. See the section "Hash Primitive".

A given hash table can only associate one *value* with a given *key*; if you try to add a second *value* it will replace the first.

Hash tables come in two kinds, the difference being whether the keys are compared using **eq** or using **equal**. The following discussion refers to the **eq** kind of hash table; the other kind is described later, and works analogously.

Hash tables of the first kind are created by instantiating an instance of the **si:eq-hash-table** flavor with the **make-instance** function, which takes various init options. New entries are added to hash tables by sending them a **:put-hash** message. To look up a key and find the associated value, the **:get-hash** message is used. To remove an entry, use **:rem-hash**. Here is a simple example.

```
(setq a (make-instance 'si:eq-hash-table))

(send a ':put-hash 'color 'brown)

(send a ':put-hash 'name 'fred)

(send a ':get-hash 'color) => brown

(send a ':get-hash 'name) => fred
```

In this example, the symbols **color** and **name** are being used as keys, and the symbols **brown** and **fred** are being used as the associated values. The hash table has two items in it, one of which associates from **color** to **brown**, and the other of which associates from **name** to **fred**.

Keys do not have to be symbols; they can be any Lisp object. Likewise values can be any Lisp object. The Lisp function **eq** is used to compare keys, rather than **equal**.

When a hash table is first created, it has a *size*, which is the maximum number of entries it can hold. Usually the actual capacity of the table is somewhat less, since the hashing is not perfectly collision-free. With the maximum possible bad luck, the capacity could be very much less, but this rarely happens. If so many entries are added that the capacity is exceeded, the hash table will automatically grow, and the entries will be *rehashed* (new hash values will be recomputed, and everything will be rearranged so that the fast hash lookup still works). This is transparent to the caller; it all happens automatically.

The **describe** function prints a variety of useful information when called with a hash table.

Hash tables are implemented as instances of flavors. The two flavors for the two kinds of hash tables are **si:eq-hash-table** and **si:equal-hash-table**. See the section "Hash Table Messages".

3.9.1 Creating Hash Tables

A new hash table using **eq** for comparisons of the key is created by making an instance of the **si:eq-hash-table** flavor. (See the function **make-instance**.) The function **make-hash-table** also will create an **eq** hash table, and takes the init options to pass on to **make-instance** as arguments.

Hash tables using **equal** for comparisons are created by making an instance of the **si:equal-hash-table** flavor, or by calling the **make-equal-hash-table** function.

si:eq-hash-table *Flavor*

This flavor is used to create a hash table using the **eq** function for comparison of the hash keys. It accepts the following init options:

:size Sets the initial size of the hash table in entries, as a fixnum. The default is 100 (decimal). The actual size is rounded up from the size you specify to the next size that is good for the hashing algorithm. An automatic rehash of the hash table might occur before this many entries are stored in the table depending upon the keys being stored.

:area Specifies the area in which the hash table should be created. This is just like the **:area** option to **make-array**. See the function **make-array**. The default is **working-storage-area**.

:growth-factor Specifies how much to increase the size of the hash table when it becomes full. This is a flonum that is the ratio of the new size to the old size. The default is 1.3, which causes the table to be made 30 percent bigger each time it has to grow.

:rehash-before-cold
Causes **disk-save** to rehash this hash table if its hashing has been invalidated. (This is part of the before-cold initializations.) Thus every user of the saved band does not have to waste the overhead of rehashing the first time they use the hash table after cold booting.

For **eq** hash tables, the hashing is invalidated whenever garbage collection or band compression occurs because the hash function is sensitive to addresses of objects, and those operations move objects to different addresses. For **equal** hash tables, the hash function is not sensitive to addresses of objects that **sxhash** knows how to hash but it is sensitive to addresses of other objects. The hash table remembers whether it contains any such objects.

Normally a hash table is automatically rehashed "on demand" the first time it is used after the hashing has become invalidated. This first **:get-hash** operation is therefore much slower than normal.

The **:rehash-before-cold** option should be used on hash tables that are a permanent part of the system, likely to be saved in a band saved by **disk-save**, and to be touched by users of that band. This applies both to hash tables in the Lisp system itself and to hash tables in user-written subsystems that are saved on disk bands.

si:equal-hash-table *Flavor*

A table of this flavor uses the **equal** function for comparison of the hash keys. It accepts the following init option as well as those described for **eq** hash tables. See the flavor **si:eq-hash-table**.

:rehash-threshold
Specifies how full the table can be before it must grow. This is typically a flonum. The default is 0.8, which represents 80 percent.

make-hash-table &rest *options* *Function*

This creates a new hash table using the **eq** function for comparison of the keys. This function just calls **make-instance** using the **si:eq-hash-table** flavor, passing *options* to **make-instance** as init options. See the flavor **si:eq-hash-table**.

make-equal-hash-table &rest *options* *Function*

This creates a new hash table using the **equal** function for comparison of the keys. This function just calls **make-instance** using the **si:equal-hash-table** flavor, passing *options* to **make-instance** as init options. See the flavor **si:equal-hash-table**.

3.9.2 Hash Table Messages

This section describes the messages that can be sent to any hash table instance.

:get-hash *key* *Message*

Find the entry in the hash table whose key is *key*, and return the associated value. If there is no such entry, return **nil**. Returns a second value, which is **t** if an entry was found or **nil** if there is no entry for *key* in this table.

:put-hash *key value* *Message*

Create an entry in the hash table associating *key* to *value*. If there is already an entry for *key* then replace the value of that entry with *value*. Returns *value*. The hash table automatically grows if necessary.

:rem-hash *key* *Message*

Remove any entry for *key* in the hash table. Returns **t** if there was an entry or **nil** if there was not.

:swap-hash *key value* *Message*

This does the same thing as **:put-hash**, but returns different values. If there was already an entry in the hash table whose key was *key*, then it returns the old associated value as its first returned value, and **t** as its second returned value. Otherwise it returns two values, **nil** and **nil**.

:map-hash *function* &rest *args* *Message*

For each entry in the hash table, call *function* on the key of the entry and the value of the entry. If *args* is supplied, they are passed along to *function* following the value of the entry argument.

:clear-hash *Message*

Remove all the entries from the hash table.

:modify-hash *key function* &rest *args* *Message*

This message combines the actions of **:get-hash** and **:put-hash**. It lets you both examine the value for a particular key and change it. It is more efficient because it does the hash lookup once instead of twice.

It finds *value*, the value associated with *key*, and *key-exists-p*, which indicates whether the key was in the table. It then calls *function* with *key*, *value*, *key-exists-p*, and *other-args*. If no value was associated with the key, then *value* is **nil** and *key-exists-p* is **nil**. It puts whatever value *function* returns into the hash table, associating it with *key*.

```
(send new-coms ':modify-hash k foo a b c) =>
(funcall foo k val key-exists-p a b c)
```

:size *Message*

Returns the number of entries in the hash table, whether empty or filled. This means the amount of storage allocated, not the number of hash associations currently stored.

:filled-elements *Message*

Returns the number of entries in the hash table that have an associated value.

3.9.3 Hash Table Functions

In addition to sending an instance of a hash table a message, the following functions can also be used to manipulate a hash table. Please note that these functions are considered obsolete and are only documented here for compatibility.

gethash *key hash-table* *Function*

Sends *hash-table* a **:get-hash** message with *key* as its argument. The values returned are the same as for the **:get-hash** message.

gethash-equal *key hash-table* *Function*

Sends *hash-table* a **:get-hash** message with *key* as its argument. The values returned are the same as for the **:get-hash** message.

puthash *key value hash-table* *Function*

Sends *hash-table* a **:put-hash** message with arguments of *key* and *value*. The values returned are the same as for the **:put-hash** message.

puthash-equal *key value hash-table* *Function*

Sends *hash-table* a **:put-hash** message with arguments of *key* and *value*. The values returned are the same as for the **:put-hash** message.

remhash *key hash-table* *Function*

Sends *hash-table* a **:rem-hash** message with an argument of *key*. The values returned are the same as for the **:rem-hash** message.

remhash-equal *key hash-table* *Function*

Sends *hash-table* a **:rem-hash** message with an argument of *key*. The values returned are the same as for the **:rem-hash** message.

swaphash *key value hash-table* *Function*

Sends *hash-table* a **:swap-hash** message with arguments of *key* and *value*. The values returned are the same as for the **:swap-hash** message.

swaphash-equal *key value hash-table* *Function*
Sends *hash-table* a **:swap-hash** message with arguments of *key* and *value*. The values returned are the same as for the **:swap-hash** message.

maphash *function hash-table* &rest *args* *Function*
Sends *hash-table* a **:map-hash** message with an argument of *function*, passing *args* to *function*.

maphash-equal *function hash-table* &rest *args* *Function*
Sends *hash-table* a **:map-hash** message with an argument of *function*, passing *args* to *function*.

clrhash *hash-table* *Function*
Sends *hash-table* a **:clear-hash** message. Returns the hash table itself.

clrhash-equal *hash-table* *Function*
Sends *hash-table* a **:clear-hash** message. Returns the hash table itself.

3.9.4 Dumping Hash Tables to Files

Instances of hash tables can be dumped to files by using any of the dump functions. See the function **sys:dump-forms-to-file**. The hash table flavors have the **:fasd-form** methods required to support dumping of their data to a fasd file.

3.9.5 Hash Tables and the Garbage Collector

The **eq** type hash tables actually hash on the address of the representation of the object. When the copying garbage collector changes the addresses of object, it lets the hash facility know so that **:get-hash** will rehash the table based on the new object addresses. **equal** hash tables also hash on the address, sometimes.

There will eventually be an init option to **si:eq-hash-table** that tells it to make a "non-GC-protecting" hash table. This is a special kind of hash table with the property that if one of its keys becomes "garbage", that is, an object not known about by anything other than the hash table, then the entry for that key will be silently removed from the table. When these exist they will be documented in this section.

3.9.6 Hash Primitive

Hashing is a technique used in algorithms to provide fast retrieval of data in large tables. A function, known as a "hash function", is created, which takes an object that might be used as a key, and produces a number associated with that key. This number, or some function of it, can be used to specify where in a table to look for the datum associated with the key. It is always possible for two different objects to "hash to the same value"; that is, for the hash function to return the same number

for two distinct objects. Good hash functions are designed to minimize this by evenly distributing their results over the range of possible numbers. However, hash table algorithms must still deal with this problem by providing a secondary search, sometimes known as a *rehash*. For more information, consult a textbook on computer algorithms.

si:equal-hash *object* *Function*

si:equal-hash computes a hash code of an object, and returns it as a fixnum. A property of **si:equal-hash** is that **(equal** *x y***)** always implies **(= (si:equal-hash** *x***) (si:equal-hash** *y***))**. The number returned by **si:equal-hash** is always a nonnegative fixnum, possibly a large one. **si:equal-hash** tries to compute its hash code in such a way that common permutations of an object, such as interchanging two elements of a list or changing one character in a string, will always change the hash code.

Here is an example of how to use **si:equal-hash** in maintaining hash tables of objects:

```
(defun knownp (x &aux i bkt)    ;look up ⧗ in the table
    (setq i (remainder (si:equal-hash x) 176))
      ;The remainder should be reasonably randomized.
    (setq bkt (aref table i))
      ;bkt is thus a list of all those expressions that
      ;hash into the same number as does x.
    (memq x bkt))
```

To write an "intern" for objects, one could:

```
(defun sintern (x &aux bkt i tem)
    (setq i (remainder (si:equal-hash x) 2n-1))
        ;2n-1 stands for a power of 2 minus one.
        ;This is a good choice to randomize the
        ;result of the remainder operation.
    (setq bkt (aref table i))
    (cond ((setq tem (memq x bkt))
           (car tem))
          (t (aset (cons x bkt) table i)
             x)))
```

si:equal-hash provides what is called "hashing on **equal**"; that is, two objects that are **equal** are considered to be "the same" by **si:equal-hash**. In particular, if two strings differ only in alphabetic case, **si:equal-hash** will return the same thing for both of them because they are **equal**. The value returned by **si:equal-hash** does not depend on the value of **alphabetic-case-affects-string-comparison**.

Therefore, **si:equal-hash** is useful for retrieving data when two keys that are not the same object but are **equal**, are considered the same. If you consider two such keys to be different, then you need "hashing on **eq**", where two different objects are always considered different. In some Lisp implementations, there is an easy way to

create a hash function that hashes on **eq**, namely, by returning the virtual address of the storage associated with the object. But in other implementations, including Zetalisp, this does not work, because the address associated with an object can be changed by the relocating garbage collector. The hash tables discussed here deal with this problem by using the appropriate subprimitives so that they interface correctly with the garbage collector. If you need a hash table that hashes on **eq**, it is already provided.

3.10 Sorting

Several functions are provided for sorting arrays and lists. These functions use algorithms that always terminate no matter what sorting predicate is used, provided only that the predicate always terminates. The main sorting functions are not *stable*; that is, equal items may not stay in their original order. If you want a stable sort, use the stable versions. But if you do not care about stability, do not use them, since stable algorithms are significantly slower.

After sorting, the argument (either list or array) has been rearranged internally 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, you must sort a copy of the argument, obtainable by **fillarray** or **copylist**, as appropriate. Furthermore, **sort** of a list is like **delq** in that it should not be used for effect; the result is conceptually the same as the argument but in fact is a different Lisp object.

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

The sorting package is smart about compact lists; it sorts compact sublists as if they were arrays. See the section "Cdr-coding". An explanation of compact lists is in that section.

sort *table predicate* *Function*

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 predicate should return **nil** if its arguments are equal. For example, to sort in the opposite direction from <, use >, not $\geq$. This is because the quicksort algorithm used to sort arrays and cdr-coded lists becomes very much slower when the predicate returns non-**nil** for equal elements while sorting many of them.

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. 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))
```

When **sort** is given a list, it may change the order of the conses of the list (using **rplacd**), and so it cannot be used merely for side effect; only the *returned value* of **sort** will be the sorted list. This will mess up the original list; if you need both the original list and the sorted list, you must copy the original and sort the copy. See the function **copylist**.

Sorting an array just moves the elements of the array into different places, and so sorting an array for side effect only is all right.

If the argument to **sort** is an array with a fill pointer, note that, like most functions, **sort** considers the active length of the array to be the length, and so only the active part of the array will be sorted. See the function **array-active-length**.

sortcar *x predicate* *Function*

sortcar is the same as **sort** except that the predicate is applied to the cars of the elements of *x*, instead of directly to the elements of *x*. Example:

```
(sortcar '((3 . dog) (1 . cat) (2 . bird)) #'<)
                    =>  ((1 . cat) (2 . bird) (3 . dog))
```

Remember that **sortcar**, when given a list, may change the order of the conses of the list (using **rplacd**), and so it cannot be used merely for side effect; only the *returned value* of **sortcar** will be the sorted list.

stable-sort *x predicate* *Function*

stable-sort is like **sort**, but if two elements of *x* are equal, that is, *predicate* returns **nil** when applied to them in either order, then those two elements will remain in their original order.

stable-sortcar *x predicate* *Function*

stable-sortcar is like **sortcar**, but if two elements of *x* are equal, that is, *predicate* returns **nil** when applied to their cars in either order, then those two elements will remain in their original order.

sort-grouped-array *array group-size predicate* *Function*

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* *Function*

This is like **sort-grouped-array** except that the *predicate* is applied to four arguments: an array, an index into that array, a second array, and an index into the second array. *predicate* should consider each index as the subscript of the first element of a record in the corresponding array, and compare the two records. This is more general than **sort-grouped-array** since the function can get at all of the elements of the relevant records, instead of only the first element.

4. Symbols

4.1 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.

Symbols are often used as special variables. See the section "Variables". The symbols **nil** and **t** are always bound to themselves; they may not be assigned, bound, or otherwise used as variables. Attempting to change the value of **nil** or **t** (usually) causes an error.

The functions described here work on *symbols*, not *variables* in general. This means that the functions below will not work if you try to use them on local variables.

set *symbol value* *Function*

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**.

set-globally works like **set** but sets the global value regardless of any bindings currently in effect. See the function **set**.

set-globally operates on the *global value* of a special variable; it bypasses any bindings of the variable in the current stack group. This function resides in the global package.

symeval *sym* *Function*

symeval is the basic primitive for retrieving a symbol'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.

symeval-globally works like **symeval** but returns the global value regardless of any bindings currently in effect. See the function **symeval**.

symeval-globally operates on the *global value* of a special variable; it bypasses any bindings of the variable in the current stack group. This function resides in the global package.

makunbound *sym* *Function*

makunbound causes *sym* to become unbound. Example:

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

makunbound returns its argument.

makunbound-globally works like **makunbound** but sets the global value regardless of any bindings currently in effect. See the function **makunbound**.

makunbound-globally operates on the *global value* of a special variable; it bypasses any bindings of the variable in the current stack group. This function resides in the global package.

boundp *sym* *Function*

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

variable-boundp *variable* *Special Form*

Returns **t** if the variable is bound and **nil** if the variable is not bound. *variable* should be any kind of variable (it is not evaluated): local, special, or instance. Note: local variables are always bound; if *variable* is local, the compiler issues a warning and replaces this form with **t**.

If **a** is a special variable, **(boundp 'a)** is the same as **(variable-boundp a)**.

variable-makunbound *variable* *Special Form*

Makes the variable be unbound and returns *variable*. *variable* should be any kind of variable (it is not evaluated): local, special, or instance. Note: since local variables are always bound, they cannot be made unbound; if *variable* is local, the compiler issues a warning.

If **a** is a special variable, **(makunbound 'a)** is the same as **(variable-makunbound a)**.

value-cell-location *sym* *Function*

value-cell-location returns a locative pointer to *sym*'s value cell. See the section "Locatives". It is preferable to write:

```
(locf (symeval sym))
```

instead of calling this function explicitly.

This is actually the internal value cell; there can also be an external value cell. See the section "Closures". Note: the function **value-cell-location**

works on symbols that get converted to local variables. See the section "Variables". The compiler knows about it specially when its argument is a quoted symbol which is the name of a local variable. It returns a pointer to the cell that holds the value of the local variable.

4.1.1 Special Forms for Dealing with Variables

value-cell-location on local variables is obsolete. In the past, the only way to generate a locative pointer to the memory cell associated with a local variable called **a** was with the form **(value-cell-location 'a)**. This is inelegant, since **value-cell-location** is a function that concerns symbols (special variables) in particular, rather than variables in general. See the section "Variables: Evaluation". This form continues to work, but the compiler issues a warning telling you that it is obsolete. A special form replaces it:

variable-location *variable* *Special Form*

Returns a locative pointer to the memory cell that holds the value of the variable. *variable* should be any kind of variable (it is not evaluated): local, special, or instance.

If **a** is a local or instance variable and you use the obsolete **(value-cell-location 'a)** form, the compiler issues a warning and converts it into the proper **variable-location** form. So if you have programs that use this form, they will continue to work. Similarly, the compiler issues warnings for obsolete uses of **boundp** and **makunbound**, and generates code that works.

(value-cell-location 'a) is still a good form when **a** is a special variable. It behaves slightly differently from the form **(variable-location a)**, in the case that **a** is a variable "closed over" by some closure. See the section "Closures". **value-cell-location** returns a locative pointer to the internal value cell of the symbol (the one that holds the invisible pointer, which is the real value cell of the symbol), whereas **variable-location** returns a locative pointer to the external value cell of the symbol (the one pointed to by the invisible pointer, which holds the actual value of the variable).

You can also use **locf** on variables (this has always been true). **(locf a)** now expands into **(variable-location a)**.

4.2 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 that 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 bound or assigned. (However, to bind a function cell you must use the **bind** subprimitive.) The following functions are analogous to the value-cell-related functions in the previous section.

fsymeval *sym* *Function*

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

fset *sym definition* *Function*

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

fboundp *sym* *Function*

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

fmakunbound *sym* *Function*

fmakunbound causes *sym* to be undefined, that is, its function cell to be empty. It returns *sym*.

function-cell-location *sym* *Function*

function-cell-location returns a locative pointer to *sym*'s function cell. See the section "Locatives". It is preferable to write:

```
(locf (fsymeval sym))
```

rather than calling this function explicitly.

Since functions are the basic building block of Lisp programs, the system provides a variety of facilities for dealing with functions. See the section "Functions".

4.3 The Property List

Every symbol has an associated property list. See the section "Property Lists". When a symbol is created, its property list is initially empty.

The Lisp language itself does not use a symbol's property list for anything. (This was not true in older Lisp implementations, where the print-name, value-cell, and function-cell of a symbol were kept on its property list.) However, various system programs use the property list to associate information with the symbol. For

instance, the editor uses the property list of a symbol that is the name of a function to remember where it has the source code for that function, and the compiler uses the property list of a symbol which is the name of a special form to remember how to compile that special form.

Because of the existence of print-name, value, function, and package cells, none of the Maclisp system property names (**expr**, **fexpr**, **macro**, **array**, **subr**, **lsubr**, **fsubr**, and in former times **value** and **pname**) exist in Zetalisp.

plist *sym* *Function*

This returns the list which represents the property list of *sym*. Note that this is not the property list itself; you cannot do **get** on it.

setplist *sym list* *Function*

This sets the list that represents the property list of *sym* to *list*. **setplist** is to be used with caution (or not at all), since property lists sometimes contain internal system properties, which are used by many useful system functions. Also, it is inadvisable to have the property lists of two different symbols be **eq**, since the shared list structure will cause unexpected effects on one symbol if **putprop** or **remprop** is done to the other.

property-cell-location *sym* *Function*

This returns a locative pointer to the location of *sym*'s property-list cell. This locative pointer is equally valid as *sym* itself, as a handle on *sym*'s property list.

4.4 The Print Name

Every symbol has an associated string called the *print-name*, or *pname* for short. This string is used as the external representation of the symbol: if the string is typed in to **read**, it is read as a reference to that symbol (if it is interned), and if the symbol is printed, **print** types out the print-name. More information about the *reader* and the *printer* can be found elsewhere: See the section "What the Reader Accepts". See the section "What the Printer Produces".

get-pname *sym* *Function*

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

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

samepnamep *sym1 sym2* *Function*

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 lowercase 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. **samepnamep** is useful for determining if two symbols would be the same except that they are in different packages. See the document *Packages*. 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**. **samepnamep** is provided mainly so that you can write programs that will work in Maclisp as well as Zetalisp; in new programs, you should just use **string-equal**.

4.5 The Package Cell

Every symbol has a *package cell* that is used, for interned symbols, to point to the package to which the symbol belongs. For an uninterned symbol, the package cell contains **nil**. See the document *Packages*. Information about package cells and packages, in general, is found there.

4.6 Creating Symbols

The functions in this section are primitives for creating symbols. However, before discussing them, it is important to point out that most symbols are created by a higher-level mechanism, namely the reader and the **intern** function. Nearly all symbols in Lisp are created by virtue of the reader's having seen a sequence of input characters that looked like the printed representation of a symbol. When the reader sees such a p.r., it calls **intern**, which looks up the sequence of characters in a big table and sees whether any symbol with this print-name already exists. If it does, **read** uses the existing symbol. If it does not exist, then **intern** creates a new symbol and puts it into the table, and **read** uses that new symbol. See the function **intern**.

A symbol that has been put into such a table is called an *interned* symbol. Interned symbols are normally created automatically; the first time someone (such as the reader) asks for a symbol with a given print-name, that symbol is automatically created.

These tables are called *packages*. In Zetalisp, interned symbols are the province of the *package* system. Although interned symbols are the most commonly used, they will not be discussed further here. See the document *Packages*.

An *uninterned* symbol is a symbol used simply as a data object, with no special cataloging. An uninterned symbol prints the same as an interned symbol with the same print-name, but cannot be read back in.

The following functions can be used to create uninterned symbols explicitly.

make-symbol *pname* &optional *permanent-p* *Function*

This creates a new uninterned symbol whose print-name is the string *pname*. The value and function bindings are unbound and the property list is empty. If *permanent-p* is specified, it is assumed that the symbol is going to be interned and probably kept around forever; in this case it and its pname will be put in the proper areas. If *permanent-p* is **nil** (the default), the symbol goes in the default area and the pname is not copied. *permanent-p* is mostly for the use of **intern** itself.

Examples:

```
(make-symbol "FOO") => FOO
(make-symbol "Foo") => |Foo|
```

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

If a symbol has lowercase characters in its print-name, the printer will quote the name using slashes or vertical bars. The vertical bars inhibit the Lisp reader's normal action, which is to convert a symbol to uppercase upon reading it. See the section "What the Printer Produces".

Example:

```
(setq a (make-symbol "Hello"))    ; => |Hello|
(princ a)                         ; will print out Hello
```

copysymbol *sym copy-props* *Function*

This returns a new uninterned symbol with the same print-name as *sym*. If *copy-props* is non-**nil**, then the 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-props* is **nil**, then the new symbol will be unbound and undefined, and its property list will be empty.

gensym &optional *x* *Function*

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**), for example, "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 symbol's 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 32.) => 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 that 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".

5. Numbers

Zetalisp 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*. Maclisp contains both generic numeric functions (like **plus**) and specific numeric functions (like +), which only operate on a certain type, and are much more efficient. In Zetalisp, this distinction does not exist; both function names exist for compatibility but they are identical. The microprogrammed structure of the machine makes it possible to have only the generic functions without loss of efficiency.

The types of numbers in Zetalisp are:

fixnum	Fixnums are 24-bit 2's complement binary integers. These are the "preferred, most efficient" type of number.
bignum	Bignums are arbitrary-precision binary integers.
flonum	(LM-2 only) 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	(LM-2 only) 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. Stable rounding is employed. Small flonums are useful because, like fixnums, but unlike flonums, they do not require any storage. Computing with small flonums is more efficient than with regular flonums because the operations are faster and consing overhead is eliminated.
single-float	(3600 only) Single-precision floating-point numbers have a precision of 24 bits, or about 7 decimal digits. Their range is from 1.1754944e-38 to 3.4028235e38.
double-float	(3600 only) Double-precision floating-point numbers have a precision of 53 bits, or about 16 decimal digits. Their range is from 2.2250738585072014d-308 to 1.7976931348623157d308.

Generally, Lisp objects have a unique identity; each exists, independent of any other, and you can use the **eq** predicate to determine whether two references are to the same object or not. Numbers are the exception to this rule; they do not work this way. The following function may return either **t** or **nil**. Its behavior is considered undefined, but as this manual is written it returns **t** when interpreted but **nil** when compiled.

```
(defun foo ()
  (let ((x (float 5)))
    (eq x (car (cons x nil)))))
```

This is very strange from the point of view of Lisp's usual object semantics, but the implementation works this way to gain efficiency, and on the grounds that identity testing of numbers is not really an interesting thing to do. So, the rule is that the result of applying **eq** to numbers is undefined, and may return either **t** or **nil** at will. If you want to compare the values of two numbers, use =.

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. **eq** works as well as = as an equality test for fixnums. Small-flonums that are = tend to be **eq** also, but it is unwise to depend on this.

The distinction between fixnums and bignums is largely transparent to the user. You simply compute with integers, and the system represents some as fixnums and the rest (less efficiently) as bignums. cally converts back and forth between fixnums and bignums based solely on the size of the integer. There are a few "low level" functions that only work on fixnums; this fact is noted in their documentation. Also when using **eq** on numbers you should be aware of the fixnum/bignum distinction.

Integer computations cannot "overflow", except for division by zero, since bignums can be of arbitrary size. Floating-point computations can get exponent overflow or underflow, if the result is too large or small to be represented. Exponent overflow always signals an error. Exponent underflow normally signals an error, and assumes **0.0** as the answer if you say to proceed from the error. However, if the value of the variable **zunderflow** is non-**nil**, the error is skipped and computation proceeds with **0.0** in place of the result that was too small.

zunderflow *Variable*

If the value of **zunderflow** is non-**nil**, any floating-point computation that results in a floating-point underflow will have zero as its result. If the value of **zunderflow** is **nil**, any such computation 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 an integer 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. When a single-precision floating-point number meets a double-precision floating-point number, the result is a double-float.

Thus, if the constants in a numerical algorithm are written as small-flonums (assuming this provides adequate precision), and if the input is a small-flonum, the computation will be done in small-flonum mode and the result will a small-flonum, while if the input is a large-flonum the computations will be done in full precision and the result will be a flonum.

Zetalisp never automatically converts between flonums and small-flonums, in the way it automatically converts between fixnums and bignums, since this would lead either to inefficiency or to unexpected numerical inaccuracies. (When a small-flonum meets a flonum, the result is a flonum, but if you use only one type, all the results will be of the same type, too.) This means that a small-flonum computation can get an exponent overflow error even when the result could have been represented as a large-flonum.

Floating-point numbers retain only a certain number of bits of precision; therefore, the results of computations are only approximate. Large-flonums have 31 bits and small-flonums have 17 bits, not counting the sign. The method of approximation is "stable rounding". The result of an arithmetic operation will be the flonum that is closest to the exact value. If the exact result falls precisely halfway between two flonums, the result will be rounded down if the least-significant bit is 0, or up if the least-significant bit is 1. This choice is arbitrary but insures that no systematic bias is introduced.

The 3600 supports IEEE-standard single-precision and double-precision floating-point numbers. Number objects exist that are outside the upper and lower limits of the ranges for single and double precision. Larger than the largest number is +1e= (or +1d= for doubles). Smaller than the smallest number is -1e= (or -1d= for doubles). Smaller than the smallest normalized positive number but larger than zero are the "denormalized" numbers. Some floating-point objects are Not-a-Number (NaN); they are the result of (// 0.0 0.0) (with trapping disabled) and like operations.

IEEE numbers are symmetric about zero, so the negative of every representable number is also a representable number (on the 3600 only). Zeros are signed in IEEE format, but +0.0 and -0.0 act the same arithmetically. For example:

```
(= +0.0 -0.0) => t
(plusp 0.0)   => nil
(plusp -0.0)  => nil
(zerop -0.0)  => t
(eq 0.0 -0.0) => nil
```

See the IEEE standard: Microprocessor Standards Committee, IEEE Computer Society, "A Proposed Standard for Binary Floating-Point Arithmetic: Draft 8.0 of IEEE Task P754," *Computer*, March 1981, pp. 51-62.

Integer addition, subtraction, and multiplication always produce an exact result. Integer division, on the other hand, returns an integer rather than the exact rational-number result. The quotient is truncated towards zero rather than rounded. The exact rule is that if *A* is divided by *B*, yielding a quotient of *C* and a remainder of *D*, then $A = B * C + D$ exactly. *D* is either zero or the same sign as *A*. Thus the absolute value of *C* is less than or equal to the true quotient of the absolute values of *A* and *B*. This is compatible with Maclisp and most computer hardware. However, it has the serious problem that it does *not* obey the rule that if *A* divided by *B* yields a quotient of *C* and a remainder of *D*, then dividing $A + k * B$

by B will yield a quotient of $C + k$ and a remainder of D for all integer k. The lack of this property sometimes makes regular integer division hard to use. New functions that implement a different kind of division, that obeys this rule, will be implemented in the future.

Unlike Maclisp, Zetalisp does not have number declarations in the compiler. Note that because fixnums and small-flonums require no associated storage they are as efficient as declared numbers in Maclisp. Bignums and (large) flonums are less efficient; however, bignum and flonum intermediate results are garbage-collected in a special way that avoids the overhead of the full garbage collector.

The different types of numbers can be distinguished by their printed representations. A leading or embedded (but *not* trailing) decimal point, and/or an exponent separated by "e", indicates a flonum on the LM-2 or a single-precision floating-point number on the 3600. If a number has an exponent separated by "s", it is a small-flonum. If a number has an exponent separated by "d", it is a double-precision floating-point number. Small-flonums require a special indicator so that new users will not accidentally compute with the lesser precision. Fixnums and bignums have similar printed representations since there is no numerical value that has a choice of whether to be a fixnum or a bignum; an integer is a bignum if and only if its magnitude too big for a fixnum. See the section "What the Reader Accepts".

5.1 Numeric Predicates

zerop *x* *Function*

Returns **t** if *x* is zero. Otherwise it returns **nil**. If *x* is not a number, **zerop** causes an error. For flonums, this only returns **t** for exactly **0.0** or **0.0s0**; there is no "fuzz".

plusp *x* *Function*

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* *Function*

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* *Function*

Returns **t** if *number* is odd, otherwise **nil**. If *number* is not a fixnum or a bignum, **oddp** causes an error.

evenp *number* *Function*

Returns **t** if *number* is even, otherwise **nil**. If *number* is not a fixnum or a bignum, **evenp** causes an error.

signp *test x* *Special Form*

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** returns **t** if *x* is a number that satisfies the *test*, **nil** if it is not a number or does not meet the test. *test* is not evaluated, but *x* is. *test* can be one of the following:

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

Examples:

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

See the function **fixp**. See the function **floatp**. See the function **bigp**. See the function **small-floatp**. See the function **sys:single-float-p**. See the function **sys:double-float-p**. See the function **numberp**.

5.2 Numeric Comparisons

All of these functions require that their arguments be numbers, and signal an error if given a nonnumber. They work on all types of numbers, automatically performing any required coercions (as opposed to Maclisp, in which generally only the spelled-out names work for all kinds of numbers).

= *x y* *Function*

Returns **t** if *x* and *y* are numerically equal. An integer can be = to a flonum.

greaterp *number [&rest] more-numbers* *Function*

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
```

Related topics:

The following function is a synonym of **greaterp**.
See the function >.

> *number* &rest *more-numbers* *Function*

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
```

Related topics:

The following function is a synonym of >.
See the function **greaterp**.

>= *number* &rest *more-numbers* *Function*

≥ compares its arguments from left to right. If any argument is less than the next, ≥ returns **nil**. But if the arguments are monotonically decreasing or equal, the result is **t**.

Related topics:

The following function is a synonym of >=.
See the function ≥.

≥ *number* &rest *more-numbers* *Function*

≥ compares its arguments from left to right. If any argument is less than the next, ≥ returns **nil**. But if the arguments are monotonically decreasing or equal, the result is **t**.

Related topics:

The following function is a synonym of ≥.
See the function >=.

lessp *number [&rest] more-numbers* *Function*

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
```

Related topics:

The following function is a synonym of **lessp**.
See the function <.

< *number* &rest *more-numbers* *Function*

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
```

Related topics:

The following function is a synonym of <.
See the function **lessp**.

<= *number* &rest *more-numbers* *Function*

≤ compares its arguments from left to right. If any argument is greater than the next, ≤ returns **nil**. But if the arguments are monotonically increasing or equal, the result is **t**.

Related topics:

The following function is a synonym of <=.
See the function ≤.

≤ *number* &rest *more-numbers* *Function*

≤ compares its arguments from left to right. If any argument is greater than the next, ≤ returns **nil**. But if the arguments are monotonically increasing or equal, the result is **t**.

Related topics:

The following function is a synonym of ≤.
See the function <=.

≠ *x y* *Function*

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

max &rest *args* *Function*

max returns the largest of its arguments. Example:

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

max requires at least one argument.

min &rest *args* *Function*

min returns the smallest of its arguments. Example:

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

min requires at least one argument.

5.3 Arithmetic

All of these functions require that their arguments be numbers, and signal an error if given a nonnumber. They work on all types of numbers, automatically performing any required coercions (as opposed to Maclisp, in which generally only the spelled-out versions work for all kinds of numbers, and the "$" versions are needed for flonums).

plus &rest *args* *Function*

Returns the sum of its arguments. If there are no arguments, it returns **0**, which is the identity for this operation.

Related topics:

The following functions are synonyms of **plus**.
See the function **+**.
See the function **+$**.

+ &rest *args* *Function*

Returns the sum of its arguments. If there are no arguments, it returns **0**, which is the identity for this operation.

Related topics:

The following functions are synonyms of **+**.
See the function **plus**.
See the function **+$**.

+$ &rest *args* *Function*

Returns the sum of its arguments. If there are no arguments, it returns **0**, which is the identity for this operation.

Related topics:

The following functions are synonyms of **+$**.
See the function **plus**.
See the function **+**.

difference *arg* &rest *args* *Function*

Returns its first argument minus all of the rest of its arguments.

minus *x* *Function*

Returns the negative of *x*. Examples:

```
(minus 1) => -1
(minus -3.0) => 3.0
```

- *arg* &rest *args* *Function*

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.

Related topics:

The following function is a synonym of -.
See the function **-$**.

-$ *arg* &rest *args* *Function*

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.

Related topics:

The following function is a synonym of **-$**.
See the function -.

abs *x* *Function*

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)))
```

times &rest *args* *Function*

Returns the product of its arguments. If there are no arguments, it returns 1, which is the identity for this operation.

Related topics:

The following functions are synonyms of **times**.
See the function *.
See the function **$**.

* &rest *args* *Function*

Returns the product of its arguments. If there are no arguments, it returns 1, which is the identity for this operation.

Related topics:

The following functions are synonyms of *.
See the function **times**.
See the function **$**.

***$** &rest *args* *Function*

Returns the product of its arguments. If there are no arguments, it returns 1, which is the identity for this operation.

Related topics:

The following functions are synonyms of **$**.
See the function **times**.
See the function *.

quotient *arg* &rest *args* *Function*

Returns the first argument divided by all of the rest of its arguments.

// arg &rest *args* *Function*

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*). The exact rules for the meaning of the quotient and remainder of two integers are given in another section. See the section "Numbers". This explains why the rules used for integer division are not correct for all applications. Examples:

```
(// 3 2) => 1          ;Fixnum division truncates.
(// 3 -2) => -1
(// -3 2) => -1
(// -3 -2) => 1
(// 3 2.0) => 1.5
(// 3 2.0s0) => 1.5s0
(// 4 2) => 2
(// 12. 2. 3.) => 2
(// 4.0) => .25
```

Related topics:

The following function is a synonym of //.
See the function //$.

//$ arg &rest *args* *Function*

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*). The exact rules for the meaning of the quotient and remainder of two integers are given in another section. See the section "Numbers". This explains why the rules used for integer division are not correct for all applications. Examples:

```
(// 3 2) => 1          ;Fixnum division truncates.
(// 3 -2) => -1
(// -3 2) => -1
(// -3 -2) => 1
(// 3 2.0) => 1.5
(// 3 2.0s0) => 1.5s0
(// 4 2) => 2
(// 12. 2. 3.) => 2
(// 4.0) => .25
```

Related topics:

The following function is a synonym of **//$**.
See the function **//**.

remainder *x y* *Function*

Returns the remainder of *x* divided by *y*. *x* and *y* must be integers (fixnums or bignums). The exact rules for the meaning of the quotient and remainder of two integers are given in another section. See the section "Numbers".

```
(\ 3 2) => 1
(\ -3 2) => -1
(\ 3 -2) => 1
(\ -3 -2) => -1
```

Related topics:

The following function is a synonym for **remainder**.
See the function ****.

**** *x y* *Function*

Returns the remainder of *x* divided by *y*. *x* and *y* must be integers (fixnums or bignums). The exact rules for the meaning of the quotient and remainder of two integers are given in another section. See the section "Numbers".

```
(\ 3 2) => 1
(\ -3 2) => -1
(\ 3 -2) => 1
(\ -3 -2) => -1
```

Related topics:

The following function is a synonym for ****.
See the function **remainder**.

mod *x y* *Function*

The same as **remainder**, except that the returned value has the sign of the *second* argument instead of the first. When there is no remainder, the returned value is **0**.

Examples:

```
(mod -3 2)  => 1
(mod 3 -2)  => -1
(mod -3 -2) => -1
(mod 4 -2)  => 0
```

add1 *x* *Function*

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

Related topics:

The following functions are synonyms of **add1**.
See the function **1+**.
See the function **1+$**.

1+ *x* *Function*

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

Related topics:

The following functions are synonyms of **1+**.
See the function **add1**.
See the function **1+$**.

1+$ *x* *Function*

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

The following functions are synonyms of **1+$**.
See the function **add1**.
See the function **1+**.

sub1 *x* *Function*

(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.

Related topics:

The following functions are synonyms of **sub1**.
See the function **1-**.
See the function **1-$**.

1- *x* *Function*

(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.

Related topics:

The following functions are synonyms of **1-**.
See the function **sub1**.
See the function **1-$**.

1-$ *x* *Function*

(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.
Related topics:

The following functions are synonyms of **1-$**.
See the function **sub1**.
See the function **1-**.

gcd *x y* &rest *args* *Function*

Returns the greatest common divisor of all its arguments. The arguments must be integers (fixnums or bignums).

Related topics:

The following function is a synonym of **gcd**.
See the function \\.

\\ *x y* &rest *args* *Function*

Returns the greatest common divisor of all its arguments. The arguments must be integers (fixnums or bignums).

Related topics:

The following function is a synonym of \\.
See the function **gcd**.

expt *x y* *Function*

Returns *x* raised to the *y*th power. The result is an integer if both arguments are integers (even if *y* is negative!) and floating-point if either *x* or *y* or both is floating-point. If the exponent is an integer a repeated-squaring algorithm is used, while if the exponent is floating the result is **(exp (* *y* (log *x*)))**.

Related topics:

The following functions are synonyms of **expt**.
See the function ^.
See the function ^**$**.

^ *x y* *Function*

Returns *x* raised to the *y*th power. The result is an integer if both arguments are integers (even if *y* is negative!) and floating-point if either *x* or *y* or both is floating-point. If the exponent is an integer a repeated-squaring algorithm is used, while if the exponent is floating the result is **(exp (* *y* (log *x*)))**.

Related topics:

The following functions are synonyms of ^.
See the function **expt**.
See the function ^**$**.

^**$** *x y* *Function*

Returns *x* raised to the *y*th power. The result is an integer if both arguments are integers (even if *y* is negative!) and floating-point if either *x* or *y* or both is floating-point. If the exponent is an integer a repeated-squaring algorithm is used, while if the exponent is floating the result is **(exp (* *y* (log *x*)))**.

Related topics:

The following functions are synonyms of ^**$**.
See the function **expt**.
See the function ^.

sqrt *x* *Function*

Returns the square root of *x*.

isqrt *x* *Function*

Integer square root. *x* must be an integer; the result is the greatest integer less than or equal to the exact square root of *x*.

signum *value* *Function*

signum is a function for determining the sign of its argument.

```
(signum -2.5)  => -1.0
(signum 3.9)  => 1.0
(signum 0)  => 0
(signum 59)  => 1
```

The definition is compatible with the current Common Lisp design.

***dif** *x y* *Function*

(LM-2 only) This is one of the internal microcoded arithmetic functions. There is no reason why anyone should need to write code with this explicitly, since the compiler knows how to generate the appropriate code for **plus**, **+**, and so on. This name is only here for Maclisp compatibility.

***plus** *x y* *Function*

(LM-2 only) This is one of the internal microcoded arithmetic functions. There is no reason why anyone should need to write code with this explicitly, since the compiler knows how to generate the appropriate code for **plus**, **+**, and so on. This name is only here for Maclisp compatibility.

***quo** *x y* *Function*

(LM-2 only) This is one of the internal microcoded arithmetic functions. There is no reason why anyone should need to write code with this explicitly, since the compiler knows how to generate the appropriate code for **plus**, **+**, and so on. This name is only here for Maclisp compatibility.

***times** *x y* *Function*

(LM-2 only) This is one of the internal microcoded arithmetic functions. There is no reason why anyone should need to write code with this explicitly, since the compiler knows how to generate the appropriate code for **plus**, **+**, and so on. This name is only here for Maclisp compatibility.

5.4 Transcendental Functions

These functions are only for floating-point arguments; if given an integer they will convert it to a flonum. If given a small-flonum, they will return a small-flonum.

exp *x* *Function*

Returns *e* raised to the *x*th power, where *e* is the base of natural logarithms.

log *x* *Function*

Returns the natural logarithm of *x*.

sin *x* *Function*

Returns the sine of *x*, where *x* is expressed in radians.

sind *x* *Function*

Returns the sine of *x*, where *x* is expressed in degrees.

cos *x* *Function*

Returns the cosine of *x*, where *x* is expressed in radians.

cosd *x* *Function*

Returns the cosine of *x*, where *x* is expressed in degrees.

atan *y x* *Function*

Returns the angle, in radians, whose tangent is *y*/*x*. **atan** always returns a nonnegative number between zero and 2π.

atan2 *y x* *Function*

Returns the angle, in radians, whose tangent is *y*/*x*. **atan2** always returns a number between $-\pi$ and π.

5.5 Numeric Type Conversions

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

fix *x* *Function*

Converts *x* from a flonum (or small-flonum) to an integer, truncating towards negative infinity. The result is a fixnum or a bignum as appropriate. If *x* is already a fixnum or a bignum, it is returned unchanged.

fixr *x* *Function*

Converts *x* from a flonum (or small-flonum) to an integer, rounding to the nearest integer. If *x* is exactly halfway between two integers, this rounds up (towards positive infinity). **fixr** could have been defined by:

```
(defun fixr (x)
  (if (fixp x) x (fix (+ x 0.5))))
```

float *x* *Function*

Converts any kind of number to a flonum on the LM-2 and to a single-precision floating-point number on the 3600. Note that, on the 3600, **float** reduces a double-precision argument to single precision.

small-float *x* *Function*

(LM-2 only) Converts any kind of number to a small-flonum.

dfloat *x* *Function*

(3600 only) Converts any kind of number to a double-precision floating-point number.

5.6 Logical Operations on Numbers

Except for **lsh** and **rot**, these functions operate on both fixnums and bignums. **lsh** and **rot** have an inherent word-length limitation and hence only operate on 24-bit fixnums. Negative numbers are operated on in their 2's-complement representation.

logior *number* &rest *more-numbers* *Function*

Returns the bit-wise logical *inclusive or* of its arguments. At least one argument is required. Example:

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

logxor *number* &rest *more-numbers* *Function*

Returns the bit-wise logical *exclusive or* of its arguments. At least one argument is required. Example:

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

logand *number* &rest *more-numbers* *Function*

Returns the bit-wise logical *and* of its arguments. At least one argument is required. Examples:

```
(logand 3456 707) => 406
(logand 3456 -100) => 3400
```

lognot *number* *Function*

Returns the logical complement of *number*. This is the same as **logxor**ing *number* with -1. Example:

```
(lognot 3456) => -3457
```

boole *fn* &rest *numbers* *Function*

boole is the generalization of **logand, logior**, and **logxor**. *fn* should be a fixnum between 0 and 17 octal inclusive; it controls the function that is computed. If the binary representation of *fn* is *abcd* (*a* is the most significant bit, *d* the least) then the truth table for the Boolean operation is as follows:

```
      y
    | 0  1
---------
  0| a  c
x  |
  1| b  d
```

If **boole** has more than three arguments, it is associated left to right; thus,

```
(boole fn x y z) = (boole fn (boole fn x y) z)
```

With two arguments, the result of **boole** is simply its second argument. At least two arguments are required.

Examples:

```
(boole 1 x y) = (logand x y)
(boole 6 x y) = (logxor x y)
(boole 2 x y) = (logand (lognot x) y)
```

logand, **logior**, and **logxor** are usually preferred over the equivalent forms of **boole**, to avoid putting magic numbers in the program.

bit-test *x y* *Function*

bit-test is a predicate that 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)))
```

lsh *x y* *Function*

Returns *x* shifted left *y* bits if *y* is positive or zero, or *x* shifted right |*y*| bits if *y* is negative. Zero bits are shifted in (at either end) to fill unused positions. *x* and *y* must be fixnums. (In some applications you may find **ash** useful for shifting bignums.) Examples:

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

ash *x y* *Function*

Shifts *x* arithmetically left *y* bits if *y* is positive, or right -*y* bits if *y* is negative. Unused positions are filled by zeroes from the right, and by copies of the sign bit from the left. Thus, unlike **lsh**, the sign of the result is always the same as the sign of *x*. If *x* is a fixnum or a bignum, this is a shifting operation. If *x* is a flonum, this does scaling (multiplication by a power of two), rather than actually shifting any bits.

rot *x y* *Function*

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. (There is no function for rotating bignums.) Examples:

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

haulong *x* *Function*

This returns the number of significant bits in |x|. x may be a fixnum or a bignum. Its sign is ignored. The result is the least integer strictly greater than the base-2 logarithm of |x|. Examples:

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

haipart *x n* *Function*

Returns the high n bits of the binary representation of |x|, or the low $-n$ bits if n is negative. x may be a fixnum or a bignum; its sign is ignored. **haipart** could have been defined by:

```
(defun haipart (x n)
  (setq x (abs x))
  (if (minusp n)
      (logand x (1- (ash 1 (- n))))
      (ash x (min (- n (haulong x))
                  0))))
```

5.7 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 (that is, 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 27 (octal), since a byte must fit in a fixnum although bytes can be loaded from and deposited into bignums. (Bytes are always positive numbers.) The format of byte-specifiers is taken from the PDP-10 byte instructions.

ldb *ppss num* *Function*

ppss specifies a byte of *num* to be extracted. 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". *num* may be a fixnum or a bignum. The returned value is always a fixnum. Example:

```
(ldb 0306 4567) => 56
```

load-byte *num position size* *Function*

This is like **ldb** except that instead of using a byte specifier, the *position* and *size* are passed as separate arguments. The argument order is not analogous to that of **ldb** so that **load-byte** can be compatible with Maclisp.

ldb-test *ppss y* *Function*

ldb-test is a predicate that 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 nonzero. **ldb-test** is implemented as a macro which expands as follows:

```
(ldb-test ppss y) ==> (not (zerop (ldb ppss y)))
```

mask-field *ppss num* *Function*

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**. *num* must be a fixnum. Example:

```
(mask-field 0306 4567) => 560
```

dpb *byte ppss num* *Function*

Returns a number that 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**. *num* may be a fixnum or a bignum. The name means "deposit byte". Example:

```
(dpb 23 0306 4567) => 4237
```

deposit-byte *num position size byte* *Function*

This is like **dpb** except that instead of using a byte specifier, the *position* and *size* are passed as separate arguments. The argument order is not analogous to that of **dpb** so that **deposit-byte** can be compatible with Maclisp.

deposit-field *byte ppss num* *Function*

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

```
(deposit-field 230 0306 4567) => 4237
```

byte *size position* *Function*

Creates a byte specifier for a byte *size* bits wide, *position* bits from the right-hand (least-significant) end of the word.

Example:

```
(ldb (byte 3 4) #o12345) => 6
```

byte-size *byte-specifier* *Function*

Extracts the size field of *byte-specifier*. You can use **setf** on this form:

```
(setq a (byte 3 4))
(setf (byte-size a) 2)
(byte-size a) => 2
```

byte-position *byte-specifier* *Function*

Extracts the position field of *byte-specifier*. You can use **setf** on this form:

```
(setq a (byte 3 4))
(setf (byte-position a) 2)
(byte-position a) => 2
```

The behavior of the following two functions depends on the size of fixnums, and so functions using them may not work the same way on future implementations of Zetalisp. Their names start with "%" because they are more like machine-level subprimitives than the previous functions.

%logldb *ppss fixnum* *Function*

%logldb is like **ldb** except that it only loads out of fixnums and allows a byte size of 30 (octal), that is, all 24. bits of the fixnum including the sign bit.

%logdpb *byte ppss fixnum* *Function*

%logdpb is like **dpb** except that it only deposits into fixnums. Using this to change the sign-bit will leave the result as a fixnum, while **dpb** would produce a bignum result for arithmetic correctness. **%logdpb** is good for manipulating fixnum bit-masks such as are used in some internal system tables and data structures.

5.8 Random Numbers

The functions in this section provide a pseudorandom number generator facility. The basic function you use is **random**, which returns a new pseudorandom number each time it is called. Between calls, its state is saved in a data object called a *random-array*. Usually there is only one random-array; however, if you want to create a reproducible series of pseudorandom numbers, and be able to reset the state to control when the series starts over, then you need some of the other functions here.

random &optional *arg random-array* *Function*

(random) returns a random fixnum, positive or negative. If *arg* is present, a fixnum between 0 and *arg* minus 1 inclusive is returned. If *random-array* is present, the given array is used instead of the default one. Otherwise, the default random-array is used (and is created if it does not already exist). The algorithm is executed inside a **without-interrupts** so two processes can use the same random-array without colliding. See the special form **without-interrupts**.

A random-array consists of an array of numbers, and two pointers into the array. The pointers circulate around the array; each time a random number is requested, both pointers are advanced by one, wrapping around at the end of the array. Thus, the distance forward from the first pointer to the second pointer, allowing for wraparound, stays the same. Let the length of the array be *length* and the distance between the pointers be *offset*. To generate a new random number, each pointer is set to its old value plus one, modulo *length*. Then the two elements of the array addressed by the pointers are added together; the sum is stored back into the array at the location where the second pointer points, and is returned as the random number after being normalized into the right range.

This algorithm produces well-distributed random numbers if *length* and *offset* are chosen carefully, so that the polynomial *x^length+x^offset+1* is irreducible over the mod-2 integers. The system uses 71. and 35.

The contents of the array of numbers should be initialized to anything moderately random, to make the algorithm work. The contents get initialized by a simple random number generator, based on a number called the *seed*. The initial value of the seed is set when the random-array is created, and it can be changed. To have several different controllable resettable sources of random numbers, you can create your own random-arrays. If you don't care about reproducibility of sequences, just use **random** without the *random-array* argument.

si:random-create-array *length offset seed* &optional *(area* **nil***)* *Function*

Creates, initializes, and returns a random-array. *length* is the length of the array. *offset* is the distance between the pointers and should be an integer less than *length*. *seed* is the initial value of the seed, and should be a fixnum. This calls **si:random-initialize** on the random array before returning it.

si:random-initialize *array* &optional *new-seed* *Function*

array must be a random-array, such as is created by **si:random-create-array**. If *new-seed* is provided, it should be a fixnum, and the seed is set to it. **si:random-initialize** reinitializes the contents of the array from the seed (calling **random** changes the contents of the array and the pointers, but not the seed).

5.9 24-bit Numbers

Sometimes it is desirable to have a form of arithmetic that has no overflow checking (which would produce bignums), and truncates results to the word size of the machine. In Zetalisp, 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 pseudorandom number generation.

%24-bit-plus *x y* *Function*
(LM-2 only) Returns the sum of *x* and *y* modulo 2^24. Both arguments must be fixnums.

%24-bit-difference *x y* *Function*
(LM-2 only) Returns the difference of *x* and *y* modulo 2^24. Both arguments must be fixnums.

%24-bit-times *x y* *Function*
(LM-2 only) Returns the product of *x* and *y* modulo 2^24. Both arguments must be fixnums.

5.10 Double-precision Arithmetic

These peculiar functions are useful in programs that do not want to use bignums for one reason or another. They should usually be avoided, as they are difficult to use and understand, and they depend on special numbers of bits and on the use of two's-complement notation.

%multiply-fractions *num1 num2* *Function*
(LM-2 only) 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 two's-complement fractions, $-1 \leq num < 1$, **%multiply-fractions** returns 1/2 of their correct product as a fraction.

%divide-double *dividend[24:46] dividend[0:23] divisor* *Function*
(LM-2 only) 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 will not fit in single precision.

%remainder-double *dividend[24:46] dividend[0:23] divisor* *Function*

(LM-2 only) 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* *Function*

(LM-2 only) *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**.

6. Locatives

6.1 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 that can hold a (pointer to a) Lisp object. For example, a symbol has five cells: the print name cell, the value cell, the function cell, the property list cell, and the package cell. The value cell holds (a pointer to) the binding of the symbol, and so on. Also, an array leader of length *n* has *n* cells, and an **art-q** array of *n* elements has *n* cells. (Numeric arrays do not have cells in this sense.) A locative is an object that points to a cell; it lets you refer to a cell, so that you can examine or alter its contents.

There are a set of functions that create locatives to cells; the functions are documented with the kind of object to which they create a pointer. See the function **ap-1**. See the function **ap-leader**. See the function **car-location**. See the function **value-cell-location**. The macro **locf** can be used to convert a form that accesses a cell to one that creates a locative pointer to that cell.

For example:

```
(locf (fsymeval x)) ==> (function-cell-location x)
```

locf is very convenient because it saves the writer and reader of a program from having to remember the names of all the functions that create locatives.

6.2 Functions That Operate on Locatives

Either of the functions **car** and **cdr** may be given a locative, and will return the contents of the cell at which the locative points. See the section "Conses".

For example:

```
(car (value-cell-location x))
```

is the same as:

```
(symeval x)
```

When using **locf** to return a locative, you should use **cdr** rather than **car** to access the contents of the cell to which the locative points. This is because **(locf (cdr** *list*)) returns the list itself instead of a locative.

Similarly, either of the functions **rplaca** and **rplacd** may be used to store an object into the cell at which a locative points.

For example:

```
(rplaca (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, the following 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 simplified-version-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 nonspecial compiled variables are not represented as symbols. However, the compiler arranges for it to work anyway, by recognizing **value-cell-location** of the name of a local variable, and compiling it as something other than a call to the **value-cell-location** function.

location-makunbound and **location-boundp** are versions of **makunbound** and **boundp** that can be used on any cell in the Lisp Machine. They take a locative pointer to designate the cell rather than a symbol. (**makunbound** is restricted to use with symbols.) The following two calls are equivalent:

```
(location-boundp (locf a))
(variable-boundp a)
```

The following two calls are also equivalent. When **a** is a special variable, they are the same as the two calls in the preceding example too.

```
(location-boundp (value-cell-location 'a))
(boundp 'a)
```

location-makunbound *loc* &optional *variable-name* *Function*

location-makunbound has been changed to take a symbol as an optional second argument: *variable-name* of the location that is being made unbound. Previously, it used to take one required argument.

location-makunbound uses *variable-name* to label the null pointer it stores so that the Debugger knows the name of the unbound location if it is

referenced. This is particularly appropriate when the location being made unbound is really a variable value cell of one sort or another, for example, closure or instance.

7. 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*. See the section "What Streams Are".

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 usually be abbreviated as "p.r.".

7.1 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 unslashified version is nicer looking in general, but if you give it to **read** it will not 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 (for example, a parenthesis appearing in the name of a symbol) are preceded 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 symbol followed by a colon).

For a fixnum or a bignum: if the number 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 **si:princ-function** property, the value of the property will be applied to two arguments: **minus** of the number to be printed, and the stream to which to print it (this is a hook to allow output in Roman numerals and the like); otherwise the value of **base** is invalid and an error is signalled Finally, if **base** equals 10. and the variable ***nopoint** is **nil**, a decimal point is printed out. Slashification does not affect the printing of numbers.

base *Variable*

The value of **base** is a number that is the radix in which fixnums are printed, or a symbol with a **si: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 flonum: the printer first decides whether to use ordinary notation or exponential notation. If the magnitude of the number is too large or too small, such that the ordinary notation would require an unreasonable number of leading or trailing zeroes, then exponential notation will be used. The number is printed as an optional leading minus sign, one or more digits, a decimal point, one or more digits, and an optional trailing exponent, consisting of the letter "e", an optional minus sign, and the power of ten. The number of digits printed is the "correct" number; no information present in the flonum is lost, and no extra trailing digits are printed that do not represent information in the flonum. Feeding the p.r. of a flonum back to the reader is always supposed to produce an equal flonum. Flonums are always printed in decimal; they are not affected by slashification nor by **base** and ***nopoint**.

For a small-flonum: the printed representation is very similar to that of a flonum, except that exponential notation is always used and the exponent is delimited by "s" rather than "e".

Ratios print in the current **ibase**, not always in decimal.

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 document *Packages*. The package name prefix is explained there. Secondly, if the p.r. would not read in as a symbol at all (that is, if the print-name looks like a number, or contains special characters), then 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. See the variable **readtable**.

Uninterned symbols are printed preceded by #:. You can turn this off by evaluating **(setf (si:pttbl-uninterned-prefix readtable) "")**.

For Common Lisp, character objects always print as #*char*.

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 that need to be preceded by slashes will be. Normally these are just double-quote and slash. Compatibly with Maclisp, carriage return is *not* ignored inside strings and vertical bars.

For an instance or an entity: if the object has a method for the **:print-self** message, that message is sent with three arguments: the stream to print to, the current *depth* of list structure, and whether slashification is enabled. The object should print a suitable p.r. on the stream. See the document *Objects, Message Passing, and Flavors*. Instances are documented there. Most such objects print like "any other data type" below, except with additional information such as a name. Some objects print only their name when slashification is not in effect (when **princ**ed).

For an array that is a named structure: if the array has a named structure symbol with a **named-structure-invoke** property that is the name of a function, then that function is called on five arguments: the symbol **:print-self**, the object itself, the stream to print to, the current *depth* of list structure, and whether slashification is enabled. A suitable printed representation should be sent to the stream. This allows you to define your own p.r. for his named structures. See the section "Named Structures". If the named structure symbol does not have a **named-structure-invoke** property, the printed-representation is like that for random data types: a number sign and a less-than sign ("<"), the named structure symbol, the numerical address of the array, and a greater-than sign (">").

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.

This is how the usual printed representations such as **(a b (foo bar) c)** are produced.

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 that is more concise and suppresses detail. Of course, neither

the ellipsis nor the "**" can be interpreted by **read**, since the relevant information is lost.

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 data type, a space, and the octal 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.

Including the machine address in the p.r. makes it possible to tell two objects of this kind apart without explicitly calling **eq** on them. This can be very useful during debugging. It is important to know that if garbage collection is turned on, objects will occasionally be moved, and therefore their octal machine addresses will be changed. It is best to shut off garbage collection temporarily when depending on these numbers.

None of the p.r.'s beginning with a number sign can be read back in, nor, in general, can anything produced by instances, entities, and named structures. See the section "What the Reader Accepts". This can be a problem if, for example, you are printing a structure into a file with the intent of reading it in later. The following feature allows you to make sure that what you are printing may indeed be read with the reader.

si:print-readably *Variable*

When **si:print-readably** is bound to **t**, the printer will signal an error if there is an attempt to print an object that cannot be interpreted by **read**. When the printer sends a **:print-self** or a **:print** message, it assumes that this error checking is done for it. Thus it is possible for these messages *not* to signal an error, if they see fit.

sys:printing-random-object *(object stream . keywords)* &body *body* *Macro*

The vast majority of objects that define **:print-self** messages have much in common. This macro is provided for convenience, so that users do not have to write out that repetitious code. It is also the preferred interface to **si:print-readably**. With no keywords, **si:printing-random-object** checks

the value of **si:print-readably** and signals an error if it is not **nil**. It then prints a number sign and a less-than sign, evaluates the forms in *body*, then prints a space, the octal machine address of the object, and a greater-than sign. A typical use of this macro might look like:

```
(si:printing-random-object (ship stream)
  (princ (typep ship) stream)
  (tyo #\space stream)
  (prin1 (ship-name ship) stream))
```

This might print **#<ship "ralph" 23655126>**.

The following keywords may be used to modify the behavior of **si:printing-random-object**:

:no-pointer This suppresses printing of the octal address of the object.

:typep This prints the result of **(typep** *object***)** after the less-than sign. In the example above, this option could have been used instead of the first two forms in the body.

If you want to control the printed representation of some object, usually the right way to do it is to make the object an array that is a named structure, or an instance of a flavor. See the section "Named Structures". See the document *Objects, Message Passing, and Flavors*. However, occasionally it is desirable to get control over all printing of objects, in order to change, in some way, how they are printed. If you need to do this, the best way to proceed is to customize the behavior of **si:print-object**, which is the main internal function of the printer. See the function **si:print-object**. All of the printing functions, such as **print** and **princ**, as well as **format**, go through this function. The way to customize it is by using the "advice" facility. See the special form **advise**.

7.2 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, and so on cannot be read in. However, it has many features that are not seen in the printer at all, such as more flexibility, comments, and convenient abbreviations for frequently used unwieldy constructs.

This section shows what kind of p.r.'s the reader understands, and explains the readtable, reader macros, and various features provided by **read**.

In general, the reader operates by recognizing tokens in the input stream. Tokens can be self-delimiting or can be separated by delimiters such as whitespace. A token

is the p.r. of an atomic object such as a symbol or a number, or a special character such as a parenthesis. The reader reads one or more tokens until the complete p.r. of an object has been seen, and then constructs and returns that object.

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 preceded 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 that is the radix in which fixnums are read. The initial value of **ibase** is 8.

read also understands a simple fixnum, followed by an underscore (_) or a circumflex (^), followed by another simple fixnum. The two simple fixnums are 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. (The second simple fixnum is not allowed to have a leading minus sign.) 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
```

The syntax for bignums is identical to the syntax for fixnums. A number is a bignum rather than a fixnum if and only if it is too large to be represented as a fixnum. Here are some examples of valid representations of bignums:

```
72361356126536125376512375126535123712635
-123456789.
105_1000
105_1000.
```

The syntax for a flonum is an optional plus or minus sign, optionally some digits, a decimal point, and one or more digits. By default, such a flonum or a simple fixnum, followed by an "e" (or "E") and a simple fixnum, is also a flonum; the fixnum after the "e" is the exponent of 10 by which the number is to be scaled. (The exponent is not allowed to have a trailing decimal point.) See the variable **cl:*read-default-float-format*** for ways of changing this default. If the exponent

is introduced by "s" (or "S") rather than "e", the number is a small-flonum. Here are some examples of printed-representations that read as flonums:

```
0.0
1.5
14.0
0.01
.707
-.3
+3.14159
6.03e23
1E-9
1.e3
```

Here are some examples of printed-representations that read as small-flonums:

```
0s0
1.5s9
-42S3
1.s5
```

The reader accepts all Common Lisp floating-point exponent characters.

Floating-point Exponent Characters

Following is a summary of floating-point exponent characters and the way numbers containing them are read on the 3600 and LM-2.

Character	*3600*	*LM-2*
B or b	single-precision	flonum
D or d	double-precision	flonum
E or e	depends on value of **cl:*read-default-float-format***	depends on value of **cl:*read-default-float-format***
F or f	single-precision	flonum
L or l	double-precision	flonum
S or s	single-precision	small-flonum

The variable **cl:*read-default-float-format*** controls how floating-point numbers with no exponent or an exponent or an exponent preceded by "E" or "e" are read.

cl:*read-default-float-format* *Variable*

Controls how floating-point numbers with no exponent or an exponent preceded by "E" or "e" are read. Following is a summary of the way possible values cause these numbers to be read on the 3600 and LM-2:

Value	*3600*	*LM-2*
cl:single-float	single-precision	flonum
cl:double-float	double-precision	flonum
cl:short-float	single-precision	small-flonum
cl:long-float	double-precision	flonum

The default value is **cl:single-float**.

Two integers separated by \ are read as a ratio of the integers. Ratios are read in the current **ibase**, not in decimal.

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. Alphabetic case is ignored in symbols; lowercase letters are translated to uppercase. When the reader sees the p.r. of a symbol, it *interns* it on a *package*. See the document *Packages*. 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 lowercase letters, parentheses, or reader macro characters) inside the name of a symbol, put a slash before each strange character. 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|
```

When a token could be read as either a symbol or an integer in a base larger than ten, the reader's action is determined by the value of **si:*read-extended-ibase-unsigned-number*** and **si:*read-extended-ibase-signed-number***.

si:*read-extended-ibase-unsigned-number* *Variable*

Controls how a token that could be a number or a symbol, and does not start with a + or - sign, is interpreted when **ibase** is greater than ten.

nil	It is never a number.
t	It is always a number.
:sharpsign	It is a symbol at top level, but a number after **#X** or **#*n*R**.

:single It is a symbol except immediately after **#X** or **#*n*R**.

The default value is **:single**.

si:*read-extended-ibase-signed-number* *Variable*

Controls how a token that could be a number or a symbol, and starts with a + or - sign, is interpreted when **ibase** is greater than ten.

nil It is never a number.

t It is always a number.

:sharpsign It is a symbol at top level, but a number after **#X** or **#*n*R**.

:single It is a symbol except immediately after **#X** or **#*n*R**.

The default value is **:sharpsign**.

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, precede 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 interned in the package rather than creating new symbols itself. Symbols are the only thing that work this way.

The dot that separates the two elements of a dotted-pair p.r. for a cons is only recognized if it is surrounded by delimiters (typically spaces). Thus dot may be freely used within print-names of symbols and within numbers. This is not compatible with Maclisp; in Maclisp **(a.b)** reads as a cons of symbols **a** and **b**, whereas in Zetalisp it reads as a list of a symbol **a.b**.

Tokens that consist of more than one dot, but no other characters, are legal symbols

in Zetalisp but errors in Common Lisp. For Common Lisp, the variable **si:*read-multi-dot-tokens-as-symbols*** should be set to **nil**.

si:*read-multi-dot-tokens-as-symbols* *Variable*

When **t**, for Zetalisp, tokens containing more than one dot, but no other characters, are read as symbols. When **nil**, for Common Lisp, tokens containing more than one dot but no other characters signal an error when read. Default: **t**.

If the circle-X (⊗) character is encountered, it is an octal escape, which may be useful for including weird characters in the input. The next three characters are read and interpreted as an octal number, and the character whose code is that number replaces the circle-X and the digits in the input stream. This character is always taken to be an alphabetic character, just as if it had been preceded by a slash.

7.3 Macro Characters

Certain characters are defined to be macro characters. When the reader sees one of these, it calls a function associated with the character. This function reads whatever syntax it likes and returns the object represented by that syntax. Macro characters are always token delimiters; however, they are not recognized when quoted by slash or vertical bar, nor when inside a string. Macro characters are a syntax-extension mechanism available to the user. Lisp comes with several predefined macro characters:

Quote (') is an abbreviation to make it easier to put constants in programs. *'foo* reads the same as **(quote** *foo*).

Semicolon (;) is used to enter comments. The semicolon and everything up through the next carriage return are ignored. Thus a comment can be put at the end of any line without affecting the reader.

Backquote (`) makes it easier to write programs to construct lists and trees by using a template. See the section "Backquote".

Comma (,) is part of the syntax of backquote and is invalid if used other than inside the body of a backquote. See the section "Backquote".

Sharp sign (#) introduces a number of other syntax extensions. See the section "Sharp-sign Abbreviations". Unlike the preceding characters, sharp sign is not a delimiter. A sharp sign in the middle of a symbol is an ordinary character.

The function **set-syntax-macro-char** can be used to define your own macro characters.

Reader macros that call a read function should call **si:read-recursive**.

si:read-recursive *stream* *Function*

si:read-recursive should be called by reader macros that need to call a function to read. It is important to call this function instead of **read** in macros that are written in Zetalisp but used by the Common Lisp readtable. In particular, this function must be called by macros used in conjunction with the Common Lisp *#n=* and *#n#* syntaxes.

stream is the stream from which to read. This function may be called only from inside a **read**.

For example, this is the reader macro called when the reader sees a quote ('):

```
si:(defun xr-quote-macro (list-so-far stream)
     list-so-far                                    ;not used
     (values (list-in-area read-area 'quote (read-recursive stream))
             'list))
```

7.4 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. Certain abbreviations allow a decimal number or certain special "modifier" characters between the sharp sign and the second character. Here are the currently defined sharp-sign constructs; more are likely to be added in the future.

#\ or #/

#\x (or *#/x*, which is identical) reads in as the number that 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 Zwei.

As in strings, upper- and lowercase letters are distinguished after #\. Any character works after #\, even those that are normally special to **read**, such as parentheses.

#\name (or *#/name*) reads in as the number which is the character code for the nonprinting character symbolized by *name*. A large number of character names are recognized. See the section "Special Character Names". For example, **#\return** reads in as a fixnum, being the character code for the Return character in the Lisp Machine character set. In general, the names that are written on the keyboard keys are accepted. The abbreviations **cr** for **return** and **sp** for **space** are accepted and generally preferred, since these characters are used so frequently. The page separator character is called **page**, although **form** and **clear-screen** are also accepted since the keyboard has one of those legends on the page key. The rules for reading

name are the same as those for symbols; thus upper- and lowercase letters are not distinguished, and the name must be terminated by a delimiter such as a space, a carriage return, or a parenthesis.

When the system types out the name of a special character, it uses the same table as the #\ reader; therefore, any character name typed out is acceptable as input.

#\ (or #/) can also be used to read in the names of characters that have control and meta bits set. The syntax looks like **#\control-meta-b** to get a "B" character with the control and meta bits set. You can use any of the prefix bit names **control**, **meta**, **hyper**, and **super**. They may be in any order, and upper- and lowercase letters are not distinguished. The last hyphen may be followed by a single character, or by any of the special character names normally recognized by #\. If it is a single character, it is treated the same way the reader normally treats characters in symbols; if you want to use a lowercase character or a special character such as a parenthesis, you must precede it with a slash character. Examples: **#\Hyper-Super-A**, **\meta-hyper-roman-i**, **#\CTRL-META-/(**.

The character can also be modified with control and meta bits by inserting one or more special characters between the # and the \. This syntax is obsolete since it is not mnemonic and it generally unclear. However, it is used in some old programs, so here is how it is defined. #α\x generates Control-*x*. #β\x generates Meta-*x*. #π\x generates Super-*x*. #λ\x generates Hyper-*x*. These can be combined, for instance #πβ\& generates Super-Meta-ampersand. Also, #ε\x is an abbreviation for #αβ\x. When control bits are specified, and *x* is a lowercase alphabetic character, the character code for the uppercase version of the character is produced.

In Common Lisp, #*char* (or #/*char*) can cause *char* to read as a character object instead of an integer, depending on the readtable.

#^ #^*x* is exactly like #α/*x* if the input is being read by Zetalisp; it generates Control-*x*. In Maclisp *x* is converted to uppercase and then exclusive-or'ed with 100 (octal). Thus #^*x* always generates the character returned by **tyi** if the user holds down the control key and types *x*. (In Maclisp #α/*x* sets the bit set by the Control key when the TTY is open in **fixnum** mode.)

#' #'*foo* is an abbreviation for **(function** *foo***)**. *foo* is the p.r. of any object. This abbreviation can be remembered by analogy with the ' macro-character, since the **function** and **quote** special forms are somewhat analogous.

#, #,*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 that cannot be written with **quote**. Note that the reader does not put **quote** around the result of the evaluation. You must do this yourself if you want it, typically by using the ' macro-character. An example of a case where you do not want **quote** around it is when this object is an element of a constant list.

#. **#.*foo*** evaluates *foo* (the p.r. of a lisp form) at read time, regardless of who is doing the reading.

#: **#:*name*** reads *name* as an uninterned symbol. It always creates a new symbol. Like all package prefixes, **#:** can be followed by any expression. Example: **#:(a b c)**.

#B **#B*rational*** reads *rational* (an integer or a ratio) in binary (radix 2). Examples:

```
#B1101 <=> 13.
#B1100\100 <=> 3
```

#O **#O** *number* reads *number* in octal regardless of the setting of **ibase**. Actually, any expression can be prefixed by **#O**; it will be read with **ibase** bound to 8.

#X **#X** *number* reads *number* in radix 16. (hexadecimal) regardless of the setting of *ibase*. As with **#O**, any expression can be prefixed by **#X**. The *number* can contain embedded hexadecimal "digits" A through F as well as the 0 through 9.

#R **#*radix*R** *number* reads *number* in radix *radix* regardless of the setting of **ibase**. As with **#O**, any expression can be prefixed by **#*radix*R**; it will be read with **ibase** bound to *radix*. *radix* must consist of only digits, and it is read in decimal. *number* can consist of both numeric and alphabetic digits, depending upon *radix*.

For example, **#3R102** is another way of writing **11.** and **#11R32** is another way of writing **35.**

#Q **#Q** *foo* reads as *foo* if the input is being read by Zetalisp, otherwise it reads as nothing (whitespace).

#M **#M** *foo* reads as *foo* if the input is being read into Maclisp, otherwise it reads as nothing (whitespace).

#N **#N** *foo* reads as *foo* if the input is being read into NIL or compiled to run in NIL, otherwise it reads as nothing (whitespace). Also, during the reading of *foo*, the reader temporarily defines various NIL-compatible sharp-sign abbreviations (such as **#!** and **#"**) in order to parse the form correctly, even though its not going to be evaluated.

#+ This abbreviation provides a read-time conditionalization facility similar to, but more general than, that provided by **#M**, **#N**, and **#Q**. It is used as **#+***feature form*. If *feature* is a symbol, then this is read as *form* if **(status feature** *feature*) is **t**. If **(status feature** *feature*) is **nil**, then this is read as whitespace. Alternately, *feature* may be a boolean expression composed of **and**, **or**, and **not** operators and symbols representing items which may appear on the **(status features)** list. **(or lispm amber)** represents evaluation of the predicate **(or (status feature lispm) (status feature amber))** in the read-time environment.

For example, **#+lispm** *form* makes *form* exist if being read by Zetalisp, and is thus equivalent to **#Q** *form*. Similarly, **#+maclisp** *form* is equivalent to **#M** *form*. **#+(or lispm nil)** *form* will make *form* exist on either Zetalisp or in NIL. Note that items may be added to the **(status features)** list by means of **(sstatus feature** *feature*), thus allowing the user to selectively interpret or compile pieces of code by parameterizing this list. See the special form **sstatus**.

#- *#-feature form* is equivalent to **#+(not** *feature*) *form*.

#| #| begins a comment for the Lisp reader. The reader ignores everything until the next |#, which closes the comment. Note that if the |# is inside a comment that begins with a semicolon, it is *not* ignored; it closes the comment that began with the preceding #|. #| and |# can be on different lines, and #|...|# pairs can be nested.

#< This is not valid reader syntax. It is used in the p.r. of objects that cannot be read back in. Attempting to read a #< will cause an error.

#◇ #◇ turns infix expression syntax into regular Lisp code. It is intended for people who like to use traditional arithmetic expressions in Lisp code. It is not intended to be extensible or to be a full programming language. We do not intend to extend it into one.

```
(defun my-add (a b)
  #◇a+b◇)
```

The quoting character is backslash. It is necessary for including special symbols (such as -) in variable names.

! reads one Lisp expression, which can use this reader-macro inside itself.

#◇ supports the following syntax:

Delimiters — Begin the reader macro with #◇, complete it with ◇.

```
#◇a+b-c◇
```

Escape characters — Special characters in symbol names must be preceded with backslash (\). You can escape to normal Lisp in an infix expression; precede the Lisp form with exclamation point (!).

Symbols — Start symbols with a letter. They may contain digits and underscore characters. Any other characters need to be quoted with \.

Operators — It accepts the following classes of operators. Arithmetic operator precedence is like that in FORTRAN and PL/I.

Operator	*Infix*	*Lisp Equivalent*
Assignment	x : y	(setf x y)
Functions	f(x,y)	(f x y) -- also

		works for defstruct accessors, and so on.
Array ref	a[i,j]	(aref a i j)
Unary ops	+ - not	*same*
Binary ops	+ - * / ^ = ≠ < ≤ > ≥ and or	*same*
Conditional	if p then c	(if p c)
	if p then c else a	(if p c a)
Grouping:	(a, b, c)	(progn a b c) -- even works for (1+2)/3

The following example shows matrix multiplication using an infix expression.

```
(defun matrix-multiply (a b)
  (let ((n (array-dimension-n 2 a)))
    (unless (= n (array-dimension-n 1 b))
      (ferror "Matrices ~S and ~S do not have compatible dimensions") a b)
    (let ((d1 (array-dimension-n 1 a))
          (d2 (array-dimension-n 2 b)))
      (let ((c #◊ make\-array(list(d1, d2), !':type, art\-float)◊ ))
        (dotimes (i d1)
          (dotimes (j d2)
            #◊ c[i,j] : !(loop for k below n sum #◊ a[i,k]*b[k,j] ◊)◊))
        c))))
```

The line containing the infix expression could also have been written like this:

```
(let ((sum 0))
  (dotimes (k n) #◊ sum:sum+a[i,k]*b[k,j] ◊)
  #◊ c[i,j]:sum ◊)
```

The function **set-syntax-#-macro-char** can be used to define your own sharp sign abbreviations.

7.5 Special Character Names

The following are the recognized special character names, in alphabetical order except with synonyms together and linked with equal signs. These names can be used after a #\ to get the character code for that character. Most of these characters type out as this name enclosed in a lozenge. First we list the special function keys.

```
abort           break           call            clear-input=clear
delete=vt       end             hand-down       hand-left
hand-right      hand-up         help            hold-output
```

```
roman-i          roman-ii          roman-iii         roman-iv
line=lf          macro=back-next   network
overstrike=backspace=bs            page=clear-screen=form
quote            resume            return=cr         rubout
space=sp         status            stop-output       system
tab              terminal=esc
```

These are printing characters that also have special names because they can be hard to type on a PDP-10.

```
altmode      circle-plus   delta        gamma
integral     lambda        plus-minus   up-arrow
```

The following are special characters sometimes used to represent single and double mouse clicks. The buttons can be called either **l, m, r** or **1, 2, 3** depending on stylistic preference. These characters all contain the **%%kbd-mouse** bit.

mouse-L-1=mouse-1-1
mouse-M-1=mouse-2-1
mouse-R-1=mouse-3-1

mouse-L-2=mouse-1-2
mouse-M-2=mouse-2-2
mouse-R-2=mouse-3-2

7.6 The Readtable

A data structure called the *readtable* that 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 you can change the meanings of characters to alter and customize the syntax of characters. It is also possible to have several readtables describing different syntaxes and to switch from one to another by binding the symbol **readtable**.

readtable *Variable*

The value of **readtable** is the current readtable. This starts out as a copy of **si:initial-readtable**. You can bind this variable to temporarily change the readtable being used.

si:initial-readtable *Variable*

The value of **si:initial-readtable** is the initial standard readtable. You should never change the contents of either this readtable or **si:initial-readtable**; only examine it, by using it as the *from-readtable* argument to **copy-readtable** or **set-syntax-from-char**. Change **readtable** instead.

You can program the reader by changing the readtable in any of three ways. The syntax of a character can be set to one of several predefined possibilities. A character can be made into a *macro character*, whose interpretation is controlled by a user-supplied function that is called when the character is read. You can create a

completely new readtable, using the readtable compiler (**sys: io; rtc**) to define new kinds of syntax and to assign syntax classes to characters. Use of the readtable compiler is not documented here.

copy-readtable &optional *from-readtable to-readtable* *Function*

from-readtable, which defaults to the current readtable, is copied. If *to-readtable* is unsupplied or **nil**, a fresh copy is made. Otherwise *to-readtable* is clobbered with the copy. Use **copy-readtable** to get a private readtable before using the following functions to change the syntax of characters in it. The value of **readtable** at the start of a Lisp Machine session is the initial standard readtable, which usually should not be modified.

set-syntax-from-char *to-char from-char* &optional *to-readtable from-readtable* *Function*

Makes the syntax of *to-char* in *to-readtable* be the same as the syntax of *from-char* in *from-readtable*. *to-readtable* defaults to the current readtable, and *from-readtable* defaults to the initial standard readtable.

set-character-translation *from-char to-char* &optional *readtable* *Function*

Changes *readtable* so that *from-char* will be translated to *to-char* upon read-in, when *readtable* is the current readtable. This is normally used only for translating lowercase letters to uppercase. Character translations are turned off by slash, string quotes, and vertical bars. *readtable* defaults to the current readtable.

set-syntax-macro-char *char function* &optional *readtable* *Function*

Changes *readtable* so that *char* is a macro character. When *char* is read, *function* is called. *readtable* defaults to the current readtable.

function is called with two arguments: *list-so-far* and the input stream. When a list is being read, *list-so-far* is that list (**nil** if this is the first element). At the "top level" of **read**, *list-so-far* is the symbol **:toplevel**. After a dotted-pair dot, *list-so-far* is the symbol **:after-dot**. *function* may read any number of characters from the input stream and process them however it likes.

function should return three values, called *thing*, *type*, and *splice-p*. *thing* is the object read. If *splice-p* is **nil**, *thing* is the result. If *splice-p* is non-**nil**, then when reading a list *thing* replaces the list being read — often it will be *list-so-far* with something else **nconc**'ed onto the end. At top level and after a dot if *splice-p* is non-**nil** the *thing* is ignored and the macro character does not contribute anything to the result of **read**. *type* is a historical artifact and is not really used; **nil** is a safe value. Most macro character functions return just one value and let the other two default to **nil**.

function should not have any side effects other than on the stream and *list-so-far*. Because of the way the input editor works, *function* can be called

several times during the reading of a single expression in which the macro character only appears once.

char is given the same syntax that single-quote, backquote, and comma have in the initial readtable (it is called **:macro** syntax).

set-syntax-#-macro-char *char function* &optional *readtable* *Function*
Causes *function* to be called when *#char* is read. *readtable* defaults to the current readtable. The function's arguments and return values are the same as for normal macro characters, documented above. When *function* is called, the special variable **si:xr-sharp-argument** contains **nil** or a number that is the number or special bits between the # and *char*.

set-syntax-from-description *char description* &optional *readtable* *Function*
Sets the syntax of *char* in *readtable* to be that described by the symbol *description*. The following descriptions are defined in the standard readtable:

si:alphabetic An ordinary character such as "A".

si:break A token separator such as "(". (Obviously left parenthesis has other properties besides being a break.)

si:whitespace A token separator that can be ignored, such as " ".

si:single A self-delimiting single-character symbol. The initial readtable does not contain any of these.

si:slash The character quoter. In the initial readtable this is "/".

si:verticalbar The symbol print-name quoter. In the initial readtable this is "|".

si:doublequote The string quoter. In the initial readtable this is "".

si:macro A macro character. Do not use this; use **set-syntax-macro-char**.

si:circlecross The octal escape for special characters. In the initial readtable this is "⊗".

si:bitscale A character that causes the fixnum to its left to be doubled the number of times indicated by the fixnum to its right. In the initial readtable this is "_". See the section "What the Reader Accepts".

si:digitscale A character that causes the fixnum to its left to be multiplied by **ibase** the number of times indicated by the fixnum to its right. In the initial readtable this is "^". See the section "What the Reader Accepts".

si:non-terminating-macro
A macro character that is not a token separator. This is a macro character if seen alone but is just a symbol

constituent inside a symbol. You can use it as a character of a symbol other than the first without slashing it. (# would be one of these if it were not built into the reader.)

These symbols will probably be moved to the standard keyword package at some point. *readtable* defaults to the current readtable.

setsyntax *character arg2 arg3* *Function*

This exists only for Maclisp compatibility. The above functions are preferred in new programs. The syntax of *character* is altered in the current readtable, according to *arg2* and *arg3*. *character* can be a fixnum, a symbol, or a string, that is, anything acceptable to the **character** function. *arg2* is usually a keyword; it can be in any package since this is a Maclisp compatibility function. The following values are allowed for *arg2*:

:macro The character becomes a macro character. *arg3* is the name of a function to be invoked when this character is read. The function takes no arguments, can **tyi** or **read** from **standard-input** (that is, can call **tyi** or **read** without specifying a stream), and returns an object which is taken as the result of the read.

:splicing Like **:macro**, but the object returned by the macro function is a list that is **nconc**ed into the list being read. If the character is read not inside a list (at top level or after a dotted-pair dot), then it may return **()**, which means it is ignored, or **(*obj*)**, which means that *obj* is read.

:single The character becomes a self-delimiting single-character symbol. If *arg3* is a fixnum, the character is translated to that character.

nil The syntax of the character is not changed, but if *arg3* is a fixnum, the character is translated to that character.

a symbol The syntax of the character is changed to be the same as that of the character *arg2* in the standard initial readtable. *arg2* is converted to a character by taking the first character of its print name. Also if *arg3* is a fixnum, the character is translated to that character.

setsyntax-sharp-macro *character type function* &optional *readtable* *Function*

This exists only for Maclisp compatibility. **set-syntax-#-macro-char** is preferred. If *function* is **nil**, *#character* is turned off, otherwise it becomes a macro that calls *function*. *type* can be **:macro**, **:peek-macro**, **:splicing**, or **:peek-splicing**. The splicing part controls whether *function* returns a single object or a list of objects. Specifying peek causes *character* to remain in the input stream when *function* is called; this is useful if *character* is something

like a left parenthesis. *function* gets one argument, which is **nil** or the number between the # and the *character*.

8. Input Functions

Most of these functions take optional arguments called *stream* and *eof-option*. *stream* is the stream from which the input is to be read; if unsupplied it defaults to the value of **standard-input**. The special pseudostreams **nil** and **t** are also accepted, mainly for Maclisp compatibility. **nil** means the value of **standard-input** (that is, the default) and **t** means the value of **terminal-io** (that is, the interactive terminal). This is all more or less compatible with Maclisp, except that instead of the variable **standard-input** Maclisp has several variables and complicated rules. See the section "What Streams Are". Streams are documented in detail in that section.

eof-option controls what happens if input is from a file (or any other input source that has a definite end) and the end of the file is reached. If no *eof-option* argument is supplied, an error is signalled. If there is an *eof-option*, it is the value to be returned. Note that an *eof-option* of **nil** means to return **nil** if the end of the file is reached; it is *not* equivalent to supplying no *eof-option*.

Functions such as **read** that read an "object" rather than a single character always signal an error, regardless of *eof-option*, if the file ends in the middle of an object. For example, if a file does not contain enough right parentheses to balance the left parentheses in it, **read** complains. If a file ends in a symbol or a number immediately followed by end-of-file, **read** reads the symbol or number successfully and when called again, sees the end-of-file and obey *eof-option*. If a file contains ignorable text at the end, such as blank lines and comments, **read** does not consider it to end in the middle of an object and obeys *eof-option*.

These end-of-file conventions are not completely compatible with Maclisp. Maclisp's deviations from this are generally considered to be bugs rather than features.

The functions below that take *stream* and *eof-option* arguments can also be called with the stream and eof-option in the other order. This functionality is only for compatibility with old Maclisp programs, and should never be used in new programs. The functions attempt to figure out which way they were called by seeing whether each argument is a plausible stream. Unfortunately, there is an ambiguity with symbols: a symbol might be a stream and it might be an eof-option. If there are two arguments, one being a symbol and the other being something that is a valid stream, or only one argument, which is a symbol, then these functions interpret the symbol as an eof-option instead of as a stream. To force them to interpret a symbol as a stream, give the symbol an **si:io-stream-p** property whose value is **t**.

Note that all of these functions echo their input if used on an interactive stream (one that supports the **:rubout-handler** operation. 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.

read &optional (*stream* **standard-input**) *eof-option* *Function*
input-editor-options

read reads in the printed representation of a Lisp object from *stream*, builds a corresponding Lisp object, and returns the object. For details: See the section "Input Functions".

(This function can take its arguments in the other order, for Maclisp compatibility only.)

read-preserve-delimiters *Variable*

Certain printed representations given to **read**, notably those of symbols and numbers, require a delimiting character after them. (Lists do not, because the matching close parenthesis serves to mark the end of the list.) Normally **read** throws away the delimiting character if it is "whitespace", but preserves it (with a **:untyi** stream operation) if the character is syntactically meaningful, since it may be the start of the next expression.

If **read-preserve-delimiters** is bound to **t** around a call to **read**, no delimiting characters are thrown away, even if they are whitespace. This may be useful for certain reader macros or special syntaxes.

read-or-end &optional (*stream* **standard-input**) *eof-option* *Function*
input-editor-options

This function is like **read**, except that if it is reading from an interactive stream and the user presses END as the first character or the first character after only whitespace characters, it returns two values, **nil** and **:end**. If it encounters any nonwhitespace characters, END has the same meaning as for **read**. *eof-option* has the same meaning as for other reading functions. *input-editor-options* are passed to the input editor if the stream supports it.

The **:expression-or-end** and **:eval-form-or-end** options for **prompt-and-read** invoke **si:read-or-end**.

tyi &optional *stream* *eof-option* *Function*

tyi inputs one character from *stream* and returns it. The character is echoed if *stream* is interactive, except that Rubout is not echoed. The Control, Meta, and so on shifts echo as prefix c-, m-, and so on.

The **:tyi** stream operation is preferred over the **tyi** function for some purposes. Note that it does not echo. See the message **:tyi**.

(This function can take its arguments in the other order, for Maclisp compatibility only)

read-for-top-level &optional (*stream* **standard-input**) *eof-option* *Function*
input-editor-options

This is a slightly different version of **read**. It differs from **read** only in that it ignores close parentheses seen at top level, and it returns the symbol

si:eof if the stream reaches end-of-file if you have not supplied an *eof-option* (instead of signalling an error as **read** would). This version of **read** is used in the system's "read-eval-print" loops.

readline &optional (*stream* **standard-input**) *eof-option* *input-editor-options* *Function*

readline reads in a line of text. If called from inside the input editor or if reading from a stream that does not support the input editor, the line is terminated by a Newline character. If the stream supports the input editor and **readline** is called from outside the input editor, the line is terminated by RETURN, LINE, or END.

This function is usually used to get a line of input from the user. If *stream* supports the input editor, **readline** calls **read-delimited-string**, and *input-editor-options* is passed as the list of options to the input editor.

readline returns four values:

- The line as a character string, without the Newline character.
- An *eof* flag, if *eof-option* was **nil**. This is **t** if the line was terminated because end-of-file was encountered, or **nil** if it was terminated because of a RETURN, LINE, or END character.
- The character that delimited the string.
- Any numeric argument given the delimiter character.

See the function **read-delimited-string**.

readline-trim &optional (*stream* **standard-input**) *eof-option* *input-editor-options* *Function*

readline-trim trims leading and trailing whitespace from string input. "Whitespace" means spaces, tabs, or newlines. It takes the same arguments as the normal **readline** and returns the same four values.

Examples:

```
(readline-trim)   exciting option   RETURN =>
"exciting option"
NIL
141
NIL

(readline-trim)RETURN =>
""
NIL
141
NIL
```

The **:string-trim** option for **prompt-and-read** and **tv:choose-variable-values** uses **readline-trim**.

readline-or-nil &optional (*stream* **standard-input**) *eof-option* *input-editor-options* *Function*

Like **readline-trim**, except that it returns a first value of **nil** instead of the empty string if the input string is empty.

The **:string-or-nil** option for **prompt-and-read** and the **:string-or-nil** choose-variable-values keyword use **readline-or-nil**.

See the function **readline-trim**.

read-delimited-string &optional (*delimiters* **#\end**) (*stream* **standard-input**) (*eof* **nil**) (*input-editor-options* **nil**) &rest (*make-array-args* **'(100. :type art-string)**) *Function*

delimiter is either a character or a list of characters. Characters are read from *stream* until one of the delimiter characters is encountered. The characters read up to the delimiter are returned as a string. This function may be invoked from inside or outside the input editor. If invoked from outside the input editor, the delimiter characters are set up as activation characters. The *eof* argument is treated the same way as the *eof* argument to the **:tyi** message to noninteractive streams. *input-editor-options* are passed on as the first argument to the **:rubout-handler** message, after having an **:activation** entry prepended. *make-array-args* are arguments to be passed to **make-array** when constructing the string to return.

read-delimited-string returns four values:

- The string
- An *eof* flag, if the *eof* parameter was **nil**
- The character that delimited the string
- Any numeric argument given the delimiter character

This function is used by **readline**, **qsend**, and the **:delimited-string** option for **prompt-and-read**.

Examples:

The following reads characters until END is typed and returns a string at least 200. characters long with a leader-length of 3:

```
(read-delimited-string #\end standard-input nil nil 200. :leader-length 3)
```

The following is the same as **(readline)**, except that it does not echo a Newline after the string is activated:

```
(read-delimited-string '(#\return #\line #\end))
```

A simple word parser:

```
(read-delimited-string '(#\space #/, #/. #/?))
```

For a more complex example of a sentence parser that uses *read-delimited-string*: See the section "Examples of Use of the Input Editor".

readch &optional *stream eof-option* *Function*

This function is provided only for Maclisp compatibility, since in the Zetalisp characters are always represented as fixnums. **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. The symbol is interned in the current package. This is just like a Maclisp "character object". (This function can take its arguments in the other order, for Maclisp compatibility only; see the note above.)

tyipeek &optional *peek-type stream eof-option* *Function*

This function is provided mainly for Maclisp compatibility; the **:tyipeek** stream operation is usually clearer.

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**. See the message **:tyipeek**.

If *peek-type* is a fixnum less than 1000 octal, then **tyipeek** reads characters from *stream* until it gets one equal to *peek-type*. That character is not removed from the input stream.

If *peek-type* is **t**, then **tyipeek** skips over input characters until the start of the printed representation of a Lisp object is reached. As above, the last character (the one that starts an object) is not removed from the input stream.

The form of **tyipeek** supported by Maclisp in which *peek-type* is a fixnum not less than 1000 octal is not supported, since the readtable formats of the Maclisp reader and the Zetalisp reader are quite different.

Characters passed over by **tyipeek** are echoed if *stream* is interactive.

The following functions are related functions that do not operate on streams. Most of the text at the beginning of this section does not apply to them.

read-from-string *string* &optional (*eof-option* **'si:no-eof-option**) (*start* **0**) *end* *Function*

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. If *string* has a fill-pointer it controls how much can be read.

eof-option is what to return if the end of the string is reached, as with other reading functions. *start* is the index in the string of the first character to be read. *end*, if given, is used instead of **(array-active-length** *string***)** as the integer that is one greater than the index of the last character to be read.

read-from-string returns two values: The first is the object read and the second is the index of the first character in the string not read. If the entire string was read, this is the length of the string.

Example:

```
(read-from-string "(a b c)") => (a b c) and 7
```

readlist *char-list* *Function*

This function is provided mainly for Maclisp compatibility. *char-list* is a list of characters. The characters may be represented by anything that the function **character** accepts: 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.

If there are more characters in *char-list* beyond those needed to define an object, the extra characters are ignored. If there are not enough characters, an "eof in middle of object" error is signalled.

See the special form **with-input-from-string**.

9. Output Functions

These functions all take an optional argument called *stream*, which is where to send the output. If unsupplied *stream* defaults to the value of **standard-output**. If *stream* is **nil**, the value of **standard-output** (that is, the default) is used. If it is **t**, the value of **terminal-io** is used (that is, the interactive terminal). If *stream* is a list of streams, then the output is performed to all of the streams (this is not implemented yet, and an error is signalled in this case). This is all more or less compatible with Maclisp, except that instead of the variable **standard-output** Maclisp has several variables and complicated rules. See the section "What Streams Are". Streams are documented in detail in that section.

prinl *x* &optional *stream* *Function*
prinl outputs the printed representation of *x* to *stream*, with slashification. *x* is returned. See the section "What the Printer Produces".

prinl-then-space *x* &optional *stream* *Function*
prinl-then-space is like **prinl** except that output is followed by a space.

print *x* &optional *stream* *Function*
print is just like **prinl** except that output is preceded by a carriage return and followed by a space. *x* is returned.

princ *x* &optional *stream* *Function*
princ is just like **prinl** except that the output is not slashified. *x* is returned.

tyo *char* &optional *stream* *Function*
tyo outputs the character *char* to *stream*.

terpri &optional *stream* *Function*
terpri outputs a carriage return character to **stream**.

The **format** function is very useful for producing nicely formatted text. See the function **format**. It can do anything any of the above functions can do, and it makes it easy to produce good-looking messages and such. **format** can generate a string or output to a stream.

The **grindef** function is useful for formatting Lisp programs. See the special form **grindef**.

See the special form **with-output-to-string**.

stream-copy-until-eof *from-stream to-stream* &optional *leader-size* *Function*
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-in** operation.

beep &optional *beep-type (stream* **terminal-io**) *Function*

This function is intended to attract the user's attention by causing an audible beep, or flashing the screen, or something similar. If the stream supports the **:beep** operation, then this function sends it a **:beep** message, passing *type* along as an argument. Otherwise it just causes an audible beep on the terminal. *type* is a keyword selecting among several different beeping noises. The allowed types have not yet been defined; *type* is currently ignored and should always be **nil**. See the message **:beep**.

cursorpos &rest *args* *Function*

This function exists primarily for Maclisp compatibility. Usually it is preferable to send the appropriate messages. See the document *Using the Window System*.

cursorpos normally operates on the **standard-output** stream; however, if the last argument is a stream or **t** (meaning **terminal-io**) then **cursorpos** uses that stream and ignores it when doing the operations described below. Note that **cursorpos** only works on streams that are capable of these operations, such as windows. A stream is taken to be any argument which is not a number and not a symbol, or a symbol other than **nil** with a name more than one character long.

(cursorpos) => (*line* . *column*), the current cursor position.

(cursorpos *line column*) moves the cursor to that position. It returns **t** if it succeeds and **nil** if it does not.

(cursorpos *op*) performs a special operation coded by *op*, and returns **t** if it succeeds and **nil** if it does not. *op* is tested by string comparison, is not a keyword symbol, and can be in any package.

F Moves one space to the right.

B Moves one space to the left.

D Moves one line down.

U Moves one line up.

T Homes up (moves to the top left corner). Note that **t** as the last argument to **cursorpos** is interpreted as a stream, so a stream *must* be specified if the **T** operation is used.

Z Home down (moves to the bottom left corner).

A Advances to a fresh line. See the **:fresh-line** stream operation.

C Clears the window.

E Clear from the cursor to the end of the window.

L Clear from the cursor to the end of the line.

K Clear the character position at the cursor.

X B then K.

exploden *x* *Function*

exploden returns a list of characters (as fixnums) that are the characters that would be typed out by **(princ** *x***)** (that is, the unslashified printed representation of *x*). Example:

```
(exploden '(+ /12 3)) => (50 53 40 61 62 40 63 51)
```

explodec *x* *Function*

explodec returns a list of characters represented by symbols that are the characters that would be typed out by **(princ** *x***)** (that is, 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* *Function*

explode returns a list of characters represented by symbols that are the characters that would be typed out by **(prinl** *x***)** (that is, 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* *Function*

flatsize returns the number of characters in the slashified printed representation of *x*.

flatc *x* *Function*

flatc returns the number of characters in the unslashified printed representation of *x*.

Index

B

B

B

C

C

C

D

D D D

E

G

G

G

H

H

H

M

N

N N

O O O

Q

Q

Q

R

R

R

S

S

S

T T T

U

U U

V

V V

W

W W

symbolics™

EVAL Evaluation

Cambridge, Massachusetts

Evaluation
 # 990056

February 1984

This document corresponds to Release 5.0.

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

No representation or affirmation of fact contained in this document should be construed as a warranty by Symbolics, and its contents are subject to change without notice. Symbolics, Inc. assumes no responsibility for any errors that might appear in this document.

Symbolics software described in this document is furnished only under license, and may be used only in accordance with the terms of such license. Title to, and ownership of, such software shall at all times remain in Symbolics, Inc. Nothing contained herein implies the granting of a license to make, use, or sell any Symbolics equipment or software.

Symbolics is a trademark of Symbolics, Inc., Cambridge, Massachusetts.

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

Table of Contents

1. Introduction

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. See the section "Variables: Evaluation".

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

In all remaining cases, *form* is a list. The evaluator examines the **car** of the list to figure out what to do next. There are three possibilities: this form may be a *special form*, a *macro form*, or a *function form*. Conceptually, the evaluator knows specially about all the symbols whose appearance in the **car** of a form make that form a special form, but the way the evaluator actually works is as follows. If the **car** of the form is a symbol, the evaluator finds the object in the function cell of the symbol and starts all over as if that object had been the **car** of the list. (See the section "Symbols".) If the **car** is not a symbol, then if it is a cons whose **car** is the symbol **macro**, then this is a macro form. If it is a "special function" then this is a special form. See the section "Kinds of Functions". Otherwise, it should be a regular function, and this is a function form.

If *form* is a special form, then it is handled accordingly; each special form works differently. See the section "Kinds of Functions". The internal workings of special forms are explained in more detail in that section, but this hardly ever affects you.

If *form* is a macro form, then the macro is expanded. See the document *Macros*.

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

There is a lot more to be said about evaluation. See the section "Variables: Evaluation". The way variables work and the ways in which they are manipulated, including the binding of arguments, is explained in that section. See the section "Functions: Evaluation". A basic explanation of functions is in that section. See the section "Multiple Values". The way functions can return more than one value is explained there. See the section "Functions: Functions". The description of all of the kinds of functions, and the means by which they are manipulated, is there. See the document *Macros*. The **evalhook** facility lets you do something arbitrary

whenever the evaluator is invoked. See the section **"evalhook"**. Special forms are described all over the documentation set; each special form is in the section on the facility it is part of.

2. Variables

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

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

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

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

Binding is an important part of the process of applying interpreted functions to arguments. See the section "Functions: Evaluation".

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

variable. Local variables do not have any "top-level" value; they do not even exist outside of the form that binds them.

The variables that are associated with symbols (the kind that are used by noncompiled programs) are called "special" variables.

Local variables and special variables do not behave quite the same way, because "binding" means different things for the two of them. Binding a special variable saves the old value away and then uses the value cell of the symbol to hold the new value, as explained above. Binding a local variable, however, does not do anything to the symbol. In fact, it creates a new memory cell to hold the value, that is, a new local variable.

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

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

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

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

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

(compile 'foo)          ;Now compile foo.

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

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

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

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

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

Had **bar** been compiled, the compiler would have seen the free reference and printed a warning message: `Warning: a declared special`. It would have automatically declared **a** to be special and proceeded with the compilation. It knows that free references mean that special declarations are needed. But when a function is compiled that binds a variable that you want to be treated as a special variable but that you have not explicitly declared, there is, in general, no way for the compiler to automatically detect what has happened, and it will produce incorrect output. So you must always provide declarations for all variables that you want to be treated as special variables.

When you declare a variable to be special using **declare** rather than **local-declare**, the declaration is "global"; that is, it applies wherever that variable name is seen. After **fuzz** has been declared special using **declare**, all following uses of **fuzz** will be treated by the compiler as references to the same special variable. Such variables are called "global variables", because any function can use them; their scope is not limited to one function. The special forms **defvar** and **defconst** are useful for creating global variables; not only do they declare the variable special, but they also provide a place to specify its initial value, and a place to add documentation. In addition, since the names of these special forms start with "**def**" and since they are used at the top level of files, the Lisp Machine editor can find them easily.

Here are the special forms used for setting variables.

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

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

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

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

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

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

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

Here are the special forms used for binding variables.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

let-globally is similar in form to **let**. The difference is that **let-globally** does not *bind* the variables; instead, it saves the old values and *sets* the variables, and sets up an **unwind-protect** to set them back. The important difference between **let-globally** and **let** is that when the current stack group calls some other stack group, the old values of the variables are *not* restored. Thus, **let-globally** makes the new values visible in all stack groups and processes that do not bind the variables themselves, not just the current stack group.

let-globally-if *predicate varlist* &body *body...* *Macro*

let-globally-if is like **let-globally**. It takes a predicate form as its first argument. It binds the variables only if *predicate* evaluates to something other than **nil**. *body* is evaluated in either case.

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

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

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

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

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

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

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

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

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

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

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

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

destructuring-bind *variable-pattern data body ...* *Special Form*

destructuring-bind binds variables to values, using **defmacro**'s destructuring facilities, and evaluates the body forms in the context of those bindings.

First *data* is evaluated. If *variable-pattern* is a symbol, it is bound to the result of evaluating *data*. If *variable-pattern* is a tree, the result of evaluating *data* should be a tree of the same shape. The trees are disassembled, and each variable that is a component of *variable-pattern* is bound to the value that is the corresponding element of the tree that results from evaluating *data*. If not enough values are supplied, the remaining variables are bound to **nil**. If too many values are supplied, the excess values are ignored. Finally, the body forms are evaluated sequentially, the old values of the variables are restored, and the result of the last body form is returned.

As with the pattern in a **defmacro** form, *variable-pattern* actually resembles the **lambda**-list of a function; it can have **&**-keywords. See the section "Advanced Features of **defmacro**".

Example:

```
(destructuring-bind (a (b) &optional (c 'd))
                    '((x y) (z))
  (values a b c))
```

returns **(x y)**, **z**, and **d**.

Here are the special forms for defining special variables.

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

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

`(defvar` *variable*`)`

declares *variable* special for the sake of compilation, and records its location for the sake of the editor so that you can ask to see where the variable is defined. If a second subform is supplied,

`(defvar` *variable initial-value*`)`

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

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

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

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

allows you to include a documentation string that describes what the variable is for or how it is to be used. Using such a documentation string is even better than commenting the use of the variable, because the documentation string is accessible to system programs that can show the documentation to you while you are using the machine.

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

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

defconst is the same as **defvar** except that if an initial value is given the variable is always set to it regardless of whether it is already bound. The rationale for this is that **defvar** declares a global variable, whose value is initialized to something but will then be changed by the functions that use it to maintain some state. On the other hand, **defconst** declares a constant, whose value will never be changed by the normal operation of the program, only by changes *to* the program. **defconst** always sets the variable to the specified value so that if, while developing or debugging the program, you change your mind about what the constant value should be, and then you evaluate the **defconst** form again, the variable will get the new value. It is

not the intent of **defconst** to declare that the value of *variable* will never change; for example, **defconst** is *not* license to the compiler to build assumptions about the value of *variable* into programs being compiled.

The special form **defconstant** is used to declare a named constant.

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

defconstant declares the use of a named constant in a program. *initial-value* is evaluated and *variable* set to the result. The value of *variable* is then fixed. It is an error if *variable* has any special bindings at the time the **defconstant** form is executed. Once a special variable has been declared constant by **defconstant**, any further assignment to or binding of that variable is an error.

The compiler is free to build assumptions about the value of the variable into programs being compiled. If the compiler does replace references to the name of the constant by the value of the constant in code to be compiled, the compiler takes care that such "copies" appear to be **eql** to the object that is the actual value of the constant. For example, the compiler may freely make copies of numbers, but it exercises care when the value is a list.

In Zetalisp, **defconstant** and **defconst** are essentially the same if the value is other than a number, a character, or an interned symbol. However, if the variable being declared already has a value, **defconst** freely changes the value, whereas **defconstant** queries before changing the value (unless the **defconstant** form is in a patch file). **defconstant** assumes that changing the value is dangerous because the old value might have been incorporated into compiled code, which would be out of date if the value changed.

In general, you should use **defconstant** to declare constants whose value is a number, character, or interned symbol and is guaranteed not to change. An example is π. The compiler can optimize expressions that contain references to these constants. If the value is another type of Lisp object or if it might change, you should use **defconst** instead.

documentation, if provided, should be a string. It is accessible to the **documentation** function.

3. Functions

In an earlier description of evaluation, we said that evaluation of a function form works by applying the function to the results of evaluating the argument subforms. What is a function, and what does it mean to apply it? In Zetalisp there are many kinds of functions, and applying them can do many different kinds of things. See the section "Functions: Functions". Here we will explain the most basic kinds of functions and how they work. In particular, this chapter explains *lambda lists* and all their important features.

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

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

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

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

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

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

The principal weakness of the simple lambda lists is that any function written with one must only take a certain fixed number of arguments. As we know, many very useful functions, such as **list**, **append**, **+**, and so on, accept a varying number of arguments. Maclisp solved this problem by the use of *lexprs* and *lsubrs*, which were somewhat inelegant since the parameters had to be referred to by numbers instead

of names (for example, **(arg 3)**). (For compatibility reasons, Zetalisp supports *lexprs*, but they should not be used in new programs). Simple lambda lists also require that arguments be matched with parameters by their position in the sequence. This makes calls hard to read when there are a great many arguments. Keyword parameters enable the use of other styles of call which are more readable.

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

Keyword arguments are always optional, regardless of whether the lambda list contains **&optional**. Any **&optional** appearing after the first keyword argument has no effect. **&key** and **&rest** are independent. They can both appear and they both use the same arguments from the argument list. The only rule is that **&rest** must appear before **&key** in the lambda list.

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

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

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

The positional parameters are dealt with first.

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

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

the parameter is bound to it, then the next default-form is evaluated, and so on. This allows the default for an argument to depend on the previous argument.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This illustrates that the parameters can be matched in either order.

`(&key a &optional b)`
a is required keyword, and **b** is optional keyword. The sample call above would be legal for this function also; so would

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

which doesn't specify **b**.

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

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

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

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

`(&rest z &key &allow-other-keys)`
This is equivalent to **(&rest z)**. So, for that matter, is the

previous example, if the function does not use the values of **a**, **b** and **c**.

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

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

```
(&optional (a 'foo) &rest d &key b (c (symeval a)))
```
a's default-form is **'foo**, **b**'s is **nil**, and **c**'s is **(symeval a)**. Note that if the function whose lambda list this is were called on no arguments, **a** would be bound to the symbol **foo**, and **c** would be bound to the 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 can take advantage of earlier parameters in the lambda list. **b** and **d** would be bound to **nil**.

Occasionally it is important to know whether a certain optional parameter was defaulted or not. You can't tell from just examining its value, since if the value is the default value, there's no way to tell whether the caller passed that value explicitly, or whether the caller did not pass any value and the parameter was defaulted. The way to tell for sure is to put a third element into the list: the third element should be a variable (a symbol), and that variable is bound to **nil** if the parameter was not passed by the caller (and so was defaulted), or **t** if the parameter was passed. The new variable is called a "supplied-p" variable; it is bound to **t** if the parameter is supplied. For example:

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

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

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

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

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

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

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

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

It is important to realize that the list of arguments to which a rest-parameter is bound is set up in whatever way is most efficiently implemented, rather than in the way that is most convenient for the function receiving the arguments. It is not guaranteed to be a "real" list. Sometimes the rest-args list is 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, as you must always assume that it is one of these special lists. See the function **copylist**. The system will not detect the error of omitting to copy a rest-argument; you will simply find that you have a value which seems to change behind your back. At other times the rest-args list will be an argument that was given to **apply**; therefore it is not safe to **rplaca** this list as you might modify permanent data structure. An attempt to **rplacd** a rest-args list will be unsafe in this case, while in the first case it would cause an error, since lists in the stack are impossible to **rplacd**.

There are some other keywords in addition to those mentioned here. See the section "Lambda-list Keywords".

4. Some Functions and Special Forms

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

eval *x* *Function*

(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 that 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 Zetalisp. They are replaced by closures. See the section "Closures".

apply *f arglist* *Function*

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

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

Of course, *arglist* can be **nil**. Note: unlike Maclisp, **apply** never takes a third argument; there are no "binding context pointers" in Zetalisp.

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

funcall *f* &rest *args* *Function*

(funcall *f a1 a2 ... an***)** applies the function *f* to the arguments *a1*, *a2*, ...,

an. *f* cannot 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
(cons 1 2) => (1 . 2)
```

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

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

lexpr-funcall is like a cross between **apply** and **funcall**. **(lexpr-funcall** *f a1 a2 ... an l***)** applies the function *f* to the arguments *a1* through *an* followed by the elements of the list *l*. Note that since it treats its last argument specially, **lexpr-funcall** requires at least two arguments. Examples:

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

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

lexpr-funcall with two arguments does the same thing as **apply**.

send is the new official function to use to send messages to objects. It should be used in the same way that **funcall** has been used up to now.

send *object message-name* &rest *arguments* *Function*

Sends the message named *message-name* to the *object*. *arguments* are the arguments passed.

Currently, **send** does exactly the same thing as **funcall**. However, in a future release, it will be possible to send messages to objects of any data type, and **send** will be changed upward-compatibly to make this work.

Another new function, **lexpr-send**, is to **send** as **lexpr-funcall** is to **funcall**.

lexpr-send *object message-name* &rest *arguments* *Function*

Sends the message named *message-name* to the *object*. *arguments* are the arguments passed, except that the last element of *arguments* should be a list, and all the elements of that list are passed as arguments. Example:

```
(send some-window ':set-edges 10 10 40 40)
```

does the same thing as

```
(setq new-edges '(10 10 40 40))
(lexpr-send some-window ':set-edges new-edges)
```

Note: **send-self** or **lexpr-send-self** do not exist, because the new implementation of Flavors eliminates any particular performance benefit. To send a message to **self**, pass **self** as the first argument to **send**.

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

call *function* &rest *argument-specifications* *Function*

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

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

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

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

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

quote *object* *Special Form*

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

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

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

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

is converted by **read** into

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

function *f* *Special Form*

This means different things depending on whether *f* is a function or the name of a function. (Note that in neither case is *f* evaluated.) The name of a function is a symbol or a function-spec list. See the section "Function Specs". A function is typically a list whose car is the symbol **lambda**; however there are several other kinds of functions available. See the section "Kinds of Functions".

If you want to pass an anonymous function as an argument to a function, you could just use **quote**. For example:

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

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

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

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

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

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

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

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

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

```
(function fred)
```

is like

```
(fdefinition 'fred)
```

which is like

```
(fsymeval 'fred)
```

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

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

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

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

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

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

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

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

false *Function*

Takes no arguments and returns **nil**.

true *Function*

Takes no arguments and returns **t**.

ignore &rest *ignore* *Function*

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

comment *Special Form*

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

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

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

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

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

progn *body...* *Special Form*

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

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

(When *form1* is **'compile**, the **progn** form has a special meaning to the compiler. See the section "Macros Expanding Into Many Forms".)

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

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

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

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

prog1 never returns multiple values. See the special form **multiple-value-prog1**.

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

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

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

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

arg *x* *Function*

(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* *Function*

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* *Function*

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

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

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

5. Multiple Values

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

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

The special forms for receiving multiple values are **multiple-value**, **multiple-value-bind**, **multiple-value-list**, **multiple-value-call**, and **multiple-value-prog1**. These consist of a form and an indication of where to put the values returned by that form. With the first two of these, the caller requests a certain number of returned values. If fewer values are returned than the number requested, then it is exactly as if the rest of the values were present and had the value **nil**. If too many values are returned, the rest of the values are ignored. This has the advantage that you do not have to pay attention to extra values if you don't care about them, but it has the disadvantage that error-checking similar to that done for function calling is not present.

values &rest *args* *Function*

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

values-list *list* *Function*

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

return and its variants can only be used within the **do** and **prog** special forms and their variants. See the section "Iteration".

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

multiple-value is a special form used for calling a function which is expected

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

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

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

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

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

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

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

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

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

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

multiple-value-call *function body ...* *Special Form*

multiple-value-call first evaluates *function* to obtain a function. It then evaluates all the forms in *body*, gathering together all the values of the forms (not just one value from each). It gives these values as arguments to the function and returns whatever the function returns.

For example, suppose the function **frob** returns the first two elements of a list of numbers:

```
(multiple-value-call #'+ (frob '(1 2 3)) (frob '(4 5 6)))
  <=> (+ 1 2 4 5) => 12.
```

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

multiple-value-prog1 is like **prog1**, except that if its first form returns multiple values, **multiple-value-prog1** returns those values.

Due to the syntactic structure of Lisp, it is often the case that the value of a certain form is the value of a subform of it. For example, the value of a **cond** is the value of the last form in the selected clause. In most such cases, if the subform produces multiple values, the original form will also produce all of those values. This *passing-back* of multiple values of course has no effect unless eventually one of the special forms for receiving multiple values is reached. The exact rule governing passing-back of multiple values is as follows:

If *X* is a form, and *Y* is a subform of *X*, then if the value of *Y* is unconditionally returned as the value of *X*, with no intervening computation, then all the multiple values returned by *Y* are returned by *X*. In all other cases, multiple values or only single values may be returned at the discretion of the implementation; users should not depend on whatever way it happens to work, as it might change in the future or in other implementations. The reason we do not guarantee nontransmission of multiple values is because such a guarantee would not be very useful and the efficiency cost of enforcing it would be high. Even **setq**'ing a variable to the result of a form, then returning the value of that variable might be made to pass multiple values by an optimizing compiler which realized that the **setq**ing of the variable was unnecessary.

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

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

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

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

progn passes back multiple values from its last form. **progv** and **progw** do so also.

prog1 and **prog2**, however, do not pass back multiple values (though **multiple-value-prog1** does).

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

cond passes back multiple values from the last form in the selected clause, but not if the clause is only one long (that is, the returned value is the value of the predicate) since the return is conditional. This rule applies even to the last clause, where the return is not really conditional (the implementation is allowed to pass or not to pass multiple values in this case, and so you should not depend on what it does). **t** should be used as the predicate of the last clause if multiple values are desired, to make it clear to the compiler (and any human readers of the code!) that the return is not conditional.

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

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

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

unwind-protect passes back multiple values from its protected form.

***catch** does not pass back multiple values from the last form in its body, because it is defined to return its own second value to tell you whether the ***catch** form was exited normally or abnormally. This is sometimes inconvenient when you want to propagate back multiple values but you also want to wrap a ***catch** around some forms. Usually people get around this problem by enclosing the ***catch** in a **prog** and using **return** to pass out the multiple values, **returning** through the ***catch**.

Index

F F F

G

G

G

I

I

I

K

K

K

L

L

L

M

M

M

T

T T

U

U U

V

V V

Variables in lambda lists 11
Variables: Evaluation 3

symbolics™

FLOW Flow of Control

Cambridge, Massachusetts

Flow of Control

 # 990045

March 1984

This document corresponds to Release 5.0.

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

No representation or affirmation of fact contained in this document should be construed as a warranty by Symbolics, and its contents are subject to change without notice. Symbolics, Inc. assumes no responsibility for any errors that might appear in this document.

Symbolics software described in this document is furnished only under license, and may be used only in accordance with the terms of such license. Title to, and ownership of, such software shall at all times remain in Symbolics, Inc. Nothing contained herein implies the granting of a license to make, use, or sell any Symbolics equipment or software.

Symbolics is a trademark of Symbolics, Inc., Cambridge, Massachusetts.

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

Table of Contents

1. Introduction

Lisp provides a variety of structures for flow of control.

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

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

The performing of an action repeatedly (usually with some changes between repetitions) is called *iteration*, and is provided as a basic control structure in most languages. The *do* statement of PL/I, the *for* statement of ALGOL/60, and so on are examples of iteration primitives. Lisp provides two general iteration facilities: **do** and **loop**, as well as a variety of special-purpose iteration facilities. (**loop** is sufficiently complex that it is explained in its own chapter later in this document. See the section "The Loop Iteration Macro".) There is also a very general construct to allow the traditional "goto" control structure, called **prog**.

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

There are some *nonlocal exit* control structures, analogous to the *leave*, *exit*, and *escape* constructs in many modern languages. The general ones are **catch** and **throw**; there is also **return** and its variants, used for exiting the iteration constructs **do**, **loop**, and **prog**.

Zetalisp also provides a coroutine capability and a multiple-process facility. See the section "Stack Groups". See the document *Processes*. There is also a facility for generic function calling using message passing. See the document *Objects, Message Passing, and Flavors*.

2. Conditionals

if *Special Form*

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

(if *predicate-form then-form*)

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

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

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

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

If there are more than three subforms, **if** assumes you want more than one *else-form*; they are evaluated sequentially and the result of the last one is returned, if the predicate returns **nil**. There is disagreement as to whether this constitutes good programming style or not.

cond *Special Form*

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

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

The idea is that each clause represents a case that 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 consequent forms that 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 evaluates to **nil**, and thus no case is selected, the value of the **cond** is **nil**. Example:

```
(cond ((zerop x)     ;First clause:
       (+ y 3))      ; (zerop x) is the 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.  If z is non-nil, it will be returned.
      (t             ;An antecedent of t
       105)          ; is always satisfied.
  )                  ;This is the end of the cond.
```

cond-every *Special Form*

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

and *form...* *Special Form*

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

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

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

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

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

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

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

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

(and bright-day
     glorious-day
     (princ "It is a bright and glorious day."))
```

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

or *form...* *Special Form*

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

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

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

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

when *test body...* *Macro*

The forms in *body* are evaluated when *test* returns non-null. In that case, it returns the value(s) of the last form evaluated. When *test* returns **nil**, **when** returns **nil**.

```
(when (eq 1 1) (setq a b) "foo") =>
"foo"
(when (eq 1 2) (setq a b) "foo") =>
NIL
```

When *body* is empty, **when** always returns **nil**.

unless *test body...* *Macro*

The forms in *body* are evaluated when *test* returns **nil**. It returns the value of the last form evaluated. When *test* returns something other than **nil**, **unless** returns **nil**.

```
(unless (eq 1 1) (setq a b) "foo") =>
NIL
(unless (eq 1 2) (setq a b) "foo") =>
"foo"
```

When *body* is empty, **unless** always returns **nil**.

selectq *Special Form*

selectq is a conditional that chooses one of its clauses to execute by comparing the value of a form against various constants, which are typically keyword symbols. Its form is as follows:

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

The first thing **selectq** does is to evaluate *key-form*; call the resulting value *key*. Then **selectq** considers each of the clauses in turn. If *key* matches the

clause's *test*, the consequents of this clause are evaluated, and **selectq** returns the value of the last consequent. If there are no matches, **selectq** returns **nil**.

A *test* may be any of the following:

A symbol	If the *key* is **eq** to the symbol, it matches.
A number	If the *key* is **eq** to the number, it matches. Only small numbers (*fixnums*) will work.
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.
t or **otherwise**	The symbols **t** and **otherwise** are special keywords that match anything. Either symbol may be used; **t** is mainly for compatibility with Maclisp's **caseq** construct. To be useful, this should be the last clause in the **selectq**.

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

```
(selectq x
  (foo (do-this))
  (bar (do-that))
  ((baz quux mum) (do-the-other-thing))
  (otherwise (ferror nil "Never heard of ~S" x)))
```

is equivalent to:

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

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

select *Special Form*

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

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

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

is equivalent to:

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

selector *Special Form*

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

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

is equivalent to:

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

typecase *form clauses...* *Special Form*

typecase is a special form for selecting various forms to be evaluated depending on the type of some object. It is something like **select**. A **typecase** form looks like:

```
(typecase form
   (types consequent consequent ...)
   (types consequent consequent ...)
   ...
   )
```

form is evaluated, producing an object. **typecase** examines each clause in sequence. *types* in each clause is either a single type (if it is a symbol) or a list of types. If the object is of that type, or of one of those types, then the consequents are evaluated and the result of the last one is returned. Otherwise, **typecase** moves on to the next clause. As a special case, *types* can be **otherwise**; in this case, the clause is always executed, so this should be used only in the last clause. For an object to be of a given type means that if **typep** is applied to the object and the type, it returns **t**. That is, a type is something meaningful as a second argument to **typep**. Example:

```
(defun tell-about-car (x)
  (typecase (car x)
     (:fixnum "The car is a number.")
     ((:string :symbol) "The car is a name.")
     (otherwise "I don't know.")))

(tell-about-car '(1 a)) => "The car is a number."
(tell-about-car '(a 1)) => "The car is a name."
(tell-about-car '("word" "more")) => "The car is a name."
(tell-about-car '(1.0)) =>
"I don't know."
```

dispatch *Special Form*

(dispatch *byte-specifier number clauses...***)** is the same as **select** (not **selectq**), but the key is obtained by evaluating **(ldb** *byte-specifier number***)**. *byte-specifier* and *number* are both evaluated. See the section "Byte Manipulation Functions". Byte specifiers and **ldb** are explained in that section. 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 that will be dispatched on.

selectq-every *Special Form*

selectq-every has the same syntax as **selectq**, but, like **cond-every**, executes every selected clause instead of just the first one. If an **otherwise** clause is present, it is selected if and only if no preceding clause is selected. The value returned is the value of the last form in the last selected clause. Multiple values are not returned. Example:

```
(selectq-every animal
  ((cat dog) (setq legs 4))
  ((bird man) (setq legs 2))
  ((cat bird) (put-in-oven animal))
  ((cat dog man) (beware-of animal)))
```

caseq *Special Form*

The **caseq** special form is provided for Maclisp compatibility. It is exactly the same as **selectq**. This is not perfectly compatible with Maclisp, because **selectq** accepts **otherwise** as well as **t** where **caseq** would not accept **otherwise**, and because Maclisp does some error checking that **selectq** does not. Maclisp programs that use **caseq** will work correctly so long as they do not use the symbol **otherwise** as the key.

3. Blocks and Exits

block and **return-from** are the primitive special forms for premature exit from a piece of code. **block** defines a place that can be exited, and **return-from** transfers control to such an exit.

block and **return-from** differ from **catch** and **throw** in their scoping rules. **block** and **return-from** have lexical scope; **catch** and **throw** have dynamic scope. See the section "Nonlocal Exits: Flow of Control".

block *name form...* *Special Form*

block evaluates each *form* in sequence and normally returns the (possibly multiple) values of the last *form*. However, **(return-from** *name value***)** or one of its variants (a **return** or **return-list** form) might be evaluated during the evaluation of some *form*. In that case, the (possibly multiple) values that result from evaluating *value* are immediately returned from the innermost block that has the same name and that lexically contains the **return-from** form. Any remaining forms in that block are not evaluated.

name is not evaluated. It must be a symbol.

The scope of *name* is lexical. That is, the **return-from** form must be inside the block itself (or inside a block that that block lexically contains), not inside a function called from the block.

do, **prog**, and their variants establish implicit blocks around their bodies; you can use **return-from** to exit from them. These blocks are named **nil** unless you specify a name explicitly.

For example, the following two forms are equivalent:

```
(cond ((predicate x)
       (do-one-thing))
      (t
       (format t "The value of X is ~S~%" x)
       (do-the-other-thing)
       (do-something-else-too)))

(block deal-with-x
  (when (predicate x)
    (return-from deal-with-x (do-one-thing)))
  (format t "The value of X is ~S~%" x)
  (do-the-other-thing)
  (do-something-else-too))
```

return-from *name value...* *Special Form*

return-from is the primitive special form for exiting from a **block** or a construct like **do** or **prog** that establishes an implicit block around its body.

The *value* subforms are optional. Any *value* forms are evaluated, and the resulting values (possibly multiple, possibly none) are returned from the innermost block that has the same name and that lexically contains the **return-from** form. The returned values depend on how many *value* subforms are provided:

value subforms	*Values returned from block*
None	None
1	All values that result from evaluating the *value* subform
>1	One value from each *value* subform

This means that

(return-from *name form1 form2 form3***)**

is the same as

(return-from *name* **(values** *form1 form2 form3***))**

but the latter form is the preferred way to return multiple values, for the sake of both clarity and compatibility with Common Lisp.

name is not evaluated. It must be a symbol.

The scope of *name* is lexical. That is, the **return-from** form must be inside the block itself (or inside a block that that block lexically contains), not inside a function called from the block.

When a construct like **do** or an unnamed **prog** establishes an implicit block, its name is **nil**. You can use either **(return-from nil** *value...***)** or the equivalent **(return** *value...***)** to exit from such a construct.

The **return-from** form is unusual: It never returns a value itself, in the conventional sense. It is not useful to write **(setq a (return-from** *name* **3))**, because when the **return-from** form is evaluated, the containing block is immediately exited, and the **setq** never happens.

For an explanation of named **dos** and **progs**: See the special form **do-named**.

Following is an example, returning a single value from an implicit block named **nil**:

```
(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-from nil n))))
```

Following is another example, returning multiple values. The function below is like **assq**, but it returns an additional value, the index in the table of the entry it found:

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

return *value...* *Special Form*

return can be used to exit from a construct like **do** or an unnamed **prog** that establishes an implicit block around its body. In this case the name of the block is **nil**, and **(return** *value...***)** is the same as **(return-from nil** *value...***)**. See the special form **return-from**.

In addition, **break** recognizes the typed-in form **(return** *value***)** specially. If this form is typed at a **break**, *value* is evaluated and returned as the value of **break**. Only the result of the first *value* form is returned, but if this form itself returns multiple values, they are all returned as the value of **break**. That is, **(return 'foo 'bar)** returns only **foo**, but **(return (values 'foo 'bar))** returns both **foo** and **bar**. See the special form **break**.

It is legal to write simply **(return)**, which exits from the block without returning any values. **(return)** inside a break loop causes **break** to return **nil**.

If not specially recognized by **break** and not inside a block, **return** signals an error.

return-list *list* *Function*

return-list is an obsolete function supported for compatibility with earlier releases. It is like **return** except that the block returns all of the elements of *list* as multiple values. This means that

(return-list *list***)**

is the same as

(return (values-list *list***))**

but the latter form is the preferred way to return list elements as multiple values from a block named **nil**. To direct the returned values to a named block, use:

(return-from *name* **(values-list** *list*)).

4. Transfer of Control

tagbody and **go** are the primitive special forms for unstructured transfer of control. **tagbody** defines places that can receive a transfer of control, and **go** transfers control to such a place.

tagbody *tag-or-statement...* *Special Form*

The body of a **tagbody** form is a series of *tags* or *statements*. A *tag* is a symbol; a *statement* is a list. **tagbody** processes each element of the body in sequence. It ignores *tags* and evaluates *statements*, discarding the results. If it reaches the end of the body, it returns **nil**.

If a (**go** *tag*) form is evaluated during evaluation of a *statement*, **tagbody** searches its body and the bodies of any **tagbody** forms that lexically contain it. Control is transferred to the innermost *tag* that is **eq** to the *tag* in the **go** form. Processing continues with the next *tag* or *statement* that follows the *tag* to which control is transferred.

The scope of the *tags* is lexical. That is, the **go** form must be inside the **tagbody** construct itself (or inside a **tagbody** form that that **tagbody** lexically contains), not inside a function called from the **tagbody**.

do, **prog**, and their variants use implicit **tagbody** constructs. You can provide *tags* within their bodies and use **go** forms to transfer control to the *tags*.

For example, the following two forms are equivalent:

```
(dotimes (i n) (print i))

(let ((i 0))
  (when (plusp n)
    (tagbody
      loop
      (print i)
      (setq i (1+ i))
      (when (< i n) (go loop)))))
```

go *tag* *Special Form*

go is the primitive special form for transferring control within a **tagbody** form or a construct like **do** or **prog** that uses an implicit **tagbody**.

The *tag* must be a symbol. It is not evaluated. **go** transfers control to the tag in the body of the **tagbody** that is **eq** to the *tag* in the **go** form. If the body has no such tag, the bodies of any lexically containing **tagbody** forms are examined as well. If no tag is found, an error is signalled.

The scope of *tag* is lexical. That is, the **go** form must be inside the

tagbody construct itself (or inside a **tagbody** form that that **tagbody** lexically contains), not inside a function called from the **tagbody**.

Example:

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

5. Iteration

do *Special Form*

The **do** special form provides a simple generalized iteration facility, with an arbitrary number of "index variables" whose values are saved when the **do** is entered and restored when it is left, that is, 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 you to specify a predicate that 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 more general, so-called "new-style" **do** looks like:

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

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

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

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

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

Note that the *end-test* gets evaluated before the first time the body is evaluated. **do** first initializes the variables from the *init* forms, then it checks the *end-test*, then it processes the body, then it deals with the *repeat* forms, then it tests the *end-test* again, and so on. If the **end-test** returns a non-**nil** value the first time, then the body will never be processed.

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

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

If a **return** special form is evaluated inside the body of a **do**, then the **do** immediately stops, unbinds its variables, and returns the values given to **return**. See the special form **return**. **return** and its variants are explained in more detail in that section. **go** special forms and **prog**-tags can also be used inside the body of a **do** and they mean the same thing that they do inside **prog** forms, but we discourage their use since they complicate the control structure in a hard-to-understand way.

The other, so-called "old-style" **do** looks like:

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

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

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 that
                                ; 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)) ; Note how the setq is avoided.

(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.
     w)               ; w starts as nil and is not changed by the do.
    (nil)             ; The end-test is nil, so this is an infinite loop.
  body)           ; Presumably the body uses return somewhere.
```

The construction:

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

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

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.

The following is like `(maplist 'f x y)`:

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

See the section "Mapping".

do* *Special Form*

do* is just like **do** except that the variable clauses are evaluated sequentially rather than in parallel. When a **do** starts, all the initialization forms are evaluated before any of the variables are set to the results; when a **do*** starts, the first initialization form is evaluated, then the first variable is set

to the result, then the second initialization form is evaluated, and so on. The stepping forms work analogously.

Also see **loop**, a general iteration facility based on a keyword syntax rather than a list-structure syntax.

do-named *Special Form*

Sometimes one **do** is contained inside the body of an outer **do**. The **return** function always returns from the innermost surrounding **do**, but sometimes you want to return from an outer **do** while within an inner **do**. You can do this by giving the outer **do** a name. You use **do-named** instead of **do** for the outer **do**, and use **return-from**, specifying that name, to return from the **do-named**.

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

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

If the symbol **t** is used as the name, it is made "invisible" to **returns**; that is, **returns** inside that **do-named** return to the next outermost level whose name is not **t**. **(return-from t ...)** returns from a **do-named** named **t**. You can also make a **do-named** invisible to **returns** by including immediately inside it the form **(declare (invisible-block t))**. This feature is not intended to be used by user-written code; it is for macros to expand into.

If the symbol **nil** is used as the name, it is as if this were a regular **do**. Not having a name is the same as being named **nil**.

progs and **loop**s can have names just as **do**s can. Since the same functions are used to return from all of these forms, all of these names are in the same namespace; a **return** returns from the innermost enclosing iteration form, no matter which of these it is, and so you need to use names if you nest any of them within any other and want to return to an outer one from inside an inner one.

do*-named *Special Form*

do*-named is just like **do-named** except that the variable clauses are evaluated sequentially, rather than in parallel. See **do***.

dotimes *(index count) body...* *Special Form*

dotimes is a convenient abbreviation for the most common integer iteration. **dotimes** performs *body* the number of times given by the value of *count*, with *index* bound to **0**, **1**, and so forth on successive iterations. Example:

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

is equivalent to:

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

except that the name **count** is not used. Note that **i** takes on values starting at 0 rather than 1, and that it stops before taking the value **(// m n)** rather than after. You can use **return** and **go** and **prog**-tags inside the body, as with **do**. **dotimes** forms return **nil** unless returned from explicitly with **return**. For example:

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

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

dolist *(item list) body...* *Special Form*

dolist is a convenient abbreviation for the most common list iteration. **dolist** performs *body* once for each element in the list which is the value of *list*, with *item* bound to the successive elements. Example:

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

is equivalent to:

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

except that the name **lst** is not used. You can use **return** and **go** and **prog**-tags inside the body, as with **do**. **dolist** forms return **nil** unless returned from explicitly with **return**.

keyword-extract *Special Form*

keyword-extract is an aid to writing functions that take keyword arguments in the standard fashion. The form:

```
(keyword-extract key-list iteration-var
        keywords flags other-clauses...)
```

will parse the keywords out into local variables of the function. *key-list* is a form that evaluates to the list of keyword arguments; it is generally the function's **&rest** argument. *iteration-var* is a variable used to iterate over the list; sometimes *other-clauses* will use the form

```
(car (setq iteration-var (cdr iteration-var)))
```

to extract the next element of the list. (Note that this is not the same as **pop**, because it does the **car** after the **cdr**, not before.)

keywords defines the symbols that are keywords to be followed by an argument. Each element of *keywords* is either the name of a local variable that receives the argument and is also the keyword, or a list of the keyword and the variable, for use when they are different or the keyword is not to go in the keyword package. Thus, if *keywords* is **(foo (ugh bletch) bar)** then the keywords recognized will be **:foo**, **ugh**, and **:bar**. If **:foo** is specified its argument will be stored into **foo**. If **:bar** is specified its argument will be stored into **bar**. If **ugh** is specified, its argument will be stored into **bletch**.

Note that **keyword-extract** does not bind these local variables; it assumes you will have done that somewhere else in the code that contains the **keyword-extract** form.

flags defines the symbols that are keywords not followed by an argument. If a flag is seen its corresponding variable is set to **t**. (You are assumed to have initialized it to **nil** when you bound it with **let** or **&aux**.) As in *keywords*, an element of *flags* may be either a variable from which the keyword is deduced, or a list of the keyword and the variable.

If there are any *other-clauses*, they are **selectq** clauses selecting on the keyword being processed. These clauses are for handling any keywords that are not handled by the *keywords* and *flags* elements. These can be used to do special processing of certain keywords for which simply storing the argument into a variable is not good enough. After the *other-clauses* there will be an **otherwise** clause to complain about any undefined keywords found in *key-list*.

You can also use the **&key** lambda-list keyword to create functions that take keyword arguments. See the section "Functions: Evaluation".

prog *Special Form*

prog is a special form that 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
     statement3
     . . .
    )
```

The first subform of a **prog** is a list of variables, each of which may optionally have an initialization form. The first thing evaluation of a **prog** form does is to evaluate all of the *init* forms. Then each variable that had an *init* form is bound to its value, and the variables that did not have an *init* form are bound 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 in *parallel*; that is, all the initial values are computed before any of the variables are changed. **prog*** is the same as **prog** except that this initialization is sequential rather than parallel.

The part of a **prog** after the variable list is called the *body*. Each element of the body is either a symbol, in which case it is called a *tag*, or anything else (almost always a list), in which case it is called a *statement*.

After **prog** binds the 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*, where processing of the body is continued. *tag* is not evaluated. **return** and **go** and their variants are explained fully below.

The compiler requires that **go** and **return** forms be *lexically* within the scope of the **prog**; it is not possible for a function called from inside a **prog** body to **return** to the **prog**. That is, the **return** or **go** must be inside the **prog** itself, not inside a function called by the **prog**. (This restriction happens not to be enforced in the interpreter, but since all programs are eventually compiled, the convention should be adhered to. The restriction will be imposed in future implementations of the interpreter.)

See also the **do** special form, which uses a body similar to **prog**. The **do**, **catch**, and **throw** special forms are included in Zetalisp as an attempt to

encourage goto-less programming style, which often leads to more readable, more easily maintained code. You should use these forms instead of **prog** wherever reasonable.

If the first subform of a **prog** is a non-**nil** symbol (rather than a variable list), it is the name of the **prog**, and **return-from** can be used to return from it. See the special form **do-named**. 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))
```

prog, **do**, and their variants are effectively constructed out of **let**, **block**, and **tagbody** forms. **prog** could have been defined as the following macro (except for processing of local **declare**, which has been omitted for clarity):

```
(defmacro prog x
  (let ((block-name (and (symbolp (car x))
                         (neq (car x) nil)
                         (pop x)))
        (variables (car x))
        (tagbody (cdr x)))
    (if block-name
        '(block ,block-name
           (block nil
             (let ,variables
               (tagbody ,@tagbody))))
        '(block nil
           (let ,variables
             (tagbody ,@tagbody))))))
```

prog* *Special Form*

The **prog*** special form is almost the same as **prog**. The only difference is that the binding and initialization of the temporary variables is done *sequentially*, so each one can depend on the previous ones. For example:

```
(prog* ((y z) (x (car y)))
  (return x))
```

returns the car of the value of **z**.

A variant of **defun** that incorporates a **prog** into the function body is described elsewhere: See the macro **defunp**.

6. Nonlocal Exits

catch and **throw** are special forms used for nonlocal exits. **catch** evaluates forms; if a **throw** occurs during the evaluation, **catch** immediately returns (possibly multiple) values specified by **throw**.

catch and **throw** differ from **block** and **tagbody** in their scoping rules. **catch** and **throw** have dynamic scope; **block** and **tagbody** have lexical scope. See the section "Blocks and Exits".

***catch** and ***throw** are supported for compatibility with earlier releases. **catch** can be used with ***throw**, and ***catch** can be used with **throw**. If control exits normally, the returned values depend on whether **catch** or ***catch** is used. If control exits abnormally, the returned values depend on whether **throw** or ***throw** is used.

catch *tag body...* *Special Form*

catch is used with **throw** for nonlocal exits. **catch** first evaluates *tag* to obtain an object that is the "tag" of the catch. Then the *body* forms are evaluated in sequence, and **catch** returns the (possibly multiple) values of the last form in the body.

However, a **throw** or ***throw** form might be evaluated during the evaluation of one of the forms in *body*. In that case, if the throw "tag" is **eq** to the catch "tag" and if this **catch** is the innermost **catch** with that tag, the evaluation of the body is immediately aborted, and **catch** returns values specified by the **throw** or ***throw** form.

If the **catch** exits abnormally because of a **throw** form, it returns the (possibly multiple) values that result from evaluating **throw**'s second subform. If the **catch** exits abnormally because of a ***throw** form, it returns two values: the first is the result of evaluating ***throw**'s second subform, and the second is the result of evaluating ***throw**'s first subform (the tag thrown to).

On the LM-2 only, ***throw** and ***unwind-stack** cause the **catch** to return two additional values. If ***throw** is used, the third and fourth values are **nil**. If ***unwind-stack** is used, the third and fourth values are the third and fourth arguments to ***unwind-stack** (the active-frame-count and the action).

(catch 'foo *form*) catches a **(throw 'foo** *form*) but not a **(throw 'bar** *form*). It is an error if **throw** is done when no suitable **catch** exists.

The scope of the *tags* is dynamic. That is, the **throw** does not have to be lexically within the **catch** form; it is possible to throw out of a function that is called from inside a **catch** form.

On the LM-2 only, the values **t** and **nil** for *tag* are special: A **catch** whose tag is one of these values catches throws to any tag. These are for internal use only: **unwind-protect** uses **t**, and **catch-all** uses **nil**. 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, so it is an error if a specific catch for this tag does not exist higher up the stack. With **nil**, the error check is not done.

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**.

throw *tag form* *Special Form*

throw is used with **catch** to make nonlocal exits. It first evaluates *tag* to obtain an object that is the "tag" of the throw. It next evaluates *form* and saves the (possibly multiple) values. It then finds the innermost **catch** or ***catch** whose "tag" is **eq** to the "tag" that results from evaluating *tag*. It causes the **catch** or ***catch** to abort the evaluation of its body forms and to return all values that result from evaluating *form*. In the process, dynamic variable bindings are undone back to the point of the **catch**, and any **unwind-protect** cleanup forms are executed. An error is signalled if no suitable **catch** is found.

The scope of the *tags* is dynamic. That is, the **throw** does not have to be lexically within the **catch** form; it is possible to throw out of a function that is called from inside a **catch** form.

On the 3600, the value of *tag* cannot be the symbol **sys:unwind-protect-tag**; that is reserved for internal use. On the LM-2, the values **t**, **nil**, and **0** for *tag* are reserved for internal use. At present you cannot use **t** and **nil** for *tag* on the 3600; this will be changed in a future release.

unwind-protect *protected-form cleanup-form...* *Special Form*

Sometimes it is necessary to evaluate a form and make sure that certain side effects take place after the form is evaluated; a typical example is:

```
(progn
   (turn-on-water-faucet)
   (hairy-function 3 nil 'foo)
   (turn-off-water-faucet))
```

The nonlocal exit facility of Lisp creates a situation in which the above code will not work, however: if **hairy-function** should do a **throw** to a **catch** that is outside of the **progn** form, then **(turn-off-water-faucet)** will never be evaluated (and the faucet will presumably be left running). This is particularly likely if **hairy-function** gets an error and the user tells the Debugger to give up and flush the computation.

In order to allow the above program to work, it can be rewritten using **unwind-protect** as follows:

```
(unwind-protect
  (progn (turn-on-water-faucet)
         (hairy-function 3 nil 'foo))
  (turn-off-water-faucet))
```

If **hairy-function** does a **throw** that 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**.

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

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

protected-form is evaluated, and when it returns or when it attempts to quit out of the **unwind-protect**, the *cleanup-forms* are evaluated. **unwind-protect** catches exits caused by **return-from** or **go** as well as those caused by **throw**. The value of the **unwind-protect** is the value of *protected-form*. Multiple values returned by the *protected-form* are propagated back through the **unwind-protect**.

The cleanup forms are run in the variable-binding environment that you would expect: that is, variables bound outside the scope of the **unwind-protect** special form can be accessed, but variables bound inside the *protected-form* cannot be. In other words, the stack is unwound to the point just outside the *protected-form*, then the cleanup handler is run, and then the stack is unwound some more.

***catch** *tag body...* *Special Form*

***catch** is an obsolete version of **catch** that is supported for compatibility with earlier releases. It is equivalent to **catch** except that if ***catch** exits normally, it returns only two values: the first is the result of evaluating the last form in the body, and the second is **nil**. If ***catch** exits abnormally, it returns the same values as **catch** when **catch** exits abnormally: that is, the returned values depend on whether the exit results from a **throw** or a ***throw**. See the special form **catch**.

***throw** *tag form* *Function*

***throw** is an obsolete version of **throw** that is supported for compatibility with earlier releases. It is equivalent to **throw** except that it causes the **catch** or ***catch** to return only two values: the first is the result of evaluating *form*, and the second is the result of evaluating *tag* (the tag thrown to).

On the LM-2 only, ***throw** causes the **catch** or ***catch** to return two additional values. The third and fourth values are **nil**.

See the special form **throw**.

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

(LM-2 only) ***unwind-stack** is a generalization of ***throw** provided for program-manipulating programs such as the Debugger.

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

A *tag* of **t** invokes a special feature whereby the entire stack is unwound, and then the function *action* is called. During this process **unwind-protect**s receive control, but **catch-all**s do not. This feature is provided for the benefit of system programs that want to unwind a stack completely.

active-frame-count, if non-**nil**, is the number of frames to be unwound. The definition of a "frame" is implementation-dependent. If this counts down to zero before a suitable ***catch** is found, the ***unwind-stack** terminates and *that frame* returns *value* to whoever called it. This is similar to Maclisp's **freturn** function.

If *action* is non-**nil**, whenever the ***unwind-stack** would be ready to terminate (either due to *active-frame-count* or due to *tag* being caught as in ***throw**), instead *action* is called with one argument, *value*. If *tag* is **t**, meaning throw out the whole way, then the function *action* is not allowed to return. Otherwise the function *action* may return and its value will be returned instead of *value* from the ***catch** — or from an arbitrary function if *active-frame-count* is in use. In this case the ***catch** does not return multiple values as it normally does when thrown to. Note that it is often useful for *action* to be a stack group.

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

catch-all *body...* *Macro*

(LM-2 only) **(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 ***unwind-stack** with a tag of **t**. **catch-all** is a macro that expands into ***catch** with a *tag* of **nil**.

If you think you want this, most likely you are mistaken and you really want **unwind-protect**.

7. Mapping

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.

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*, and so on. The iteration stops as soon as any of the lists is exhausted. (If there are no lists at all, then there are no lists to be exhausted, so the function will be called repeatedly over and over. This is an obscure way to write an infinite loop. It is supported for consistency.) If you want to call a function of many arguments where one of the arguments successively takes on the values of the elements of a list and the other arguments are constant, you can use a circular list for the other arguments to **mapcar**. The function **circular-list** is useful for creating such lists. See the function **circular-list**.

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 a function, acceptable to **apply** — it cannot be a macro or the name of a special form.

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

		applies function to	
		successive sublists	successive elements
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 document *Packages*.

You can also do what the mapping functions do in a different way by using **loop**. See the section "The Loop Iteration Macro".

map *fcn* &rest *lists* *Function*

map is like **maplist**, except that it does not return any useful value. This function is used when the function is being called merely for its side effects, rather than its returned values. See the function **maplist**.

mapc *fcn* &rest *lists* *Function*

mapc is like **mapcar**, except that it does not return any useful value. This function is used when the function is being called merely for its side effects, rather than its returned values. See the function **mapcar**.

maplist *fcn* &rest *lists* *Function*

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. See the function **mapcar**.

mapcar *fcn* &rest *lists* *Function*

mapcar operates on successive elements of each list in *lists*. As it goes down the list, it calls *fcn*, giving it an element of the list as its one argument: first the **car**, then the **cadr**, then the **caddr**, and so on, 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)**.

mapcon *fcn* &rest *lists* *Function*

mapcon is like **maplist**, except that it combines the results of the function using **nconc** instead of **list**. See the function **maplist**. That is, **mapcon** could have been defined by:

```
(defun mapcon (f x y)
    (apply 'nconc (maplist f x y)))
```

Of course, this definition is less general than the real one.

mapcan *fcn* &rest *lists* *Function*

mapcan is like **mapcar**, except that it combines the results of the function using **nconc** instead of **list**. See the function **mapcar**.

8. The Loop Iteration Macro

8.1 Introduction

loop is a Lisp macro that provides a programmable iteration facility. The same **loop** module operates compatibly in Zetalisp, Maclisp (PDP-10 and Multics), and NIL. **loop** was inspired by the "FOR" facility of CLISP in Interlisp; however, it is not compatible and differs in several details.

The general approach is that a form introduced by the word **loop** generates a single program loop, into which a large variety of features can be incorporated. The loop consists of some initialization (*prologue*) code, a body that can be executed several times, and some exit (*epilogue*) code. Variables that can be declared local to the loop. The features are concerned with loop variables, deciding when to end the iteration, putting user-written code into the loop, returning a value from the construct, and iterating a variable through various real or virtual sets of values.

The **loop** form consists of a series of clauses, each introduced by a keyword symbol. Forms appearing in or implied by the clauses of a **loop** form are classed as those to be executed as initialization code, body code, and/or exit code; within each part of the template that **loop** fills in, they are executed strictly in the order implied by the original composition. Thus, just as in ordinary Lisp code, side effects may be used, and one piece of code may depend on following another for its proper operation. This is the principal philosophical difference from Interlisip's "FOR" facility.

Note that **loop** forms are intended to look like stylized English rather than Lisp code. There is a notably low density of parentheses, and many of the keywords are accepted in several synonymous forms to allow writing of more euphonious and grammatical English. Some find this notation verbose and distasteful, while others find it flexible and convenient. The former are invited to stick to **do**.

Here are some examples to illustrate the use of **loop**.

```
(defun print-elements-of-list (list-of-elements)
    (loop for element in list-of-elements
          do (print element)))
```

The above function prints each element in its argument, which should be a list. It returns **nil**.

```
(defun gather-alist-entries (list-of-pairs)
    (loop for pair in list-of-pairs
          collect (car pair)))
```

gather-alist-entries takes an association list and returns a list of the "keys"; that is, **(gather-alist-entries '((foo 1 2) (bar 259) (baz)))** returns **(foo bar baz)**.

```
(defun extract-interesting-numbers (start-value end-value)
   (loop for number from start-value to end-value
         when (interesting-p number) collect number))
```

The above function takes two arguments, which should be fixnums, and returns a list of all the numbers in that range (inclusive) that satisfy the predicate **interesting-p**.

```
(defun find-maximum-element (an-array)
   (loop for i from 0 below (array-dimension-n 1 an-array)
         maximize (aref an-array i)))
```

find-maximum-element returns the maximum of the elements of its argument, a one-dimensional array. For Maclisp, **aref** could be a macro that turns into either **funcall** or **arraycall** depending on what is known about the type of the array.

```
(defun my-remove (object list)
   (loop for element in list
         unless (equal object element) collect element))
```

my-remove is like the Lisp function **delete**, except that it copies the list rather than destructively splicing out elements. This is similar, although not identical, to the Zetalisp function **remove**.

```
(defun find-frob (list)
   (loop for element in list
         when (frobp element) return element
         finally (ferror nil "No frob found in the list ~S" list)))
```

This returns the first element of its list argument that satisfies the predicate **frobp**. If none is found, an error is generated.

8.2 Clauses

Internally, **loop** constructs a **prog** that includes variable bindings, pre-iteration (initialization) code, post-iteration (exit) code, the body of the iteration, and stepping of variables of iteration to their next values (which happens on every iteration after executing the body).

A *clause* consists of the keyword symbol and any Lisp forms and keywords with which it deals. For example:

```
 (loop for x
in l do (print x)),
```

contains two clauses, `"for x in l"` and `"do (print x)"`. Certain of the parts of the clause will be described as being *expressions*, such as **(print x)** in the example above.

An expression can be a single Lisp form, or a series of forms implicitly collected with **progn**. An expression is terminated by the next following atom, which is taken to be a keyword. This syntax allows only the first form in an expression to be atomic, but makes misspelled keywords more easily detectable.

loop uses print-name equality to compare keywords so that **loop** forms may be written without package prefixes; in Lisp implementations that do not have packages, **eq** is used for comparison.

Bindings and iteration variable steppings can be performed either sequentially or in parallel, which affects how the stepping of one iteration variable may depend on the value of another. The syntax for distinguishing the two will be described with the corresponding clauses. When a set of things is "in parallel", all of the bindings produced will be performed in parallel by a single lambda binding. Subsequent bindings will be performed inside of that binding environment.

8.2.1 Iteration-driving Clauses

These clauses all create a *variable of iteration*, which is bound locally to the loop and takes on a new value on each successive iteration. Note that if more than one iteration-driving clause is used in the same loop, several variables are created that all step together through their values; when any of the iterations terminates, the entire loop terminates. Nested iterations are not generated; for those, you need a second **loop** form in the body of the loop. In order to not produce strange interactions, iteration- driving clauses are required to precede any clauses that produce "body" code: that is, all except those that produce prologue or epilogue code (**initially** and **finally**), bindings (**with**), the **named** clause, and the iteration termination clauses (**while** and **until**).

Clauses that drive the iteration can be arranged to perform their testing and stepping either in series or in parallel. They are by default grouped in series, which allows the stepping computation of one clause to use the just-computed values of the iteration variables of previous clauses. They may be made to step "in parallel", as is the case with the **do** special form, by "joining" the iteration clauses with the keyword **and**. The form this typically takes is something like:

```
(loop ... for x = (f) and for y = init then (g x) ...)
```

which sets **x** to **(f)** on every iteration, and binds **y** to the value of *init* for the first iteration, and on every iteration thereafter sets it to **(g x)**, where **x** still has the value from the *previous* iteration. Thus, if the calls to **f** and **g** are not order-dependent, this would be best written as:

```
(loop ... for y = init then (g x) for x = (f) ...)
```

because, as a general rule, parallel stepping has more overhead than sequential stepping. Similarly, the example:

```
(loop for sublist on some-list
      and for previous = 'undefined then sublist
      ...)
```

which is equivalent to the **do** construct:

```
(do ((sublist some-list (cdr sublist))
     (previous 'undefined sublist))
    ((null sublist) ...)
  ...)
```

in terms of stepping, would be better written as:

```
(loop for previous = 'undefined then sublist
      for sublist on some-list
      ...)
```

When iteration-driving clauses are joined with **and**, if the token following the **and** is not a keyword that introduces an iteration-driving clause, it is assumed to be the same as the keyword that introduced the most recent clause; thus, the above example showing parallel stepping could have been written as:

```
(loop for sublist on some-list
      and previous = 'undefined then sublist
      ...)
```

The order of evaluation in iteration-driving clauses is that those expressions that are only evaluated once are evaluated in order at the beginning of the form, during the variable-binding phase, while those expressions that are evaluated each time around the loop are evaluated in order in the body.

One common and simple iteration-driving clause is **repeat**:

repeat *expression*
: This evaluates *expression* (during the variable-binding phase), and causes the **loop** to iterate that many times. *expression* is expected to evaluate to a fixnum. If *expression* evaluates to a 0 or negative result, the body code will not be executed.

All remaining iteration-driving clauses are subdispatches of the keyword **for**, which is synonymous with **as**. In all of them a *variable of iteration* is specified. Note that, in general, if an iteration-driving clause implicitly supplies an endtest, the value of this iteration variable as the loop is exited (that is, when the epilogue code is run) is undefined. See the section "The Iteration Framework".

Here are all of the varieties of **for** clauses. Optional parts are enclosed in curly brackets. See the section "Data Types: the Loop Iteration Macro". The *data-types* as used here are discussed fully in that section.

for *var* {*data-type*} **in** *expr1* {**by** *expr2*}
: This iterates over each of the elements in the list *expr1*. If the **by** subclause is present, *expr2* is evaluated once on entry to the loop to supply the function to be used to fetch successive sublists, instead of **cdr**.

for *var* {*data-type*} **on** *expr1* {**by** *expr2*}
This is like the previous **for** format, except that *var* is set to successive sublists of the list instead of successive elements. Note that since *var* will always be a list, it is not meaningful to specify a *data-type* unless *var* is a *destructuring pattern*, as described in the section on *destructuring*. Note also that **loop** uses a **null** rather than an **atom** test to implement both this and the preceding clause.

for *var* {*data-type*} = *expr*
On each iteration, *expr* is evaluated and *var* is set to the result.

for *var* {*data-type*} = *expr1* **then** *expr2*
var is bound to *expr1* when the loop is entered, and set to *expr2* (reevaluated) at all but the first iteration. Since *expr1* is evaluated during the binding phase, it cannot reference other iteration variables set before it; for that, use the following:

for *var* {*data-type*} **first** *expr1* **then** *expr2*
This sets *var* to *expr1* on the first iteration, and to *expr2* (reevaluated) on each succeeding iteration. The evaluation of both expressions is performed *inside* of the **loop** binding environment, before the **loop** body. This allows the first value of *var* to come from the first value of some other iteration variable, allowing such constructs as:

```
(loop for term in poly
      for ans first (car term) then (gcd ans (car term))
      finally (return ans))
```

for *var* {*data-type*} **from** *expr1* {**to** *expr2*} {**by** *expr3*}
This performs numeric iteration. *var* is initialized to *expr1*, and on each succeeding iteration is incremented by *expr3* (default **1**). If the **to** phrase is given, the iteration terminates when *var* becomes greater than *expr2*. Each of the expressions is evaluated only once, and the **to** and **by** phrases may be written in either order. **downto** may be used instead of **to**, in which case *var* is decremented by the step value, and the endtest is adjusted accordingly. If **below** is used instead of **to**, or **above** instead of **downto**, the iteration will be terminated before *expr2* is reached, rather than after. Note that the **to** variant appropriate for the direction of stepping must be used for the endtest to be formed correctly; that is, the code will not work if *expr3* is negative or 0. If no limit-specifying clause is given, then the direction of the stepping may be specified as being decreasing by using **downfrom** instead of **from**. **upfrom** may also be used instead of **from**; it forces the stepping direction to be increasing. The *data-type* defaults to **fixnum**.

for *var* {*data-type*} **being** *expr* **and its** *path* ...

for *var* {*data-type*} **being** {**each**|**the**} *path* ...
This provides a user-definable iteration facility. *path* names the manner in which the iteration is to be performed. The ellipsis indicates where various path-dependent preposition/expression pairs may appear. See the section "Iteration Paths".

8.2.2 Bindings

The **with** keyword may be used to establish initial bindings, that is, variables that are local to the loop but are only set once, rather than on each iteration. The **with** clause looks like:

with *var1* {*data-type*} {= *expr1*}
{**and** *var2* {*data-type*} {= *expr2*}}...

If no *expr* is given, the variable is initialized to the appropriate value for its data type, usually **nil**. **with** bindings linked by **and** are performed in parallel; those not linked are performed sequentially. That is:

```
(loop with a = (foo) and b = (bar) and c
      ...)
```

binds the variables like:

```
((lambda (a b c) ...)
 (foo) (bar) nil)
```

whereas:

```
(loop with a = (foo) with b = (bar a) with c ...)
```

binds the variables like:

```
((lambda (a)
    ((lambda (b)
        ((lambda (c) ...)
         nil))
     (bar a)))
 (foo))
```

All *expr*'s in **with** clauses are evaluated in the order they are written, in lambda-expressions surrounding the generated **prog**. The **loop** expression:

```
(loop with a = xa and b = xb
      with c = xc
      for d = xd then (f d)
        and e = xe then (g e d)
      for p in xp
      with q = xq
      ...)
```

produces the following binding contour, where **t1** is a **loop**-generated temporary:

```
((lambda (a b)
   ((lambda (c)
      ((lambda (d e)
         ((lambda (p t1)
            ((lambda (q) ...)
             xq))
          nil xp))
       xd xe))
    xc))
 xa xb)
```

Because all expressions in **with** clauses are evaluated during the variable-binding phase, they are best placed near the front of the **loop** form for stylistic reasons.

For binding more than one variable with no particular initialization, one may use the construct:

with *variable-list* {*data-type-list*} {**and** ...}

as in:

```
with (i j k t1 t2) (fixnum fixnum fixnum) ...
```

A slightly shorter way of writing this is:

```
with (i j k) fixnum and (t1 t2) ...
```

These are cases of *destructuring* which **loop** handles specially. See the section "Data Types: the Loop Iteration Macro". See the section "Destructuring".

Occasionally there are various implementational reasons for a variable *not* to be given a local type declaration. If this is necessary, the **nodeclare** clause may be used:

nodeclare *variable-list*

The variables in *variable-list* are noted by **loop** as not requiring local type declarations. Consider the following:

```
(declare (special k) (fixnum k))
(defun foo (l)
    (loop for x in l as k fixnum = (f x) ...))
```

If **k** did not have the **fixnum** data-type keyword given for it, then **loop** would bind it to **nil**, and some compilers would complain. On the other hand, the **fixnum** keyword also produces a local **fixnum** declaration for **k**; since **k** is special, some compilers will complain (or error out). The solution is to do:

```
(defun foo (l)
    (loop nodeclare (k)
          for x in l as k fixnum = (f x) ...))
```

which tells **loop** not to make that local declaration. The **nodeclare** clause must come *before* any reference to the variables so noted. Positioning it incorrectly will cause this clause to not take effect, and may not be diagnosed.

8.2.3 Entrance and Exit

initially *expression*
This puts *expression* into the *prologue* of the iteration. It will be evaluated before any other initialization code other than the initial bindings. For the sake of good style, the **initially** clause should therefore be placed after any **with** clauses but before the main body of the loop.

finally *expression*
This puts *expression* into the *epilogue* of the loop, which is evaluated when the iteration terminates (other than by an explicit **return**). For stylistic reasons, then, this clause should appear last in the loop body. Note that certain clauses may generate code that terminates the iteration without running the epilogue code; this behavior is noted with those clauses. See the section "Aggregated Boolean Tests". This clause may be used to cause the loop to return values in a nonstandard way:

```
(loop for n in l
      sum n into the-sum
      count t into the-count
      finally (return (quotient the-sum the-count)))
```

8.2.4 Side Effects

do *expression*

doing *expression*
expression is evaluated each time through the loop, as shown in the **print-elements-of-list** example. See the section "Introduction: the Loop Iteration Macro".

8.2.5 Values

The following clauses accumulate a return value for the iteration in some manner. The general form is:

type-of-collection expr {*data-type*} {**into** *var*}

where *type-of-collection* is a **loop** keyword, and *expr* is the thing being "accumulated" somehow. If no **into** is specified, then the accumulation will be returned when the **loop** terminates. If there is an **into**, then when the epilogue of the **loop** is reached, *var* (a variable automatically bound locally in the loop) will have been set to the accumulated result and may be used by the epilogue code. In this way, a user may accumulate and somehow pass back multiple values from a single **loop**, or use them during the loop. It is safe to reference these variables during the loop, but they should not be modified until the epilogue code of the loop is reached. For example:

```
(loop for x in list
      collect (foo x) into foo-list
      collect (bar x) into bar-list
      collect (baz x) into baz-list
     finally (return (list foo-list bar-list baz-list)))
```

has the same effect as:

```
(do ((g0001 list (cdr g0001))
     (x) (foo-list) (bar-list) (baz-list))
    ((null g0001)
     (list (nreverse foo-list)
           (nreverse bar-list)
           (nreverse baz-list)))
  (setq x (car g0001))
  (setq foo-list (cons (foo x) foo-list))
  (setq bar-list (cons (bar x) bar-list))
  (setq baz-list (cons (baz x) baz-list)))
```

except that **loop** arranges to form the lists in the correct order, obviating the **nreverse**s at the end, and allowing the lists to be examined during the computation.

collect *expr* {**into** *var*}

collecting ...

This causes the values of *expr* on each iteration to be collected into a list.

nconc *expr* {**into** *var*}

nconcing ...

append ...

appending ...

These are like **collect**, but the results are **nconc**ed or **append**ed together as appropriate.

```
(loop for i from 1 to 3
      nconc (list i (* i i)))
  => (1 1 2 4 3 9)
```

count *expr* {**into** *var*} {*data-type*}

counting ...

If *expr* evaluates non-**nil**, a counter is incremented. The *data-type* defaults to **fixnum**.

sum *expr* {*data-type*} {**into** *var*}

summing ...

Evaluates *expr* on each iteration, and accumulates the sum of all the values. *data-type* defaults to **number**, which for all practical purposes is **notype**. Note that specifying *data-type* implies that *both* the sum and the number being summed (the value of *expr*) will be of that type.

maximize *expr* {*data-type*} {**into** *var*}

minimize ...

Computes the maximum (or minimum) of *expr* over all iterations. *data-type* defaults to **number**. Note that if the loop iterates zero times, or if conditionalization prevents the code of this clause from being executed, the result will be meaningless. If **loop** can determine that the arithmetic being performed is not contagious (by virtue of *data-type* being **fixnum**, **flonum**, or **small-flonum**), then it may choose to code this by doing an arithmetic comparison rather than calling either **max** or **min**. As with the **sum** clause, specifying *data-type* implies that both the result of the **max** or **min** operation and the value being maximized or minimized will be of that type.

Not only can there be multiple *accumulations* in a **loop**, but a single *accumulation* can come from multiple places *within the same* **loop** *form*. Obviously, the types of the collection must be compatible. **collect**, **nconc**, and **append** may all be mixed, as may **sum** and **count**, and **maximize** and **minimize**. For example:

```
(loop for x in '(a b c) for y in '((1 2) (3 4) (5 6))
      collect x
      append y)
  => (a 1 2 b 3 4 c 5 6)
```

The following computes the average of the entries in the list *list-of-frobs*:

```
(loop for x in list-of-frobs
      count t into count-var
      sum x into sum-var
    finally (return (quotient sum-var count-var)))
```

8.2.6 Endtests

The following clauses may be used to provide additional control over when the iteration gets terminated, possibly causing exit code (due to **finally**) to be performed and possibly returning a value (for example, from **collect**).

while *expr*

If *expr* evaluates to **nil**, the loop is exited, performing exit code (if any), and returning any accumulated value. The test is placed in the body of the loop where it is written. It can appear between sequential **for** clauses.

until *expr*

Identical to **while (not** *expr*).

This may be needed, for example, to step through a strange data structure, as in:

```
(loop until (top-of-concept-tree? concept)
      for concept = expr then (superior-concept concept)
      ...)
```

Note that the placement of the **until** clause before the **for** clause is valid in this

case because of the definition of this particular variant of **for**, which *binds* **concept** to its first value rather than setting it from inside the **loop**.

The following may also be of use in terminating the iteration:

loop-finish *Macro*

(loop-finish) causes the iteration to terminate "normally", the same as implicit termination by an iteration-driving clause, or by the use of **while** or **until** — the epilogue code (if any) will be run, and any implicitly collected result will be returned as the value of the **loop**. For example:

```
(loop for x in '(1 2 3 4 5 6)
      collect x
      do (cond ((= x 4) (loop-finish))))
=> (1 2 3 4)
```

This particular example would be better written as **until (= x 4)** in place of the **do** clause.

8.2.7 Aggregated Boolean Tests

All of these clauses perform some test, and may immediately terminate the iteration depending on the result of that test.

always *expr*

Causes the loop to return **t** if *expr* **always** evaluates non-**null**. If *expr* evaluates to **nil**, the loop immediately returns **nil**, without running the epilogue code (if any, as specified with the **finally** clause); otherwise, **t** will be returned when the loop finishes, after the epilogue code has been run.

never *expr*

Causes the loop to return **t** if *expr* **never** evaluates non-**null**. This is equivalent to **always (not** *expr*).

thereis *expr*

If *expr* evaluates non-**nil**, then the iteration is terminated and that value is returned, without running the epilogue code.

8.2.8 Conditionalization

These clauses may be used to "conditionalize" the following clause. They may precede any of the side-effecting or value-producing clauses, such as **do, collect, always**, or **return**.

when *expr*

if *expr*

If *expr* evaluates to **nil**, the following clause will be skipped, otherwise not.

unless *expr*

This is equivalent to **when (not** *expr*)).

Multiple conditionalization clauses may appear in sequence. If one test fails, then any following tests in the immediate sequence, and the clause being conditionalized, are skipped.

Multiple clauses may be conditionalized under the same test by joining them with **and**, as in:

```
(loop for i from a to b
      when (zerop (remainder i 3))
        collect i and do (print i))
```

which returns a list of all multiples of **3** from **a** to **b** (inclusive) and prints them as they are being collected.

If-then-else conditionals may be written using the **else** keyword, as in:

```
(loop for i from a to b
      when (oddp i)
        collect i into odd-numbers
      else collect i into even-numbers)
```

Multiple clauses may appear in an **else**-phrase, using **and** to join them in the same way as above.

Conditionals may be nested. For example:

```
(loop for i from a to b
      when (zerop (remainder i 3))
        do (print i)
        and when (zerop (remainder i 2))
              collect i)
```

returns a list of all multiples of **6** from **a** to **b**, and prints all multiples of **3** from **a** to **b**.

When **else** is used with nested conditionals, the "dangling else" ambiguity is resolved by matching the **else** with the innermost **when** not already matched with an **else**. Here is a complicated example.

```
(loop for x in l
      when (atom x)
        when (memq x *distinguished-symbols*)
          do (process1 x)
        else do (process2 x)
      else when (memq (car x) *special-prefixes*)
             collect (process3 (car x) (cdr x))
             and do (memorize x)
           else do (process4 x))
```

Useful with the conditionalization clauses is the **return** clause, which causes an explicit return of its "argument" as the value of the iteration, bypassing any epilogue code. That is:

when *expr1* **return** *expr2*

is equivalent to:

when *expr1* **do (return** *expr2***)**

Conditionalization of one of the "aggregated boolean value" clauses simply causes the test that would cause the iteration to terminate early not to be performed unless the condition succeeds. For example:

```
(loop for x in l
      when (significant-p x)
        do (print x) (princ "is significant.")
        and thereis (extra-special-significant-p x))
```

does not make the **extra-special-significant-p** check unless the **significant-p** check succeeds.
The format of a conditionalized clause is typically something like:

when *expr1 keyword expr2*

If *expr2* is the keyword **it**, then a variable is generated to hold the value of *expr1*, and that variable gets substituted for *expr2*. Thus, the composition:

when *expr* **return it**

is equivalent to the clause:

thereis *expr*

and one may collect all non-null values in an iteration by saying:

when *expression* **collect it**

If multiple clauses are joined with **and**, the **it** keyword may only be used in the first. If multiple **whens**, **unlesses**, and/or **ifs** occur in sequence, the value substituted for **it** will be that of the last test performed. The **it** keyword is not recognized in an **else**-phrase.

8.2.9 Miscellaneous Other Clauses

named *name*

This gives the **prog** that **loop** generates a name of *name*, so that one may use the **return-from** form to return explicitly out of that particular **loop**:

```
(loop named sue
      ...
      do (loop ... do (return-from sue value) ...)
      ...)
```

The **return-from** form shown causes *value* to be immediately returned as the value of the outer **loop**. Only one name may be given to any particular **loop** construct. This feature does not exist in the Maclisp version of **loop**, since Maclisp does not support "named progs".

return *expression*

Immediately returns the value of *expression* as the value of the loop, without running the epilogue code. This is most useful with some sort of conditionalization, as discussed in the previous section. Unlike most of the

other clauses, **return** is not considered to "generate body code", so it is allowed to occur between iteration clauses, as in:

```
(loop for entry in list
      when (not (numberp entry))
        return (error ...)
      as frob = (times entry 2)
      ...)
```

If you instead desire the loop to have some return value when it finishes normally, you may place a call to the **return** function in the epilogue (with the **finally** clause). See the section "Entrance and Exit".

8.3 Loop Synonyms

define-loop-macro *keyword* *Macro*

May be used to make *keyword*, a **loop** keyword (such as **for**), into a Lisp macro which may introduce a **loop** form. For example, after evaluating:

```
(define-loop-macro for),
```

you can now write an iteration as:

```
(for i from 1 below n do ...)
```

This facility exists primarily for diehard users of a predecessor of **loop**. Its unconstrained use is not recommended, as it tends to decrease the transportability of the code and needlessly uses up a function name.

8.4 Data Types

In many of the clause descriptions, an optional *data-type* is shown. A *data-type* in this sense is an atomic symbol, and is recognizable as such by **loop**. These are used for declaration and initialization purposes; for example, in:

```
(loop for x in l
      maximize x flonum into the-max
      sum x flonum into the-sum
      ...)
```

the **flonum** data-type keyword for the **maximize** clause says that the result of the **max** operation, and its "argument" (**x**), will both be flonums; hence **loop** may choose to code this operation specially since it knows there can be no contagious arithmetic. The **flonum** data-type keyword for the **sum** clause behaves similarly, and in addition causes **the-sum** to be correctly initialized to **0.0** rather than **0**. The **flonum** keywords will also cause the variables **the-max** and **the-sum** to be declared to be **flonum**, in implementations where such a declaration exists. In general, a numeric

data-type more specific than **number**, whether explicitly specified or defaulted, is considered by **loop** to be license to generate code using type-specific arithmetic functions where reasonable. The following data-type keywords are recognized by **loop** (others may be defined; for that, consult the source code):

fixnum	An implementation-dependent limited-range integer.
flonum	An implementation-dependent limited-precision floating-number.
small-flonum	This is recognized in the Zetalisp implementation only, where its only significance is for initialization purposes, since no such declaration exists.
integer	Any integer (no range restriction).
number	Any number.
notype	Unspecified type (that is, anything else).

Note that explicit specification of a nonnumeric type for an operation that is numeric (such as the **summing** clause) may cause a variable to be initialized to **nil** when it should be **0**.

If local data-type declarations must be inhibited, you can use the **nodeclare** clause.

8.5 Destructuring

Destructuring provides you with the ability to "simultaneously" assign or bind multiple variables to components of some data structure. Typically this is used with list structure. For example:

```
(loop with (foo . bar) = '(a b c) ...)
```

has the effect of binding **foo** to **a** and **bar** to **(b c)**.

loop's destructuring support is intended to parallel if not augment that provided by the host Lisp implementation, with a goal of minimally providing destructuring over list structure patterns. Thus, in Lisp implementations with no system destructuring support at all, you can still use list-structure patterns as **loop** iteration variables, and in **with** bindings. In NIL, **loop** also supports destructuring over vectors.

You can specify the data-types of the components of a pattern by using a corresponding pattern of the data type keywords in place of a single data type keyword. This syntax remains unambiguous because wherever a data-type keyword is possible, a **loop** keyword is the only other possibility. Thus, if you want to do:

```
(loop for x in l
      as i fixnum = (car x)
      and j fixnum = (cadr x)
      and k fixnum = (cddr x)
      ...)
```

and no reference to **x** is needed, you may instead write:

```
(loop for (i j . k) (fixnum fixnum . fixnum) in l ...)
```

To allow some abbreviation of the data-type pattern, an atomic component of the data-type pattern is considered to state that all components of the corresponding part of the variable pattern are of that type. That is, the previous form could be written as:

```
(loop for (i j . k) fixnum in l ...)
```

This generality allows binding of multiple typed variables in a reasonably concise manner, as in:

```
(loop with (a b c) and (i j k) fixnum ...)
```

which binds **a**, **b**, and **c** to **nil** and **i**, **j**, and **k** to **0** for use as temporaries during the iteration, and declares **i**, **j**, and **k** to be fixnums for the benefit of the compiler.

```
(defun map-over-properties (fn symbol)
  (loop for (propname propval) on (plist symbol) by 'cddr
        do (funcall fn symbol propname propval)))
```

maps *fn* over the properties on *symbol*, giving it arguments of the symbol, the property name, and the value of that property.

In Lisp implementations where **loop** performs its own destructuring, notably Multics Maclisp and Zetalisp, you can cause **loop** to use already provided destructuring support instead:

si:loop-use-system-destructuring? *Variable*

This variable exists *only* in **loop** implementations in Lisps that do not provide destructuring support in the default environment. It is by default **nil**. If changed, then **loop** will behave as it does in Lisps that *do* provide destructuring support: destructuring binding will be performed using **let**, and destructuring assignment will be performed using **desetq**. Presumably, if your personalized environment supplies these macros, then you should set this variable to **t**; there is, however, little (if any) efficiency loss if this is not done.

8.6 The Iteration Framework

This section describes the way **loop** constructs iterations. It is necessary if you will be writing your own iteration paths, and may be useful in clarifying what **loop** does with its input.

loop considers the act of *stepping* to have four possible parts. Each iteration-driving clause has some or all of these four parts, which are executed in this order:

pre-step-endtest
: This is an endtest that determines if it is safe to step to the next value of the iteration variable.

steps
: Variables that get "stepped". This is internally manipulated as a list of the form (*var1 val1 var2 val2* ...); all of those variables are stepped in parallel, meaning that all of the *val*s are evaluated before any of the *var*s are set.

post-step-endtest
: Sometimes you cannot see if you are done until you step to the next value; that is, the endtest is a function of the stepped-to value.

pseudo-steps
: Other things that need to be stepped. This is typically used for internal variables that are more conveniently stepped here, or to set up iteration variables that are functions of some internal variable(s) that are actually driving the iteration. This is a list like *steps*, but the variables in it do not get stepped in parallel.

The above alone is actually insufficient in just about all iteration- driving clauses that **loop** handles. What is missing is that in most cases, the stepping and testing for the first time through the loop is different from that of all other times. So, what **loop** deals with is two four-tuples as above; one for the first iteration, and one for the rest. The first may be thought of as describing code that immediately precedes the loop in the **prog**, and the second as following the body code — in fact, **loop** does just this, but severely perturbs it in order to reduce code duplication. Two lists of forms are constructed in parallel: one is the first-iteration endtests and steps, the other the remaining-iterations endtests and steps. These lists have dummy entries in them so that identical expressions will appear in the same position in both. When **loop** is done parsing all of the clauses, these lists get merged back together such that corresponding identical expressions in both lists are not duplicated unless they are "simple" and it is worth doing.

Thus, one *may* get some duplicated code if one has multiple iterations. Alternatively, **loop** may decide to use and test a flag variable that indicates whether one iteration has been performed. In general, sequential iterations have less overhead than parallel iterations, both from the inherent overhead of stepping multiple variables in parallel, and from the standpoint of potential code duplication.

Note also that although the user iteration variables are guaranteed to be stepped in parallel, the placement of the endtest for any particular iteration may be either before or after the stepping. A notable case of this is:

```
(loop for i from 1 to 3 and dummy = (print 'foo)
      collect i)
  => (1 2 3)
```

but prints **foo** *four* times. Certain other constructs, such as **for** *var* **on**, may or may not do this depending on the particular construction.

This problem also means that it might not be safe to examine an iteration variable in the epilogue of the loop form. As a general rule, if an iteration driving clause implicitly supplies an endtest, then you cannot know the state of the iteration variable when the loop terminates. Although you can guess on the basis of whether the iteration variable itself holds the data upon which the endtest is based, that guess *might* be wrong. Thus:

```
(loop for subl on expr
      ...
      finally (f subl))
```

is incorrect, but:

```
(loop as frob = expr while (g frob)
      ...
      finally (f frob))
```

is safe because the endtest is explicitly dissociated from the stepping.

8.7 Iteration Paths

Iteration paths provide a mechanism for user extension of iteration-driving clauses. The interface is constrained so that the definition of a path need not depend on much of the internals of **loop**. The typical form of an iteration path is

```
for var {data-type} being {each|the} pathname {preposition1 expr1}...
```

pathname is an atomic symbol that is defined as a **loop** path function. The usage and defaulting of *data-type* is up to the path function. Any number of preposition/expression pairs may be present; the prepositions allowable for any particular path are defined by that path. For example:

```
(loop for x being the array-elements of my-array from 1 to 10
      ...)
```

To enhance readability, pathnames are usually defined in both the singular and plural forms; this particular example could have been written as:

```
(loop for x being each array-element of my-array from 1 to 10
      ...)
```

Another format, which is not so generally applicable, is:

`for` *var* {*data-type*} `being` *expr0* `and its` *pathname* {*preposition1 expr1*}...

In this format, *var* takes on the value of *expr0* the first time through the loop. Support for this format is usually limited to paths that step through some data structure, such as the "superiors" of something. Thus, we can hypothesize the **cdrs** path, such that:

```
(loop for x being the cdrs of '(a b c . d) collect x)
 => ((b c . d) (c . d) d)
```

but:

```
(loop for x being '(a b c . d) and its cdrs collect x)
 => ((a b c . d) (b c . d) (c . d) d)
```

his, **her**, or **their** may be substituted for the **its** keyword, as may **each**. Egocentricity is not condoned. See the section "Predefined Paths". Some example uses of iteration paths are shown in that section.

Very often, iteration paths step internal variables that the you do not specify, such as an index into some data structure. Although in most cases the user does not wish to be concerned with such low-level matters, it is occasionally useful to have a handle on such things. **loop** provides an additional syntax with which you can provide a variable name to be used as an "internal" variable by an iteration path, with the **using** "prepositional phrase".

The **using** phrase is placed with the other phrases associated with the path, and contains any number of keyword/variable-name pairs:

```
(loop for x being the array-elements of a using (index i)
      ...)
```

which says that the variable **i** should be used to hold the index of the array being stepped through. The particular keywords that may be used are defined by the iteration path; the **index** keyword is recognized by all **loop** sequence paths. See the section "Sequence Iteration". Note that any individual **using** phrase applies to only one path; it is parsed along with the "prepositional phrases". It is an error if the path does not call for a variable using that keyword.

By special dispensation, if a *pathname* is not recognized, then the **default-loop-path** path will be invoked upon a syntactic transformation of the original input. Essentially, the **loop** fragment:

`for` *var* `being` *frob*

is taken as if it were:

`for` *var* `being default-loop-path in` *frob*

and:

`for` *var* `being` *expr* `and its` *frob* ...

is taken as if it were:

`for` *var* `being` *expr* `and its default-loop-path in` *frob*

Thus, this "undefined pathname hook" only works if the **default-loop-path** path is defined. Obviously, the use of this "hook" is competitive, since only one such hook may be in use, and the potential for syntactic ambiguity exists if *frob* is the name of a defined iteration path. This feature is not for casual use; it is intended for use by large systems that wish to use a special syntax for some feature they provide.

8.7.1 Loop Iteration Over Hash Tables

A new iteration path was added to **loop** to support iterating over every entry in a hash table.

```
(loop for x being the hash-elements of new-coms ...)
(loop for x being the hash-elements of new-coms with-key k ...)
```

This provides for *x* to take on the values of successive values of hash table entries. The loop runs once for every entry of the hash table. *x* could have the same value more than once, since it is the key that is unique, not the value.

The **with-key** phrase is optional. It provides for the variable *k* to have the hash key for the particular hash entry value *x* that you are examining.

8.7.2 Predefined Paths

loop comes with two predefined iteration path functions; one implements a **mapatoms**-like iteration path facility, and the other is used for defining iteration paths for stepping through sequences.

8.7.2.1 The interned-symbols Path

The **interned-symbols** iteration path is like a **mapatoms** for **loop**.

```
(loop for sym being interned-symbols ...)
```

iterates over all of the symbols in the current package and its superiors (or, in Maclisp, the current obarray). This is the same set of symbols that **mapatoms** iterates over, although not necessarily in the same order. The particular package to look in may be specified as in:

```
(loop for sym being the interned-symbols in package ...)
```

which is like giving a second argument to **mapatoms**.

In Lisp implementations such as Zetalisp with some sort of hierarchical package structure, you can restrict the iteration to be over just the package specified and not its superiors, by using the **local-interned-symbols** path:

```
(loop for sym being the local-interned-symbols {in package}
      ...)
```

Example:

```
(defun my-apropos (sub-string &optional (pkg package))
    (loop for x being the interned-symbols in pkg
          when (string-search sub-string x)
            when (or (boundp x) (fboundp x) (plist x))
              do (print-interesting-info x)))
```

In the Zetalisp and NIL implementations of **loop**, a package specified with the **in** preposition may be anything acceptable to the **pkg-find-package** function. The code generated by this path will contain calls to internal **loop** functions, with the effect that it will be transparent to changes to the implementation of packages. In the Maclisp implementation, the obarray *must* be an array pointer, *not* a symbol with an **array** property.

8.7.2.2 Sequence Iteration

One very common form of iteration is that over the elements of some object that is accessible by means of an integer index. **loop** defines an iteration path function for doing this in a general way, and provides a simple interface to allow users to define iteration paths for various kinds of "indexable" data.

define-loop-sequence-path *path-name-or-names fetchfun sizefun* *Macro*
&optional *sequence-type element-type*

path-name-or-names is either an atomic path name or list of path names. *fetchfun* is a function of two arguments: the sequence, and the index of the item to be fetched. (Indexing is assumed to be zero-origined.) *sizefun* is a function of one argument, the sequence; it should return the number of elements in the sequence. *sequence-type* is the name of the data-type of the sequence, and *element-type* the name of the data-type of the elements of the sequence. These last two items are optional.

The Zetalisp implementation of **loop** utilizes the Zetalisp array manipulation primitives to define both **array-element** and **array-elements** as iteration paths:

```
(define-loop-sequence-path (array-element array-elements)
    aref array-active-length)
```

Then, the **loop** clause:

```
for var being the array-elements of array
```

will step *var* over the elements of *array*, starting from **0**. The sequence path function also accepts **in** as a synonym for **of**.

The range and stepping of the iteration may be specified with the use of all the same keywords that are accepted by the **loop** arithmetic stepper (**for** *var* **from** ...); they are **by**, **to**, **downto**, **from**, **downfrom**, **below**, and **above**, and are interpreted in the same manner. Thus:

```
(loop for var being the array-elements of array
      from 1 by 2
     ...)
```

steps *var* over all of the odd elements of *array*, and:

```
(loop for var being the array-elements of array
      downto 0
     ...)
```

steps in "reverse" order.

```
(define-loop-sequence-path (vector-elements vector-element)
    vref vector-length notype notype)
```

is how the **vector-elements** iteration path can be defined in NIL (which it is). One can then do such things as:

```
(defun cons-a-lot (item &restv other-items)
    (and other-items
         (loop for x being the vector-elements of other-items
               collect (cons item x))))
```

All such sequence iteration paths allow you to specify the variable to be used as the index variable, by use of the **index** keyword with the **using** prepositional phrase. See the section "Iteration Paths".

8.7.3 Defining Paths

This section and the next might not be of interest to those not interested in defining their own iteration paths.

A **loop** iteration clause (for example, a **for** or **as** clause) produces, in addition to the code that defines the iteration, variables that must be bound, and pre-iteration (*prologue*) code. See the section "The Iteration Framework". This breakdown allows a user interface to **loop** that does not have to depend on or know about the internals of **loop**. To complete this separation, the iteration path mechanism parses the clause before giving it to the user function that will return those items. A function to generate code for a path can be declared to **loop** with the **define-loop-path** function:

define-loop-path *Macro*

```
(define-loop-path pathname-or-names path-function
    list-of-allowable-prepositions
    datum-1 datum-2 ...)
```

This defines *path-function* to be the handler for the path(s) *pathname-or-names*, which may be either a symbol or a list of symbols. Such a handler should follow the conventions described below. The *datum-i* are optional; they are passed in to *path-function* as a list.

The handler will be called with the following arguments:

path-name The name of the path that caused the path function to be invoked.

variable The "iteration variable".

data-type The data type supplied with the iteration variable, or **nil** if none was supplied.

prepositional-phrases
This is a list with entries of the form *(preposition expression)*, in the order in which they were collected. This may also include some supplied implicitly (for example, an **of** phrase when the iteration is inclusive, and an **in** phrase for the **default-loop-path** path); the ordering will show the order of evaluation which should be followed for the expressions.

inclusive? This is **t** if *variable* should have the starting point of the path as its value on the first iteration (by virtue of being specified with syntax like **for** *var* **being** *expr* **and its** *pathname*), **nil** otherwise. When **t**, *expr* will appear in *prepositional-phrases* with the **of** preposition; for example, **for x being foo and its cdrs** gets *prepositional-phrases* of **((of foo))**.

allowed-prepositions
This is the list of allowable prepositions declared for the pathname that caused the path function to be invoked. It and *data* may be used by the path function such that a single function may handle similar paths.

data This is the list of "data" declared for the pathname that caused the path function to be invoked. It may, for instance, contain a canonicalized pathname, or a set of functions or flags to aid the path function in determining what to do. In this way, the same path function may be able to handle different paths.

The handler should return a list of either six or ten elements:

variable-bindings
This is a list of variables that need to be bound. The entries in it may be of the form *variable*, (*variable expression*), or (*variable expression data-type*). Note that it is the responsibility of the handler to make sure the iteration variable gets bound. All of these variables will be bound in parallel; if initialization of one depends on others, it should be done with a **setq** in the *prologue-forms*. Returning only the variable without any initialization expression is not allowed if the variable is a destructuring pattern.

prologue-forms
This is a list of forms that should be included in the **loop** prologue.

the four items of the iteration specification
These are the four items: *pre-step-endtest*, *steps*, *post-step-endtest*, and *pseudo-steps*. See the section "The Iteration Framework".

another four items of iteration specification
If these four items are given, they apply to the first iteration, and the previous four apply to all succeeding iterations; otherwise, the previous four apply to *all* iterations.

Here are the routines that are used by **loop** to compare keywords for equality. In all cases, a *token* may be any Lisp object, but a *keyword* is expected to be an atomic symbol. In certain implementations these functions may be implemented as macros.

si:loop-tequal *token keyword* *Function*
This is the **loop** token comparison function. *token* is any Lisp object; *keyword* is the keyword it is to be compared against. It returns **t** if they represent the same token, comparing in a manner appropriate for the implementation.

si:loop-tmember *token keyword-list* *Function*
The **member** variant of **si:loop-tequal**.

si:loop-tassoc *token keyword-alist* *Function*
The **assoc** variant of **si:loop-tequal**.

If an iteration path function desires to make an internal variable accessible to the user, it should call the following function instead of **gensym**:

si:loop-named-variable *keyword* *Function*
This should only be called from within an iteration path function. If *keyword* has been specified in a **using** phrase for this path, the corresponding variable is returned; otherwise, **gensym** is called and that new symbol returned. Within a given path function, this routine should only be called once for any given keyword.

If you specify a **using** preposition containing any keywords for which the path function does not call **si:loop-named-variable**, **loop** will inform you of the error.

8.7.3.1 An Example Path Definition

Here is an example function that defines the **string-characters** iteration path. This path steps a variable through all of the characters of a string. It accepts the format:

```
(loop for var being the string-characters of str ...)
```

The function is defined to handle the path by:

```
(define-loop-path string-characters string-chars-path
     (of))
```

Here is the function:

```
(defun string-chars-path (path-name variable data-type
                          prep-phrases inclusive?
                          allowed-prepositions data
                          &aux (bindings nil)
                               (prologue nil)
                               (string-var (gensym))
                               (index-var (gensym))
                               (size-var (gensym)))
  allowed-prepositions data ; unused variables
  ; To iterate over the characters of a string, we need
  ; to save the string, save the size of the string,
  ; step an index variable through that range, setting
  ; the user's variable to the character at that index.
  ; Default the data-type of the user's variable:
  (cond ((null data-type) (setq data-type 'fixnum)))
  ; We support exactly one "preposition", which is
  ; required, so this check suffices:
  (cond ((null prep-phrases)
         (ferror nil "OF missing in ~S iteration path of ~S"
                 path-name variable)))
  ; We do not support "inclusive" iteration:
  (cond ((not (null inclusive?))
         (ferror nil
           "Inclusive stepping not supported in ~S path ~
            of ~S (prep phrases = ~:S)"
           path-name variable prep-phrases)))
  ; Set up the bindings
  (setq bindings (list (list variable nil data-type)
                       (list string-var (cadar prep-phrases))
                       (list index-var 0 'fixnum)
                       (list size-var 0 'fixnum)))
  ; Now set the size variable
  (setq prologue (list `(setq ,size-var (string-length
                                          ,string-var))))
  ; and return the appropriate stuff, explained below.
  (list bindings
        prologue
        `(= ,index-var ,size-var)
        nil
        nil
       ; char-n is the NIL string referencing primitive.
       ; In Zetalisp, aref could be used instead.
        (list variable `(char-n ,string-var ,index-var)
              index-var `(1+ ,index-var))))
```

The first element of the returned list is the bindings. The second is a list of forms to be placed in the *prologue*. The remaining elements specify how the iteration is to be performed. This example is a particularly simple case, for two reasons: the actual "variable of iteration", **index-var**, is purely internal (being **gensym**med), and the stepping of it (1+) is such that it may be performed safely without an endtest. Thus **index-var** may be stepped immediately after the setting of the user's variable, causing the iteration specification for the first iteration to be identical to the iteration specification for all remaining iterations. This is advantageous from the standpoint of the optimizations **loop** is able to perform, although it is frequently not possible due to the semantics of the iteration (for example, **for** *var* **first** *expr1* **then** *expr2*) or to subtleties of the stepping. It is safe for this path to step the user's variable in the *pseudo-steps* (the fourth item of an iteration specification) rather than the "real" steps (the second), because the step value can have no dependencies on any other (user) iteration variables. Using the pseudo-steps generally results in some efficiency gains.

If you wanted the index variable in the above definition to be user-accessible through the **using** phrase feature with the **index** keyword, the function would need to be changed in two ways. First, **index-var** should be bound to **(si:loop-named-variable 'index)** instead of **(gensym)**. Secondly, the efficiency hack of stepping the index variable ahead of the iteration variable must not be done. This is effected by changing the last form to be:

```
(list bindings prologue
      nil
      (list index-var '(1+ ,index-var))
      '(= ,index-var ,size-var)
      (list variable '(char-n ,string-var ,index-var))
      nil
      nil
      '(= ,index-var ,size-var)
      (list variable '(char-n ,string-var ,index-var)))
```

Note that although the second **'(= ,index-var ,size-var)** could have been placed earlier (where the second **nil** is), it is best for it to match up with the equivalent test in the first iteration specification grouping.

Index

D **D** **D**

E

F

K K K

L

L

L

M

M

M

N

O

P

R

S

T

T T

U

U U

V

V V

W W W

ARR Arrays and Strings

Arrays and Strings
 # 990047

March 1984

This document corresponds to Release 5.0.

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

No representation or affirmation of fact contained in this document should be construed as a warranty by Symbolics, and its contents are subject to change without notice. Symbolics, Inc. assumes no responsibility for any errors that might appear in this document.

Symbolics software described in this document is furnished only under license, and may be used only in accordance with the terms of such license. Title to, and ownership of, such software shall at all times remain in Symbolics, Inc. Nothing contained herein implies the granting of a license to make, use, or sell any Symbolics equipment or software.

Symbolics is a trademark of Symbolics, Inc., Cambridge, Massachusetts.

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

Table of Contents

1. Arrays

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

The *dimensionality* of an array (or, the number of dimensions that 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 0; the highest value is a property of the array. Each dimension has a size, which is the lowest number that 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 0, 1, 2, 3, and 4.

The most basic primitive functions for handling arrays are:

- **make-array** — used for the creation of arrays
- **aref** — used for examining the contents of arrays
- **aset** — used for storing into arrays

An array is a regular Lisp object, and it is common for an array to be the binding of a symbol, or the car or cdr of a cons, or, in fact, an element of an array. There are many functions, described in this chapter, that take arrays as arguments and perform useful operations on them.

Another way of handling arrays, inherited from Maclisp, is to treat them as functions. In this case each array has a name, which is a symbol whose function definition is the array. Zetalisp supports this style by allowing an array to be *applied* to arguments, as if it were a function. The arguments are treated as subscripts and the array is referenced appropriately. The **store** special form is also supported, but it is supported on the LM-2 only. See the special form **store**. This kind of array referencing is considered to be obsolete, and is slower than the usual kind. It should not be used in new programs.

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 or flonums. The array types are known by a set of symbols whose names begin with "**art-**" (for ARray Type).

The most commonly used type is called **art-q**. An **art-q** array simply holds Lisp objects of any type.

Similar to the **art-q** type is the **art-q-list**. Like the **art-q**, its elements may be any Lisp object. The difference is that the **art-q-list** array "doubles" as a list; the function **g-l-p** takes an **art-q-list** array and returns a list whose elements are those

of the array, and whose actual substance is that of the array. If you **rplaca** elements of the list, the corresponding element of the array changes, and if you store into the array, the corresponding element of the list changes the same way. An attempt to **rplacd** the list causes an error, since arrays cannot implement that operation.

There is a set of types called **art-1b, art-2b, art-4b, art-8b**, and **art-16b**; these names are short for "1 bit", "2 bits", and so on. Each element of an **art-*n*b** array is a nonnegative fixnum, and only the least significant *n* bits are remembered in the array; all of the others are discarded. Thus **art-1b** arrays store only 0 and 1, and if you store a 5 into an **art-2b** array and look at it later, you will find a 1 rather than a 5.

These arrays are used when it is known beforehand that the fixnums that will be stored are nonnegative and limited in size to a certain number of bits. Their advantage over the **art-q** array is that they occupy less storage, because more than one element of the array is kept in a single machine word. (For example, 32 elements of an **art-1b** array or 2 elements of an **art-16b** array will fit into one word).

There are also **art-32b** arrays that have 32 bits per element. Since fixnums only have 24 bits anyway, these are the same as **art-q** arrays except that they only hold fixnums. They do not behave consistently with the other "bit" array types, and generally they should not be used.

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*. See the section "Strings". That section deals with functions that manipulate these type of arrays.

An **art-fat-string** array is a character string with wider characters, containing 16 bits rather than 8 bits. The extra bits are ignored by string operations, such as comparison, on these strings; typically they are used to hold font information.

An **art-half-fix** array contains half-size fixnums. Each element of the array is a signed 16-bit integer; the range is from -32768 to 32767 inclusive.

The **art-float** array type is a special-purpose type whose elements are flonums. When storing into such an array the value (any kind of number) will be converted to a flonum, using the **float** function. The advantage of storing flonums in an **art-float** array rather than an **art-q** array is that the numbers in an **art-float** array are not true Lisp objects. Instead the array remembers the numerical value, and when it is **aref**'ed creates a Lisp object (a flonum) to hold the value. Because the system does special storage management for bignums and flonums that are intermediate results, the use of **art-float** arrays can save a lot of work for the garbage collector and hence greatly increase performance. An intermediate result is

a Lisp object passed as an argument, stored in a local variable, or returned as the value of a function, but not stored into a global variable, a non-**art-float** array, or list structure. **art-float** arrays also provide a locality of reference advantage over **art-q** arrays containing flonums, since the flonums are contained in the array rather than being separate objects probably on different pages of memory.

The **art-fps-float** array type is another special-purpose type whose elements are flonums. The internal format of this array is compatible with the PDP-11/VAX single-precision floating-point format. The primary purpose of this array type is to interface with the FPS array processor, which can transfer data directly in and out of such an array.

When storing into an **art-fps-float** array any kind of number may be stored. It will be rounded off to the 24-bit precision of the PDP-11. If the magnitude of the number is too large, the largest valid floating-point number will be stored. If the magnitude is too small, 0 will be stored.

When reading from an **art-fps-float** array, a new flonum is created containing the value, just as with an **art-float** array.

There are three types of arrays that exist only for the implementation of *stack groups*; these types are called **art-stack-group-head, art-special-pdl**, and **art-reg-pdl**. Their elements may be any Lisp object. See the section "Stack Groups".

Currently, multidimensional arrays are stored in column-major order rather than row-major order as in Maclisp. Row-major order means that successive memory locations differ in the last subscript, while column-major order means that successive memory locations differ in the first subscript. This has an effect on paging performance when using large arrays; if you want to reference every element in a multidimensional array and move linearly through memory to improve locality of reference, you must vary the first subscript fastest rather than the last.

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. The values of these symbols are internal array type code numbers for the corresponding type.

array-types *array-type-code* *Function*

Given an internal numeric array-type code, returns the symbolic name of that type.

array-elements-per-q *Variable*

array-elements-per-q is an association list that associates each array type symbol with the number of array elements stored in one word, for an array of that type. See the section "Association Lists". If the value is negative, it is instead the number of words per array element, for arrays whose elements are more than one word long.

array-elements-per-q *array-type-code* *Function*

Given the internal array-type code number, returns the number of array elements stored in one word, for an array of that type. If the value is negative, it is instead the number of words per array element, for arrays whose elements are more than one word long.

array-bits-per-element *Variable*

The value of **array-bits-per-element** is an association list that 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. See the section "Association Lists".

array-bits-per-element *array-type-code* *Function*

Given the internal array-type code numbers, returns the number of bits per cell for unsigned numeric arrays, or **nil** for a type of array that can contain Lisp objects.

array-element-size *array* *Function*

Given an array, returns the number of bits that fit in an element of that array. For arrays that can hold general Lisp objects, the result is 24., assuming you will be storing unsigned fixnums in the array.

1.1 Extra Features of Arrays

Any array can have an *array leader*. An array leader is like a one-dimensional **art-q** array that is attached to the main array. So an array that has a leader acts like two arrays joined together. The leader can be stored into and examined by a special set of functions, different from those used for the main array: **array-leader** and **store-array-leader**. The leader is always one-dimensional, and always can hold any kind of Lisp object, regardless of the type or dimensionality of the main part of the array.

Very often the main part of an array will be a homogeneous set of objects, while the leader will be used to remember a few associated nonhomogeneous pieces of data. In this case the leader is not used like an array; each slot is used differently from the others. Explicit numeric subscripts should not be used for the leader elements of such an array; instead the leader should be described by a **defstruct**. See the macro **defstruct**.

By convention, element zero 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*. Many array-processing functions recognize the fill pointer. For instance, if a string (an array of type **art-string**) has seven elements, but its fill pointer contains the value 5, 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, and so on.

The system does not provide a way to turn off the fill-pointer convention; any array that has a leader must reserve element 0 for the fill pointer or avoid using many of the array functions.

Leader element one is used in conjunction with the "named structure" feature to associate a "data type" with the array. See the section "Named Structures". Element one is only treated specially if the array is flagged as a named structure.

Normally, an array is represented as a small amount of header information, followed by the contents of the array. However, sometimes it is desirable to have the header information removed from the actual contents. One such occasion is when the contents of the array must be located in a special part of the Lisp Machine's address space, such as the area used for the control of input/output devices, or the bitmap memory that generates the TV image. Displaced arrays are also used to reference certain special system tables, which are at fixed addresses so the microcode can access them easily.

If you give **make-array** a fixnum or a locative as the value of the **:displaced-to** option, it will create a displaced array referring to that location of virtual memory and its successors.

References to elements of the displaced array will access that part of storage, and return the contents; the regular **aref** and **aset** functions are used. If the array is one whose elements are Lisp objects, caution should be used: if the region of address space does not contain typed Lisp objects, the integrity of the storage system and the garbage collector could be damaged. If the array is one whose elements are bytes (such as an **art-4b** type), then there is no problem. It is important to know, in this case, that the elements of such arrays are allocated from the right to the left within the 32-bit words.

It is also possible to have an array whose contents, instead of being located at a fixed place in virtual memory, are defined to be those of another array. Such an array is called an *indirect array*, and is created by giving **make-array** an array as the value of the **:displaced-to** option. The effects of this are simple if both arrays have the same type; the two arrays share all elements. An object stored in a certain element of one can be retrieved from the corresponding element of the other. This, by itself, is not very useful. However, if the arrays have different dimensionality, the manner of accessing the elements differs. Thus, by creating a one-dimensional array of nine elements that 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. Unexpected effects can be produced if the new array is of a different type than the old array; this is not generally recommended. Indirecting an **art-*m*b** array to an **art-*n*b** array will do the "obvious" thing. For instance, if *m* is 4 and *n* is 1, each element of the first array will contain four bits from the second array, in right-to-left order.

It is also possible to create an indirect array in such a way that when an attempt is made to reference it or store into it, a constant number is added to the subscript given. This number is called the *index-offset*, and is specified at the time the indirect array is created, by giving a fixnum to **make-array** as the value of the **:displaced-index-offset** option. Similarly, the length of the indirect array need not be the full length of the array it indirects to; it can be smaller. The **nsubstring** function creates such arrays. When using index offsets with multidimensional arrays, there is only one index offset; it is added in to the "linearized" subscript which is the result of multiplying each subscript by an appropriate coefficient and adding them together.

Conformal Indirection

Multidimensional arrays on the 3600 remember their actual dimensions, separately from the magic numbers by which to multiply the subscripts before adding them together to get the index into the array.

As a result of this, multidimensional indirect arrays on the 3600 can have *conformal indirection*. If A is indirected to B, and they do not have the same width, then normally the part of B that is shared with A does not have the same shape as A. If conformal indirection is used, then it does have the same shape and there are gaps between the rows of A. For example:

```
(setq b (make-array '(10. 20.)))
(setq a (make-array '(3 5) ':displaced-to b ':displaced-index-offset 12.))
```

Now:

```
(aref a 1 0) = (aref b 3 1) and (aref a 1 1) = (aref b 6 1).
```

In contrast:

```
(setq a (make-array '(3 5) ':displaced-to b ':displaced-index-offset 12.
                           ':displaced-conformally t))
```

(aref a 1 0) = (aref b 3 1) still, but (aref a 1 1) = (aref b 3 2). Each row of A corresponds to part of a row of B, always starting at the same column (2).

A graphic illustration:

```
(setq a (make-array '(6 20.))
      b (make-array '(3 5) ':displaced-to a ':displaced-index-offset 22.)
      c (make-array '(3 5) ':displaced-to a ':displaced-index-offset 22.
                           ':displaced-conformally t))
```

```
  Normal case                    Conformal case
  0                   19         0                   19
  +--------------------+         +--------------------+
 0|aaaaaaaaaaaaaaaaaaaa|        0|aaaaaaaaaaaaaaaaaaaa|
  |aaBBBBBBBBBBBBBBBBaaa|        |aaCCCCCaaaaaaaaaaaaa|
  |aaaaaaaaaaaaaaaaaaaa|        |aaCCCCCaaaaaaaaaaaaa|
  |aaaaaaaaaaaaaaaaaaaa|        |aaCCCCCaaaaaaaaaaaaa|
  |aaaaaaaaaaaaaaaaaaaa|        |aaaaaaaaaaaaaaaaaaaa|
 5|aaaaaaaaaaaaaaaaaaaa|       5|aaaaaaaaaaaaaaaaaaaa|
  +--------------------+         +--------------------+
```

Arrays are stored in column-major order, so the units in which the index-offset is measured should be read first from left to right and then from top to bottom.

The meaning of **adjust-array-size** for conformal indirect arrays is undefined.

1.2 Basic Array Functions

make-array *dimensions* &rest *options*. *Function*

This is the primitive function for making arrays. *dimensions* should be a list of fixnums that are the dimensions of the array; the length of the list will be the dimensionality of the array. For convenience when making a one-dimensional array, the single dimension may be provided as a fixnum rather than a list of one fixnum.

options are alternating keywords and values. The keywords may be any of the following:

:area The value specifies in which area the array should be created. It should be either an area number (a fixnum), or **nil** to mean the default area. See the section "Areas".

:type The value should be a symbolic name of an array type; the most common of these is **art-q**, which is the default. The elements of the array are initialized according to the type: if the array is of a type whose elements may only be fixnums or flonums, then every element of the array will initially be 0 or 0.0; otherwise, every element will initially be **nil**. See the section "Arrays: Arrays and Strings". Array types are described in that section. The value of the option may also be the value of a symbol that is an array type name (that is, an internal numeric array type code).

:displaced-to

If this is not **nil**, then the array will be a *displaced* array. If the value is a fixnum or a locative, **make-array** will create a regular displaced array that refers to the specified section of virtual address space. If the value is an array, **make-array** will create an indirect array. See the section "Extra Features of Arrays".

:initial-value
This makes its value the initial value of every element of the array. Example:

```
(make-array 5 ':type 'art-string ':initial-value #/a)
                    => "aaaaa"
```

:leader-length
The value should be a fixnum. The array will have a leader with that many elements. The elements of the leader will be initialized to **nil** unless the **:leader-list** option is given.

:leader-list
The value should be a list. Call the number of elements in the list *n*. The first *n* elements of the leader will be initialized from successive elements of this list. If the **:leader-length** option is not specified, then the length of the leader will be *n*. If the **:leader-length** option is given, and its value is greater than *n*, then the *n*th and following leader elements will be initialized to **nil**. If its value is less than *n*, an error is signalled. The leader elements are filled in forward order; that is, the **car** of the list will be stored in leader element **0**, the **cadr** in element **1**, and so on.

:fill-pointer
It causes **make-array** to give the array a fill pointer and initializes it to the value following the keyword. Use this instead of **:leader-length** or **:leader-list** when you are using the leader only for a fill pointer. This keyword is compatible with the current Common Lisp design, which has no array leaders.

:displaced-index-offset
If this is present, the value of the **:displaced-to** option should be an array, and the value should be a nonnegative fixnum; it is made to be the index-offset of the created indirect array. See the section "Extra Features of Arrays".

:displaced-conformally
(3600 only) Can be used with the **:displaced-to** option. If the value is **t** and **make-array** is creating an indirect array, the array uses conformal indirection.

:named-structure-symbol
If this is not **nil**, it is a symbol to be stored in the named-structure cell of the array. The array will be tagged as a named structure. See the section "Named Structures". If the array has a leader, then

this symbol will be stored in leader element 1 regardless of the value of the **:leader-list** option. If the array does not have a leader, then this symbol will be stored in array element zero.

Examples:

```
;; Create a one-dimensional array of five elements.
(make-array 5)
;; Create a two-dimensional array,
;; three by four, with four-bit elements.
(make-array '(3 4) ':type 'art-4b)
;; Create an array with a three-element leader.
(make-array 5 ':leader-length 3)
;; Create an array with a leader, providing
;; initial values for the leader elements.
(setq a (make-array 100 ':type 'art-1b
                        ':leader-list '(t nil)))
(array-leader a 0) => t
(array-leader a 1) => nil

;; Create a named-structure with five leader
;; elements, initializing some of them.
(setq b (make-array 20 ':leader-length 5
                       ':leader-list '(0 nil foo)
                       ':named-structure-symbol 'bar))
(array-leader b 0) => 0
(array-leader b 1) => bar
(array-leader b 2) => foo
(array-leader b 3) => nil
(array-leader b 4) => nil
```

make-array returns the newly created array, and also returns, as a second value, the number of words allocated in the process of creating the array, that is, the **%structure-total-size** of the array.

When **make-array** was originally implemented, it took its arguments in the following fixed pattern:

```
(make-array area type dimensions
            &optional displaced-to leader
                      displaced-index-offset
                      named-structure-symbol)
```

leader was a combination of the **:leader-length** and **:leader-list** options, and the list was in reverse order.

This form is obsolete and should not be used. The compiler warns about uses of this form of **make-array**; however, it continues to accept the obsolete form.

aref *array* &rest *subscripts* *Function*

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* *Function*

This is an obsolete version of **aref** that only works for one-dimensional arrays. There is no reason ever to use it.

ar-2 *array i j* *Function*

This is an obsolete version of **aref** that only works for two-dimensional arrays. There is no reason ever to use it.

ar-3 *array i j k* *Function*

This is an obsolete version of **aref** that only works for three-dimensional arrays. There is no reason ever to use it. **ar-3** is available only on the LM-2.

aset *x array* &rest *subscripts* *Function*

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*. The returned value is *x*.

as-1 *x array i* *Function*

This is an obsolete version of **aset** that only works for one-dimensional arrays. There is no reason ever to use it.

as-2 *x array i j* *Function*

This is an obsolete version of **aset** that only works for two-dimensional arrays. There is no reason ever to use it.

as-3 *x array i j k* *Function*

This is an obsolete version of **aset** that only works for three-dimensional arrays. There is no reason ever to use it. **as-3** is available only on the LM-2.

aloc *array* &rest *subscripts* *Function*

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*. See the section "Locatives".

ap-1 *array i* *Function*

This is an obsolete version of **aloc** that only works for one-dimensional arrays. There is no reason ever to use it.

ap-2 *array i j* *Function*

This is an obsolete version of **aloc** that only works for two-dimensional arrays. There is no reason ever to use it.

ap-3 *array i j k* *Function*

This is an obsolete version of **aloc** that only works for three-dimensional arrays. There is no reason ever to use it. **ap-3** is available only on the LM-2.

The compiler turns **aref** into **ar-1**, **ar-2**, and so on according to the number of subscripts specified, turns **aset** into **as-1**, **as-2**, and so on, and turns **aloc** into **ap-1**, **ap-2**, and so on. For arrays with more than three dimensions the compiler uses the slightly less efficient form since the special routines only exist for one, two, and three dimensions. There is no reason for any program to call **ar-1**, **as-1**, **ar-2**, and so forth explicitly; they are documented because there used to be such a reason, and many old programs use these functions. New programs should use **aref**, **aset**, and **aloc**.

A related function, provided only for Maclisp compatibility, is **arraycall**.

array-leader *array i* *Function*

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* *Function*

array should be an array with a leader, and *i* should be a fixnum. *x* can 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* *Function*

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 section "Locatives". This is analogous to **aloc**.

fill-pointer *array* *Function*

Returns the value of the fill pointer. *array* must have a fill pointer. **fill-pointer** is actually a subst, so it compiles inline instead of as a function call. **setf** can be used on a **fill-pointer** form to set the value of the fill pointer.

Programs access the fill pointer by explicitly asking for the zeroth element of the array leader.

1.3 Getting Information About an Array

array-type *array* *Function*

Returns the symbolic type of *array*. Example:

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

array-length *array* *Function*

array may be any array. This returns the total number of elements in *array*. For a one-dimensional array, this is one greater than the maximum allowable subscript. (But if fill pointers are being used, you may want to use **array-active-length**.) Example:

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

array-active-length *array* *Function*

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 section "Extra Features of Arrays". A general explanation of the use of fill pointers is in that section.

array-#-dims *array* *Function*

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 read your program in Maclisp. (It does not need to be slashified for the Zetalisp reader, which is smarter.) Example:

```
(array-#-dims (make-array '(3 5))) => 2
```

array-dimension-n *n array* *Function*

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**, this returns the length of the leader of *array*; if *array* has no leader it returns **nil**. If *n* is any other value, this returns **nil**. Examples:

```
(setq a (make-array '(3 5) ':leader-length 7))
(array-dimension-n 1 a) => 3
(array-dimension-n 2 a) => 5
(array-dimension-n 3 a) => nil
(array-dimension-n 0 a) => 7
```

array-dimensions *array* *Function*

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

```
(setq a (make-array '(3 5)))
(array-dimensions a) => (3 5)
```

Note: the list returned by **(array-dimensions** *x***)** is equal to the cdr of the list returned by **(arraydims** *x***)**.

arraydims *array* *Function*

array may be any array; it also may be a symbol whose function cell contains an array, for Maclisp compatibility. See the section "Maclisp Array Compatibility". **arraydims** 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 '(3 5)))
(arraydims a) => (art-q 3 5)
```

array-in-bounds-p *array* &rest *subscripts* *Function*
This function checks whether *subscripts* is a legal set of subscripts for *array*, and returns **t** if they are; otherwise it returns **nil**.

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

array-indirect-p *array* *Function*
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* *Function*
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**.

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

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

1.4 Changing the Size of an Array

adjust-array-size *array* *new-size* *Function*
If *array* is a one-dimensional array, its size is changed to be *new-size*. If *array* has more than one dimension, its size (**array-length**) is changed to *new-size* by changing only the last dimension.

If *array* is made smaller, the extra elements are lost; if *array* is made bigger, the new elements are initialized in the same fashion as **make-array** would initialize them: either to **nil** or **0**, depending on the type of array. Example:

```
(setq a (make-array 5))
(aset 'foo a 4)
(aref a 4) => foo
(adjust-array-size a 2)
(aref a 4) => an error occurs
```

If the size of the array is being increased, **adjust-array-size** may have to allocate a new array somewhere. In that case, it alters *array* so that references to it will be made to the new array instead, by means of "invisible pointers". See the function **structure-forward**. **adjust-array-size** will return this new array if it creates one, and otherwise it will return *array*. Be careful to be consistent about using the returned result of **adjust-array-size**, because you may end up holding two arrays that are not the same (that is, not **eq**), but that share the same contents.

The meaning of **adjust-array-size** for conformal indirect arrays is undefined.

array-grow *array* &rest *dimensions* *Function*

array-grow creates a new array of the same type as *array*, with the specified dimensions. Those elements of *array* that are still in bounds are copied into the new array. The elements of the new array that are not in the bounds of *array* are initialized to **nil** or **0** as appropriate. If *array* has a leader, the new array will have a copy of it. **array-grow** returns the new array and also forwards *array* to it, like **adjust-array-size**.

Unlike **adjust-array-size**, **array-grow** always creates a new array rather than growing or shrinking the array in place. But **array-grow** of a multidimensional array can change all the subscripts and move the elements around in memory to keep each element at the same logical place in the array.

return-array *array* *Function*

This peculiar function attempts to return *array* to free storage. If it is displaced, this returns the displaced array itself, not the data that the array points to. Currently **return-array** does nothing if the array is not at the end of its region, that is, if it was not the most recently allocated nonlist object in its area. This will eventually be renamed to **reclaim**, when it works for other objects than arrays.

If you still have any references to *array* anywhere in the Lisp world after this function returns, the garbage collector can get a fatal error if it sees them. Since the form that calls this function must get the array from somewhere, it may not be clear how to legally call **return-array**. One of the only ways to do it is as follows:

```
(defun func ()
  (let ((array (make-array 100)))
    ...
    (return-array (prog1 array (setq array nil)))))
```

so that the variable **array** does not refer to the array when **return-array** is called. You should only call this function if you know what you are doing; otherwise the garbage collector can get fatal errors. Be careful.

1.5 Arrays Overlaid with Lists

These functions manipulate **art-q-list** arrays. See the section "Arrays: Arrays and Strings".

g-l-p *array* *Function*

array should be an **art-q-list** array. This returns a list that shares the storage of *array*. Example:

```
(setq a (make-array 4 ':type 'art-q-list))
(aref a 0) => nil
(setq b (g-l-p a)) => (nil nil nil nil)
(rplaca b t)
b => (t nil nil nil)
(aref a 0) => t
(aset 30 a 2)
b => (t nil 30 nil)
```

The following two functions work strangely, in the same way that **store** does, and should not be used in new programs.

get-list-pointer-into-array *array-ref* *Function*

The argument *array-ref* is ignored, but should be a reference to an **art-q-list** array by applying the array to subscripts (rather than by **aref**). This returns a list object which is a portion of the "list" of the array, beginning with the last element of the last array which has been called as a function. **get-list-pointer-into-array** is available only on the LM-2.

get-locative-pointer-into-array *array-ref* *Function*

get-locative-pointer-into-array is similar to **get-list-pointer-into-array**, except that it returns a locative, and does not require the array to be **art-q-list**. Use **aloc** instead of this function in new programs. **get-locative-pointer-into-array** is available only on the LM-2.

1.6 Adding to the End of an Array

array-push *array x* *Function*

array must be a one-dimensional array that 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, that is, the array index in which it stored *x*. If the array is of type **art-q-list**, an operation similar to **nconc**

has taken place, in that the element has been added to the list by changing the cdr of the formerly last element. The cdr coding is updated to ensure this.

array-push-extend *array x* &optional *extension* *Function*

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; that is, it never "fails" the way **array-push** does, and so never returns **nil**. *extension* is the number of elements to be added to the array if it needs to be grown. It defaults to something reasonable, based on the size of the array.

array-pop *array* *Function*

array must be a one-dimensional array that has a fill pointer. The fill pointer is decreased by one, and the array element designated by the new value of the fill pointer is returned. If the new value does not designate any element of the array (specifically, if it had already reached zero), an error is caused. The two operations (decrementing and array referencing) happen uninterruptibly. If the array is of type **art-q-list**, an operation similar to **nbutlast** has taken place. The cdr coding is updated to ensure this.

1.7 Copying an Array

fillarray *array source* *Function*

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 *source*.

If *source* is a list, then **fillarray** fills up *array* with the elements of *list*. If *source* is too short to fill up all of *array*, then the last element of *source* is used to fill the remaining elements of *array*. If *source* is too long, the extra elements are ignored. If *source* is **nil** (the empty list), *array* is filled with the default initial value for its array type (**nil** or **0**).

If *source* 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 *source*. If *source* is too small, then the extra elements of *array* are not affected.

If *array* is multidimensional, the elements are accessed in row-major order: the last subscript varies the most quickly. The same is true of *source* if it is an array.

fillarray returns *array*.

listarray *array* &optional *limit* *Function*

array may be any type of array, or, for Maclisp compatibility, a symbol whose function cell contains an array. **listarray** creates and returns a list whose elements are those of *array*. If *limit* is present, it should be a fixnum, and only the first *limit* (if there are more than that many) elements of *array* are used, and so the maximum length of the returned list is *limit*.

If *array* is multidimensional, the elements are accessed in row-major order: the last subscript varies the most quickly.

list-array-leader *array* &optional *limit* *Function*

array may be any type of array, or, for Maclisp compatibility, a symbol whose function cell contains an array. **list-array-leader** creates and returns a list whose elements are those of *array*'s leader. If *limit* is present, it should be a fixnum, and only the first *limit* (if there are more than that many) elements of *array*'s leader are used, and so the maximum length of the returned list is *limit*. If *array* has no leader, **nil** is returned.

copy-array-contents *from-array to-array* *Function*

from and *to* must be arrays. The contents of *from* is copied into the contents of *to*, element by element. If *to* is shorter than *from*, the rest of *from* is ignored. If *from* is shorter than *to*, the rest of *to* is filled with **nil** if it is a q-type array, or 0 if it is a numeric array or a string, or 0.0 if it is a flonum array. This function always returns **t**.

Note that even if *from* or *to* has a leader, the whole array is used; the convention that leader element 0 is the "active" length of the array is not used by this function. The leader itself is not copied.

copy-array-contents works on multidimensional arrays. *from* and *to* are "linearized" subscripts, and column-major order is used, *that is, the first subscript varies fastest (opposite from* **fillarray***).*

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

This is just like **copy-array-contents**, but the leader of *from* (if any) is also copied into *to*. **copy-array-contents** copies only the main part of the array.

copy-array-portion *from-array from-start from-end to-array to-start to-end* *Function*

The portion of the array *from-array* with indices greater than or equal to *from-start* and less than *from-end* is copied into the portion of the array *to-array* with indices greater than or equal to *to-start* and less than *to-end*, element by element. If there are more elements in the selected portion of *to-array* than in the selected portion of *from-array*, the extra elements are filled with the default value as by **copy-array-contents**. If there are more elements in the selected portion of *from-array*, the extra ones are ignored. Multidimensional arrays are treated the same way as **copy-array-contents** treats them. This function always returns **t**.

Currently, **copy-array-portion** (as well as **copy-array-contents** and **copy-array-contents-and-leader**) copies one element at a time in increasing order of subscripts (this behavior might change in the future). This means that when copying from and to the same array, the results might be unexpected if *from-start* is less than *to-start*. You can safely copy from and to the same array as long as *from-start* $\geq$ *to-start*.

bitblt *alu width height from-array from-x from-y to-array to-x to-y* *Function*

from-array and *to-array* must be two-dimensional arrays of bits or bytes (**art-1b**, **art-2b**, **art-4b**, **art-8b**, **art-16b**, or **art-32b**). **bitblt** copies a rectangular portion of *from-array* into a rectangular portion of *to-array*. The value stored can be a Boolean function of the new value and the value already there, under the control of *alu*. This function is most commonly used in connection with raster images for TV displays.

The top-left corner of the source rectangle is (**aref** *from-array from-x from-y*). The top-left corner of the destination rectangle is (**aref** *to-array to-x to-y*). *width* and *height* are the dimensions of both rectangles. If *width* or *height* is zero, **bitblt** does nothing.

from-array and *to-array* are allowed to be the same array. **bitblt** normally traverses the arrays in increasing order of x and y subscripts. If *width* is negative, then (**abs** *width*) is used as the width, but the processing of the x direction is done backwards, starting with the highest value of x and working down. If *height* is negative it is treated analogously. When **bitblt**ing an array to itself, when the two rectangles overlap, it may be necessary to work backwards to achieve the desired effect, such as shifting the entire array upwards by a certain number of rows. Note that negativity of *width* or *height* does not affect the *(x,y)* coordinates specified by the arguments, which are still the top-left corner even if **bitblt** starts at some other corner.

If the two arrays are of different types, **bitblt** works bit-wise and not element-wise. That is, if you **bitblt** from an **art-2b** array into an **art-4b** array, then two elements of the *from-array* will correspond to one element of the *to-array*. *width* is in units of elements of the *to-array*.

If **bitblt** goes outside the bounds of the source array, it wraps around. This allows such operations as the replication of a small stipple pattern through a large array. If **bitblt** goes outside the bounds of the destination array, it signals an error.

If *src* is an element of the source rectangle, and *dst* is the corresponding element of the destination rectangle, then **bitblt** changes the value of *dst* to (**boole** *alu src dst*). See the **boole** function. The following are the symbolic names for some of the most useful *alu* functions:

tv:alu-seta plain copy

tv:alu-ior	inclusive or
tv:alu-xor	exclusive or
tv:alu-andca	and with complement of source

bitblt is written in highly optimized microcode and goes very much faster than the same thing written with ordinary **aref** and **aset** operations would. Unfortunately this causes **bitblt** to have a couple of strange restrictions. Wraparound does not work correctly if *from-array* is an indirect array with an index-offset. **bitblt** will signal an error if the first dimensions of *from-array* and *to-array* are not both integral multiples of the machine word length. For **art-1b** arrays, the first dimension must be a multiple of 32., for **art-2b** arrays it must be a multiple of 16., and so on.

1.8 Matrices and Systems of Linear Equations

The functions in this section perform some useful matrix operations. The matrices are represented as two-dimensional Lisp arrays. These functions are part of the mathematics package rather than the kernel array system, hence the "**math:**" in the names.

math:multiply-matrices *matrix-1 matrix-2* &optional *matrix-3* *Function*
Multiplies *matrix-1* by *matrix-2*. If *matrix-3* is supplied, **multiply-matrices** stores the results into *matrix-3* and returns *matrix-3*; otherwise it creates an array to contain the answer and returns that. All matrices must be two-dimensional arrays, and the first dimension of *matrix-2* must equal the second dimension of *matrix-1*.

math:invert-matrix *matrix* &optional *into-matrix* *Function*
Computes the inverse of *matrix*. If *into-matrix* is supplied, stores the result into it and returns it; otherwise it creates an array to hold the result, and returns that. *matrix* must be two-dimensional and square. The Gauss-Jordan algorithm with partial pivoting is used. Note: if you want to solve a set of simultaneous equations, you should not use this function; use **math:decompose** and **math:solve**.

math:transpose-matrix *matrix* &optional *into-matrix* *Function*
Transposes *matrix*. If *into-matrix* is supplied, stores the result into it and returns it; otherwise it creates an array to hold the result, and returns that. *matrix* must be a two-dimensional array. *into-matrix*, if provided, must be two-dimensional and have sufficient dimensions to hold the transpose of *matrix*.

math:determinant *matrix* *Function*

Returns the determinant of *matrix*. *matrix* must be a two-dimensional square matrix.

The next two functions are used to solve sets of simultaneous linear equations. **math:decompose** takes a matrix holding the coefficients of the equations and produces the LU decomposition; this decomposition can then be passed to **math:solve** along with a vector of right-hand sides to get the values of the variables. If you want to solve the same equations for many different sets of right-hand side values, you only need to call **math:decompose** once. In terms of the argument names used below, these two functions exist to solve the vector equation $A x = b$ for x. A is a matrix. b and x are vectors.

math:decompose *a* &optional *lu ps* *Function*

Computes the LU decomposition of matrix *a*. If *lu* is non-**nil**, stores the result into it and returns it; otherwise it creates an array to hold the result, and returns that. The lower triangle of *lu*, with ones added along the diagonal, is L, and the upper triangle of *lu* is U, such that the product of L and U is *a*. Gaussian elimination with partial pivoting is used. The *lu* array is permuted by rows according to the permutation array *ps*, which is also produced by this function. If the argument *ps* is supplied, the permutation array is stored into it; otherwise, an array is created to hold it. This function returns two values: the LU decomposition and the permutation array.

math:solve *lu ps b* &optional *x* *Function*

This function takes the LU decomposition and associated permutation array produced by **math:decompose**, and solves the set of simultaneous equations defined by the original matrix *a* and the right-hand sides in the vector *b*. If *x* is supplied, the solutions are stored into it and it is returned; otherwise, an array is created to hold the solutions and that is returned. *b* must be a one-dimensional array.

math:list-2d-array *array* *Function*

Returns a list of lists containing the values in *array*, which must be a two-dimensional array. There is one element for each row; each element is a list of the values in that row.

math:fill-2d-array *array list* *Function*

This is the opposite of **math:list-2d-array**. *list* should be a list of lists, with each element being a list corresponding to a row. *array*'s elements are stored from the list. Unlike **fillarray**, if *list* is not long enough, **math:fill-2d-array** "wraps around", starting over at the beginning. The lists that are elements of *list* also work this way.

1.9 Planes

A *plane* is an array whose bounds, in each dimension, are plus-infinity and minus-infinity; all integers are legal as indices. Planes are distinguished not by size and shape, but by number of dimensions alone. When a plane is created, a default value must be specified. At that moment, every component of the plane has that value. As you cannot ever change more than a finite number of components, only a finite region of the plane need actually be stored.

The regular array accessing functions do not work on planes. You can use **make-plane** to create a plane, **plane-aref** or **plane-ref** to get the value of a component, and **plane-aset** or **plane-store** to store into a component. **array-#-dims** will work on a plane.

A plane is actually stored as an array with a leader. The array corresponds to a rectangular, aligned region of the plane, containing all the components in which a **plane-store** has been done (and others, in general, which have never been altered). The lowest-coordinate corner of that rectangular region is given by the **plane-origin** in the array leader. The highest coordinate corner can be found by adding the **plane-origin** to the **array-dimensions** of the array. The **plane-default** is the contents of all the elements of the plane that are not actually stored in the array. The **plane-extension** is the amount to extend a plane by in any direction when the plane needs to be extended. The default is 32.

If you never use any negative indices, then the **plane-origin** will be all zeroes and you can use regular array functions, such as **aref** and **aset**, to access the portion of the plane which is actually stored. This can be useful to speed up certain algorithms. In this case you can even use the **bitblt** function on a two-dimensional plane of bits or bytes, provided you don't change the **plane-extension** to a number that is not a multiple of 32.

make-plane *rank* &rest *options* *Function*

Creates and returns a plane. *rank* is the number of dimensions. *options* is a list of alternating keyword symbols and values. The allowed keywords are:

:type The array type symbol (for example, **art-1b**) specifying the type of the array out of which the plane is made.

:default-value
The default component value.

:extension
The amount by which to extend the plane. See the section "Planes".

:initial-dimensions
A list of dimensions for the initial creation of the plane. You might want to use this option to create a plane whose first dimension is a multiple of 32, so you can use **bitblt** on it. Default: the result returned by **(make-list** *rank* **':initial-value 1)**.

:initial-origins
A list of origins for the initial creation of the plane. Default: the result returned by **(make-list** *rank* **':initial-value 0)**.

Example:

```
(make-plane 2 ':type 'art-4b ':default-value 3)
```

creates a two-dimensional plane of type **art-4b**, with default value **3**.

plane-origin *plane* *Function*
A list of numbers, giving the lowest coordinate values actually stored.

plane-default *plane* *Function*
This is the contents of the infinite number of plane elements that are not actually stored.

plane-extension *plane* *Function*
The amount to extend the plane by in any direction when **plane-store** is done outside of the currently stored portion.

plane-aref *plane* &rest *subscripts* *Function*
plane-aref and **plane-ref** return the contents of a specified element of a plane. They differ only in the way they take their arguments; **plane-aref** takes the subscripts as arguments, while **plane-ref** takes a list of subscripts.

plane-ref *plane subscripts* *Function*
plane-aref and **plane-ref** return the contents of a specified element of a plane. They differ only in the way they take their arguments; **plane-aref** takes the subscripts as arguments, while **plane-ref** takes a list of subscripts.

plane-aset *datum plane* &rest *subscripts* *Function*
plane-aset and **plane-store** store *datum* into the specified element of a plane, extending it if necessary, and return *datum*. They differ only in the way they take their arguments; **plane-aset** takes the subscripts as arguments, while **plane-store** takes a list of subscripts.

plane-store *datum plane subscripts* *Function*
plane-aset and **plane-store** store *datum* into the specified element of a plane, extending it if necessary, and return *datum*. They differ only in the way they take their arguments; **plane-aset** takes the subscripts as arguments, while **plane-store** takes a list of subscripts.

1.10 Maclisp Array Compatibility

The functions in this section are provided only for Maclisp compatibility, and should not be used in new programs.

Fixnum arrays do not exist (however, see Zetalisp's small-positive-integer arrays). Flonum arrays exist but you do not use them in the same way; no declarations are required or allowed. "Un-garbage-collected" arrays do not exist.

Readtables and obarrays are represented as arrays, but unlike Maclisp special array types are not used. Information about readtables and obarrays (packages) can be found elsewhere: See the function **read**. See the function **intern**. There are no "dead" arrays, nor are Multics "external" arrays provided.

The **arraycall** function exists for compatibility but should not be used. See the function **aref**.

Subscripts are always checked for validity, regardless of the value of ***rset** and whether the code is compiled or not. However, in a multidimensional array, an error is only caused if the subscripts would have resulted in a reference to storage outside of the array. For example, if you have a 2 by 7 array and refer to an element with subscripts 3 and 1, no error 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, subscript errors will be caught if and only if they refer to storage outside the array; some errors are undetected, but they will only clobber some other element of the same array rather than clobbering something completely unpredictable.

Currently, multidimensional arrays are stored in column-major order rather than row-major order as in Maclisp. See the section "Arrays: Arrays and Strings". This issue is discussed further in that section.

loadarrays and **dumparrays** are not provided. However, arrays can be put into compiled code files. See the section "Putting Data in Compiled Code Files".

The ***rearray** function is not provided, since not all of its functionality is available in Zetalisp. The most common uses can be replaced by **adjust-array-size**.

In Maclisp, arrays are usually kept on the **array** property of symbols, and the symbols are used instead of the arrays. In order to provide some degree of compatibility for this manner of using arrays, the **array**, ***array**, and **store** functions (**store** on the LM-2 only) are provided, and when arrays are applied to arguments, the arguments are treated as subscripts and **apply** returns the corresponding element of the array.

store is not supported on the 3600. Supporting **store** would require two additional words of state in each stack group. This would prevent storing into the last array that was referenced by some other process, in the event of a process switch in the middle of a **store** operation. Maintaining this state would slow down all stack group switches. In addition, using arrays as functions, as in **store**, is many times slower

than using functions like **aref** and **aset** on the 3600. The use of arrays as functions is not implemented in microcode, and the macrocode has not been optimized.

array &quote *symbol type* &eval &rest *dims* *Function*
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 first argument.) *type* is ignored. If *symbol* is **nil**, the array is returned; otherwise, the array is put in the function cell of *symbol*, and *symbol* is returned.

***array** *symbol type* &rest *dims* *Function*
This is just like **array**, except that all of the arguments are evaluated.

store *array-ref x* *Special Form*
store stores *x* into the specified array element. *array-ref* should be a form that references an array by calling it as a function (**aref** forms are not acceptable). First *x* is evaluated, then *array-ref* is evaluated, and then the value of *x* is stored into the array cell last referenced by a function call, presumably the one in *array-ref*.

xstore *x array-ref* *Function*
This is just like **store**, but it is not a special form; this is because the arguments are in the other order. This function only exists for the compiler to compile the **store** special form into, and should never be used by programs. **xstore** is available only on the LM-2.

arraycall *ignore array* &rest *subscripts* *Function*
(arraycall t *array sub1 sub2...***)** is the same as **(aref** *array sub1 sub2...***)**. It exists for Maclisp compatibility.

2. Strings

Strings are a type of array that represent a sequence of characters. The printed representation of a string is its characters enclosed in quotation marks, for example, **"foo bar"**. Strings are constants, that is, evaluating a string returns that string. Strings are the right data type to use for text processing.

Strings are arrays of type **art-string**, where each element holds an eight-bit unsigned fixnum. This is because characters are represented as fixnums, and for fundamental characters only eight bits are used. A string can also be an array of type **art-fat-string**, where each element holds a sixteen-bit unsigned fixnum; the extra bits allow for multiple fonts or an expanded character set.

See the section "The Character Set". The way characters work, including multiple fonts and the extra bits from the keyboard, is explained in that section. Note that you can type in the fixnums that represent characters using "#/" and "#\"; for example, **#/f** reads in as the fixnum that represents the character "f", and **#\return** reads in as the fixnum that represents the special "return" character. See the section "Sharp-sign Abbreviations". Details of this syntax are explained there.

The functions described in this section provide a variety of useful operations on strings. 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 one-character string containing the character designated by that fixnum will be used. Several of the functions actually work on any type of one-dimensional array and may be useful for other than string processing; these are the functions such as **substring** and **string-length** that do not depend on the elements of the string being characters.

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
```

Note that the character at the beginning of the string is element zero of the array (rather than one); as usual in Zetalisp, everything is zero-based.

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. When you are making strings that you intend to change later, you probably want to create an array with a fill-pointer so that you can change the length of the string as well as the contents. See the section "Extra Features of Arrays". The length of a string is always computed using **array-active-length**, so that if a string has a fill-pointer, its value will be used as the length.

2.1 Characters

character *x* *Function*

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. See the section "The Character Set". The way characters are represented as fixnums is explained in that section.

char-equal *char1 char2* *Function*

This is the primitive for comparing characters for equality; many of the string functions call it. *char1* and *char2* 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 *char1 char2* *Function*

This is the primitive for comparing characters for order; many of the string functions call it. *char1* and *char2* must be fixnums. The result is **t** if *char1* comes before *char2* ignoring case and font, otherwise **nil**. See the section "The Character Set". Details of the ordering of characters are in that section.

2.2 Upper and Lowercase Letters

alphabetic-case-affects-string-comparison *Variable*

This variable is normally **nil**. If it is **t**, **char-equal**, **char-lessp**, and the string searching and comparison functions will distinguish between uppercase and lowercase letters. If it is **nil**, lowercase characters behave as if they were the same character but in uppercase. It is all right to bind this to **t** around a string operation, but changing its global value to **t** will break many system functions and user interfaces and so is not recommended.

char-upcase *char* *Function*

If *char*, which must be a fixnum, is a lowercase alphabetic character its uppercase form is returned; otherwise, *char* itself is returned. If font information is present it is preserved. The result of **char-upcase** is undefined for characters with modifier bits.

char-downcase *char* *Function*

If *char*, which must be a fixnum, is an uppercase alphabetic character its lowercase form is returned; otherwise, *char* itself is returned. If font information is present it is preserved. The result of **char-downcase** is undefined for characters with modifier bits.

string-upcase *string* &optional (*from* **0**) *to* (*copy-p* **t**) *Function*

If *copy-p* is not **nil**, returns a copy of *string*, with lowercase alphabetic characters replaced by the corresponding uppercase characters. If *copy-p* is **nil**, uppercases characters in *string* itself and then returns the modified *string*. *from* is the index in *string* at which to begin uppercasing characters. If *to* is supplied, it is used in place of (**array-active-length** *string*) as the index one greater than the last character to be uppercased.

string-downcase *string* &optional (*from* **0**) *to* (*copy-p* **t**) *Function*

If *copy-p* is not **nil**, returns a copy of *string*, with uppercase alphabetic characters replaced by the corresponding lowercase characters. If *copy-p* is **nil**, lowercases characters in *string* itself and then returns the modified *string*. *from* is the index in *string* at which to begin lowercasing characters. If *to* is supplied, it is used in place of (**array-active-length** *string*) as the index one greater than the last character to be lowercased.

string-capitalize-words *string* &optional (*copy-p* **t**) *Function*

Transforms *string* by changing hyphens to spaces and capitalizing each word.

```
(string-capitalize-words "Lisp-listener") => "Lisp Listener"
(string-capitalize-words "LISP-LISTENER")  => "Lisp Listener"
(string-capitalize-words "lisp--listener") => "Lisp  Listener"
(string-capitalize-words "symbol-processor-3") => "Symbol Processor 3"
```

copy-p indicates whether to return a copy of the string argument or to modify the argument itself. The default, **t**, returns a copy.

2.3 Basic String Operations

string *x* *Function*

string coerces *x* into a string. Most of the string functions apply this to their string arguments. If *x* is a string (or any array), it is returned. If *x* is a symbol, its pname is returned. If *x* is a nonnegative fixnum less than **400** octal, a one-character-long string containing it is created and returned. If *x* is a pathname, the "string for printing" is returned. See the section "Naming of Files". Otherwise, an error is signalled.

If you want to get the printed representation of an object into the form of a string, this function is *not* what you should use. You can use **format**, passing a first argument of **nil**. You might also want to use **with-output-to-string**.

string-length *string* *Function*

string-length returns the number of characters in *string*. This function uses the same coercion rules as **string** in interpreting *string* as a string. **string-length** returns the **array-active-length** if *string* is a string, or the **array-active-length** of the pname if *string* is a symbol.

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

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 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-equal *string1 index1 string2 index2 count* *Function*

%string-equal is the microcode primitive that **string-equal** calls. It returns **t** if the *count* characters of *string1* starting at *idx1* are **char-equal** to the *count* characters of *string2* starting at *idx2*, or **nil** if the characters are not equal or if *count* runs off the length of either array.

Instead of a fixnum, *count* may also be **nil**. In this case, **%string-equal** compares the substring from *idx1* to (**string-length** *string1*) against the substring from *idx2* to (**string-length** *string2*). If the lengths of these substrings differ, then they are not equal and **nil** is returned.

Note that *string1* and *string2* must really be strings; the usual coercion of symbols and fixnums to strings is not performed. This function is documented because certain programs that require high efficiency and are willing to pay the price of less generality may want to use **%string-equal** in place of **string-equal**. Examples:

To compare the two strings *foo* and *bar*:

```
(%string-equal foo 0 bar 0 nil)
```

To see if the string *foo* starts with the characters **"bar"**:

```
(%string-equal foo 0 "bar" 0 3)
```

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

string-lessp compares two strings using alphabetical order (as defined by **char-lessp**). The result is **t** if *string1* is the lesser, or **nil** if they are equal or *string2* is the lesser.

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

Compares the characters of *string1* starting at *idx1* and ending just below *lim1* with the characters of *string2* starting at *idx2* and ending just below *lim2*. The comparison is in alphabetical order. *lim1* and *lim2* default to the lengths of the strings. **string-compare** returns:

- a positive number if *string1* > *string2*
- zero if *string1* = *string2*
- a negative number if *string1* < *string2*

If the strings are not equal, the absolute value of the number returned is one more than the index (in *string1*) at which the difference occurred.

string-compare uses the same rules as **string** in coercing *string1* and *string2* into strings.

substring *string from* &optional *to* (*area* **nil**) *Function*

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 from* &optional *to* (*area* **nil**) *Function*

nsubstring is the same as **substring** except that the substring is not copied; instead an indirect array is created that shares part of the argument *string*. See the section "Extra Features of Arrays". Modifying one string modifies the other.

Note that **nsubstring** does not necessarily use less storage than **substring**; an **nsubstring** of any length uses at least as much storage as a **substring** 12 characters long. So you should not use this just "for efficiency"; it is intended for uses in which it is important to have a substring that, if modified, will cause the original string to be modified too.

string-append &rest *strings* *Function*

Any number of strings are copied and concatenated into a single string. With a single argument, **string-append** simply copies it. The result is an array of the same type as the argument with the greatest number of bits per element. For example, if the arguments are arrays of type **art-string** and **art-fat-string**, an array of type **art-fat-string** is returned. **string-append** can be used to copy and concatenate any type of one-dimensional array. Example:

```
(string-append #/! "foo" #/!) => "!foo!"
```

string-nconc *modified-string* &rest *strings* *Function*

string-nconc is like **string-append** except that instead of making a new string containing the concatenation of its arguments, **string-nconc** modifies its first argument. *modified-string* must have a fill-pointer so that additional characters can be tacked onto it. Compare this with **array-push-extend**. The value of **string-nconc** is *modified-string* or a new, longer copy of it; in

the latter case the original copy is forwarded to the new copy (see **adjust-array-size**). Unlike **nconc**, **string-nconc** with more than two arguments modifies only its first argument, not every argument but the last.

string-trim *char-set string* *Function*

This returns a **substring** of *string*, with all characters in *char-set* stripped off the beginning and end. *char-set* is a set of characters, which can be represented as a list of characters or a string of characters. Example:

```
(string-trim '(#\sp) "  Dr. No  ") => "Dr. No"
(string-trim "ab" "abbafooabb") => "foo"
```

string-left-trim *char-set string* *Function*

This returns a **substring** of *string*, with all characters in *char-set* stripped off the beginning. *char-set* is a set of characters, which can be represented as a list of characters or a string of characters.

string-right-trim *char-set string* *Function*

This returns a **substring** of *string*, with all characters in *char-set* stripped off the end. *char-set* is a set of characters, which can be represented as a list of characters or a string of characters.

string-reverse *string* *Function*

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

string-nreverse *string* *Function*

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 one-dimensional array of any type.

string-pluralize *string* *Function*

string-pluralize returns a string containing the plural of the word in the argument *string*. Any added characters go in the same case as the last character of *string*. Example:

```
(string-pluralize "event") => "events"
(string-pluralize "Man") => "Men"
(string-pluralize "Can") => "Cans"
(string-pluralize "key") => "keys"
(string-pluralize "TRY") => "TRIES"
```

For words with multiple plural forms depending on the meaning, **string-pluralize** cannot always do the right thing.

parse-number *string* &optional (*from* **0**) (*to nil*) (*radix nil*) *Function*
(*fail-if-not-whole-string nil*)

parse-number takes a string and "reads" a number from it. It returns two

values: the number found (or **nil**) and the character position of the next unparsed character in the string. It returns **nil** when the first character that it looks at cannot be part of a number. The function currently does not handle anything but integers. (**read-from-string** is a more general function that uses the Lisp Reader; **prompt-and-read** reads a number from the keyboard.)

```
(parse-number "123    ")  =>  123 3
(parse-number "   123")  =>  NIL 0
(parse-number "-123")  =>  -123 4
(parse-number "25.3")  =>  25 2
(parse-number "$$$123" 3 4)  =>  1 4
(parse-number "123$$$" 0 nil nil nil) => 123 3
(parse-number "123$$$" 0 nil nil t) => NIL 0
```

Four optional arguments:

from The character position in the string to start parsing. The default is the first one, position 0.

to The character position past the last one to consider. The default, **nil**, means the end of the string.

radix The radix to read the string in. The default, **nil**, means base 10.

fail-if-not-whole-string
The default is **nil**. **nil** means to read up to the first character that is not a digit and stop there, returning the result of the parse so far. **t** means to stop at the first non-digit and to return **nil** and 0 length if that is not the end of the string.

2.4 String Searching

string-search-char *char string* &optional (*from* **0**) *to* *Function*
string-search-char searches through *string* starting at the index *from*, which defaults to the beginning, and returns the index of the first character that is **char-equal** to *char*, or **nil** if none is found. If the *to* argument is supplied, it is used in place of (**string-length** *string*) to limit the extent of the search. Example:

```
(string-search-char #/a "banana") => 1
```

%string-search-char *char string from to* *Function*
%string-search-char is the microcode primitive that **string-search-char** and other functions call. *string* must be an array and *char*, *from*, and *to* must be fixnums. Except for this lack of type-coercion, and the fact that

none of the arguments is optional, **%string-search-char** is the same as **string-search-char**. This function is documented for the benefit of those who require the maximum possible efficiency in string searching.

string-search-not-char *char string* &optional (*from* **0**) *to* *Function*
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. If the *to* argument is supplied, it is used in place of (**string-length** *string*) to limit the extent of the search. Example:

```
(string-search-not-char #/b "banana") => 1
```

string-search *key string* &optional (*from* **0**) *to* (*key-start* **0**) *key-end* *Function*
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. If the *to* argument is supplied, it is used in place of (**string-length** *string*) to limit the extent of the search. Example:

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

string-search-set *char-set string* &optional (*from* **0**) *to* *Function*
string-search-set searches through *string* looking for a character that is in *char-set*. The search begins at the index *from*, which defaults to the beginning. It returns the index of the first character that is **char-equal** to some element of *char-set*, or **nil** if none is found. If the *to* argument is supplied, it is used in place of (**string-length** *string*) to limit the extent of the search. *char-set* is a set of characters, which can be represented as a list of characters or a string of characters. Example:

```
(string-search-set '(#/n #/o) "banana") => 2
(string-search-set "no" "banana") => 2
```

string-search-not-set *char-set string* &optional (*from* **0**) *to* *Function*
string-search-not-set searches through *string* looking for a character that is not in *char-set*. The search begins at the index *from*, which defaults to the beginning. It returns the index of the first character that is not **char-equal** to any element of *char-set*, or **nil** if none is found. If the *to* argument is supplied, it is used in place of (**string-length** *string*) to limit the extent of the search. *char-set* is a set of characters, which can be represented as a list of characters or a string of characters. Example:

```
(string-search-not-set '(#/a #/b) "banana") => 2
```

string-reverse-search-char *char string* &optional *from* (*to* **0**) *Function*
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 that 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. If the *to* argument is supplied, it limits the extent of the search. Example:

```
(string-reverse-search-char #/n "banana") => 4
```

string-reverse-search-not-char *char string* &optional *from* (*to* **0**) *Function*

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 that 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. If the *to* argument is supplied, it limits the extent of the search. Example:

```
(string-reverse-search-not-char #/a "banana") => 4
```

string-reverse-search *key string* &optional *from* (*to* **0**) (*key-start* **0**) *Function*
key-end

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*. If the *to* argument is supplied, it limits the extent of the search. Example:

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

string-reverse-search-set *char-set string* &optional *from* (*to* **0**) *Function*

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 that is **char-equal** to some element of *char-set*, 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. If the *to* argument is supplied, it limits the extent of the search. *char-set* is a set of characters, which can be represented as a list of characters or a string of characters.

```
(string-reverse-search-set "ab" "banana") => 5
```

string-reverse-search-not-set *char-set string* &optional *from* (*to* **0**) *Function*

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 that is not **char-equal** to any element of *char-set*, 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. If the *to* argument is supplied, it limits the extent of the search. *char-set* is a set of characters, which can be represented as a list of characters or a string of characters.

```
(string-reverse-search-not-set '(#/a #/n) "banana") => 0
```

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

2.5 I/O to Strings

The special forms in this section allow you to create I/O streams that input from or output to a string rather than a real I/O device. See the section "What Streams Are". I/O streams are documented there.

with-input-from-string *(var string [index] [limit]) body...* *Special Form*

The form:

```
(with-input-from-string (var string)
    body)
```

evaluates the forms in *body* with the variable *var* bound to a stream that reads characters from the string which is the value of the form *string*. The value of the special form is the value of the last form in its body.

The stream is a function that only works inside the **with-input-from-string** special form, so be careful what you do with it. You cannot use it after control leaves the body, and you cannot nest two **with-input-from-string** special forms and use both streams since the special-variable bindings associated with the streams will conflict. It is done this way to avoid any allocation of memory.

After *string* you may optionally specify two additional "arguments". The first is *index*:

```
(with-input-from-string (var string index)
    body)
```

uses *index* as the starting index into the string, and sets *index* to the index of the first character not read when **with-input-from-string** returns. If the whole string is read, it will be set to the length of the string. Since *index* is updated it may not be a general expression; it must be a variable or a **setf**able reference. The *index* is not updated in the event of an abnormal exit from the body, such as a ***throw**. The value of *index* is not updated until **with-input-from-string** returns, so you cannot use its value within the body to see how far the reading has proceeded.

Use of the *index* feature prevents multiple values from being returned out of the body, currently.

```
(with-input-from-string (var string index limit)
    body)
```

uses the value of the form *limit*, if the value is not **nil**, in place of the length of the string. If you want to specify a *limit* but not an *index*, write **nil** for *index*.

with-output-to-string *(var [string] [index]) body...* *Special Form*

This special form provides a variety of ways to send output to a string through an I/O stream.

```
(with-output-to-string (var)
  body)
```

evaluates the forms in *body* with *var* bound to a stream that saves the characters output to it in a string. The value of the special form is the string.

```
(with-output-to-string (var string)
  body)
```

will append its output to the string which is the value of the form *string*. (This is like the **string-nconc** function). The value returned is the value of the last form in the body, rather than the string. Multiple values are not returned. *string* must have an array-leader; element 0 of the array-leader will be used as the fill-pointer. If *string* is too small to contain all the output, **adjust-array-size** will be used to make it bigger.

```
(with-output-to-string (var string index)
  body)
```

is similar to the above except that *index* is a variable or **setf**able reference that contains the index of the next character to be stored into. It must be initialized outside the **with-output-to-string** and will be updated upon normal exit. The value of *index* is not updated until **with-output-to-string** returns, so you cannot use its value within the body to see how far the writing has gotten. The presence of *index* means that *string* is not required to have a fill-pointer; if it does have one it will be updated.

The stream is a "downward closure" simulated with special variables, so be careful what you do with it. You cannot use it after control leaves the body, and you cannot nest two **with-output-to-string** special forms and use both streams since the special-variable bindings associated with the streams will conflict. It is done this way to avoid any allocation of memory.

You can to use a **with-input-from-string** and **with-output-to-string** nested within one another, so long as there is only one of each.

Another way of doing output to a string is to use the **format** facility.

2.6 Maclisp-compatible Functions

The following functions are provided primarily for Maclisp compatibility.

alphalessp *string1 string2* *Function*

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

getchar *string index* *Function*

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* *Function*

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* *Function*

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 current package.

maknam *char-list* *Function*

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

```
(maknam '(a b #/0 d)) => ab0d
```

implode *char-list* *Function*

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

The **samepnamep** function is also provided.

Index

#

A

G

G

G

H

I

L

M

N

N N

O

O O

W

X