Lisp Machine Manual

Fourth Edition

July 1981

Daniel Weinreb
David Moon

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

Preface

The Lisp Machine manual describes both the language and the "operating system" of the Lisp Machine. The language, a dialect of Lisp called Zetalisp, is completely documented by this manual. The software environment and operating-system-like parts of the system contain many things which are still in a state of flux. This manual confines itself primarily to the stabler parts of the system, and does not address the window system and user interface at all. That documentation will be released as a separate volume at a later time.

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

Those not on the Arpanet may send U.S. mail to
Daniel L. Weinreb or David A. Moon
Room 926
545 Technology Square
Cambridge, Mass. 02139

Note

The Lisp Machine is a product of the efforts of many people too numerous to list here and of the unique environment of the M.I.T. Artificial Intelligence Laboratory.

Portions of this manual were written by Richard Stallman, Mike McMahon, and Alan Bawden. The chapter on the LOOP iteration macro is a reprint of Laboratory for Computer Science memo TM-169, by Glenn Burke.

Summary Table of Contents

1. Introduction 1
2. Primitive Object Types 7
3. Evaluation 13
4. Flow of Control. 33
5. Manipulating List Structure. 52
6. Symbols 86
7. Numbers 92
8. Arrays 107
9. Strings 126
10. Functions 136
11. Closures. 158
12. Stack Groups 163
13. Locatives 170
14. Subprimitives 172
15. Areas 192
16. The Compiler 197
17. Macros 208
18. The LOOP Iteration Macro 233
19. Defstruct 257
20. Objects, Message Passing, and Flavors 279
21. The I/O System 314
22. Naming of Files 376
23. Packages 392
24. Maintaining Large Systems 406
25. Processes 428
26. Errors and Debugging 440
27. How to Read Assembly Language 469
28. Querying the User. 487
29. Initializations 490
30. Dates and Times. 493
31. Miscellaneous Useful Functions 499
Concept Index. 509
Flavor Index. 512
Message Index. 513
Keyword Index 515
Object Creation Options 518
Meter Index 519
Variable Index. 520
Function Index 523

Table of Contents

1. Introduction. 1
1.1 General Information 1
1.2 Structure of the Manual. 1
1.3 Notational Conventions and Helpful Notes 2

2. Primitive Object Types 7
2.1 Data Types 7
2.2 Predicates. 8

3. Evaluation 13
3.1 Variables 14
3.2 Functions. 20
3.3 Some Functions and Special Forms 23
3.4 Multiple Values. 29

4. Flow of Control. 33
4.1 Conditionals 33
4.2 Iteration 38
4.3 Non-Local Exits. 47
4.4 Mapping 50

5. Manipulating List Structure 52
5.1 Conses 53
5.2 Lists 54
5.3 Alteration of List Structure 61
5.4 Cdr-Coding. 63
5.5 Tables 65
5.6 Lists as Tables. 66
5.7 Association Lists 69
5.8 Property Lists. 71
5.9 Hash Tables. 74
5.9.1 Hashing on Eq. 75
5.9.2 Hashing on Equal 76
5.9.3 Hash Tables and the Garbage Collector. 78
5.9.4 Hash Primitive. 78
5.10 Sorting 79
5.11 Resources 81

6. Symbols. 86
6.1 The Value Cell 86
6.2 The Function Cell. 87
6.3 The Property List. 88
6.4 The Print Name. 89
6.5 The Package Cell 89
6.6 Creating Symbols. 90

7. Numbers 92
7.1 Numeric Predicates 94
7.2 Numeric Comparisons 95
7.3 Arithmetic 96
7.4 Transcendental Functions 99
7.5 Numeric Type Conversions 99
7.6 Logical Operations on Numbers 100
7.7 Byte Manipulation Functions 102
7.8 Random Numbers 104
7.9 24-Bit Numbers 105
7.10 Double-Precision Arithmetic 105

8. Arrays 107
8.1 Extra Features of Arrays 110
8.2 Basic Array Functions 111
8.3 Getting Information About an Array 114
8.4 Changing the Size of an Array 116
8.5 Arrays Overlaid With Lists 117
8.6 Adding to the End of an Array 118
8.7 Copying an Array 119
8.8 Matrices and Systems of Linear Equations 121
8.9 Planes 122
8.10 Maclisp Array Compatibility 124

9. Strings 126
9.1 Characters 127
9.2 Upper and Lower Case Letters 127
9.3 Basic String Operations 128
9.4 String Searching 131
9.5 I/O to Strings 133
9.6 Maclisp-Compatible Functions 135

10. Functions 136
10.1 What Is a Function? 136
10.2 Function Specs 136
10.3 Simple Function Definitions 139
10.4 Operations the User Can Perform on Functions 141
10.5 Kinds of Functions 141
10.5.1 Interpreted Functions 143
10.5.2 Compiled Functions 144
10.5.3 Other Kinds of Functions 144
10.6 Function-Defining Special Forms 145
10.7 Lambda-List Keywords 148
10.8 How Programs Manipulate Definitions 149
10.9 How Programs Examine Functions 150
10.10 Encapsulations 153
10.10.1 Rename-Within Encapsulations 156

11. Closures 158
11.1 What a Closure Is 158
11.2 Examples of the Use of Closures 160
11.3 Closure-Manipulating Functions 161
11.4 Entities 162

12. Stack Groups 163
12.1 Resuming of Stack Groups 164
12.2 Stack Group States 165
12.3 Stack Group Functions 166
12.4 Input/Output in Stack Groups 167
12.5 An Example of Stack Groups 168

13. Locatives 170
13.1 Cells and Locatives 170
13.2 Functions Which Operate on Locatives 170

14. Subprimitives 172
14.1 Data Types 173
14.2 Forwarding 174
14.3 Pointer Manipulation 176
14.4 Analyzing Structures 176
14.5 Creating Objects 177
14.6 Locking Subprimitive 178
14.7 I/O Device Subprimitives 178
14.8 Special Memory Referencing 179
14.9 Storage Layout Definitions 181
14.10 Function-Calling Subprimitives 182
14.11 Lambda-Binding Subprimitive 183
14.12 The Paging System 183
14.13 Closure Subprimitives 185
14.14 Microcode Variables 186
14.15 Meters 188

15. Areas 192
15.1 Area Functions and Variables 193
15.2 Interesting Areas 195

16. The Compiler 197
16.1 The Basic Operations of the Compiler 197
16.2 How to Invoke the Compiler 197
16.3 Input to the Compiler 198
16.4 Compiler Declarations 201
16.5 Controlling Compiler Warnings 203
16.6 Compiler Source-Level Optimizers 204
16.7 Files that Maclisp Must Compile 205
16.8 Putting Data in QFASL Files 206

17. Macros 208
17.1 Introduction to Macros 208
17.2 Aids for Defining Macros 210

17.2.1 Defmacro 210
17.2.2 Backquote 211
17.3 Substitutable Functions 215
17.4 Hints to Macro Writers 216
17.4.1 Name Conflicts. 216
17.4.2 prog-context Conflicts 218
17.4.3 Macros Expanding into Many Forms 219
17.4.4 Macros that Surround Code. 220
17.4.5 Multiple and Out-of-order Evaluation. 222
17.4.6 Nesting Macros. 223
17.4.7 Functions Used During Expansion 225
17.5 Aids for Debugging Macros 226
17.6 Displacing Macros. 226
17.7 Advanced Features of Defmacro. 227
17.8 Functions to Expand Macros 228
17.9 Generalized Variables 229

18. The LOOP Iteration Macro 233
18.1 Introduction. 233
18.2 Clauses 234
18.2.1 Iteration-Driving Clauses 235
18.2.2 Bindings 237
18.2.3 Entrance and Exit 239
18.2.4 Side Effects. 240
18.2.5 Values 240
18.2.6 Endtests 242
18.2.7 Aggregated Boolean Tests 242
18.2.8 Conditionalization 243
18.2.9 Miscellaneous Other Clauses 244
18.3 Loop Synonyms. 245
18.4 Data Types 245
18.5 Destructuring 246
18.6 The Iteration Framework 247
18.7 Iteration Paths. 249
18.7.1 Pre-Defined Paths 250
18.7.1.1 The Interned-Symbols Path 250
18.7.1.2 Sequence Iteration. 251
18.7.2 Defining Paths 252
18.7.2.1 An Example Path Definition. 254

19. Defstruct 257
19.1 Introduction to Structure Macros 257
19.2 How to Use Defstruct 259
19.3 Options to Defstruct. 260
19.4 Using the Constructor and Alterant Macros 267
19.4.1 Constructor Macros. 267
19.4.2 By-position Constructor Macros 268
19.4.3 Alterant Macros 269
19.5 Byte Fields 269

19.6 Grouped Arrays 271
19.7 Named Structures 271
19.8 The si:defstruct-description Structure. 273
19.9 Extensions to Defstruct 275
19.9.1 An Example 275
19.9.2 Syntax of defstruct-define-type 275
19.9.3 Options to defstruct-define-type. 276

20. Objects, Message Passing, and Flavors 279
20.1 Introduction. 279
20.2 Objects 279
20.3 Modularity 280
20.4 Generic Operations 283
20.5 Generic Operations in Lisp. 284
20.6 Simple Use of Flavors 286
20.7 Mixing Flavors 289
20.8 Flavor Functions. 292
20.9 Defflavor Options 300
20.10 Flavor Families. 304
20.11 Vanilla Flavor 305
20.12 Method Combination. 306
20.13 Implementation of Flavors 309
20.13.1 Order of Definition 309
20.13.2 Changing a Flavor. 310
20.13.3 Restrictions 311
20.14 Entities. 311
20.15 Useful Editor Commands. 311
20.16 Property List Messages 312

21. The I/O System 314
21.1 The Character Set 314
21.2 Printed Representation. 319
21.2.1 What the Printer Produces. 319
21.2.2 What The Reader Accepts. 322
21.2.3 Macro Characters. 325
21.2.4 Sharp-sign Abbreviations 325
21.2.5 Special Character Names 328
21.2.6 The Readtable 328
21.3 Input Functions 331
21.4 Output Functions 334
21.5 I/O Streams. 338
21.5.1 What Streams Are. 338
21.5.2 General Purpose Stream Operations. 338
21.5.3 Special Purpose Stream Operations 341
21.5.4 Standard Streams. 343
21.5.5 Making Your Own Stream 344
21.6 Formatted Output 346
21.6.1 The Format Function. 346
21.6.2 The Output Subsystem 356

21.6.3 Formatting Lisp Code . . . 360
21.7 Rubout Handling . . . 361
21.8 The :read and :print Stream Operations . . . 364
21.9 Accessing Files . . . 365
21.9.1 Loading Files. . . . 368
21.9.2 File Property Lists . . . 369
21.9.3 File Stream Operations . . . 371
21.10 Accessing Directories. . . . 373

22. Naming of Files . . . 376
22.1 Pathnames . . . 376
22.2 Defaults and Merging . . . 380
22.3 Generic Pathnames . . . 381
22.4 Pathname Functions. . . . 382
22.5 Pathname Messages . . . 384
22.6 Host File Systems Supported . . . 386
22.6.1 ITS . . . 386
22.6.2 TOPS-20 and Tenex . . . 387
22.6.3 Logical Pathnames . . . 388
22.7 Maclisp Conversion . . . 390
22.8 Examples . . . 390

23. Packages . . . 392
23.1 The Need for Multiple Contexts . . . 392
23.2 The Organization of Name Spaces . . . 393
23.3 Shared Programs . . . 394
23.4 Declaring Packages . . . 395
23.5 Packages and Writing Code . . . 396
23.6 Shadowing . . . 397
23.7 Packages and Interning . . . 398
23.8 Status Information . . . 401
23.9 Packages, Loading, and Compilation . . . 402
23.10 Subpackages . . . 402
23.11 Initialization of the Package System . . . 403
23.12 Initial Packages. . . . 404

24. Maintaining Large Systems . . . 406
24.1 Defining a System . . . 406
24.2 Transformations. . . . 409
24.3 Making a System . . . 411
24.4 Adding New Keywords to make-system . . . 412
24.5 Adding New Options for defsystem . . . 414
24.6 More Esoteric Transformations . . . 415
24.7 The Patch Facility . . . 416
24.7.1 Defining a System . . . 418
24.7.2 Patch files . . . 419
24.7.3 Loading Patches . . . 420
24.7.4 Making Patches . . . 420
24.7.5 System Status. . . . 421
24.8 Saving New Versions: Disk Partitions . . . 422

24.8.1 Concepts . . . 422
24.8.2 Manipulating the Label . . . 422
24.8.3 Updating Software . . . 424
24.8.4 Installing New Software. . . . 425
24.8.5 Installing New Microcode. . . . 426

25. Processes. . . . 428
25.1 The Scheduler. . . . 429
25.2 Locks . . . 432
25.3 Creating a Process . . . 433
25.4 Process Messages . . . 435
25.4.1 Process Attributes. . . . 435
25.4.2 Run and Arrest Reasons. . . . 436
25.4.3 Bashing the Process . . . 437
25.5 Process Flavors . . . 438
25.6 Other Process Functions . . . 439

26. Errors and Debugging . . . 440
26.1 The Error System . . . 440
26.1.1 Conditions . . . 440
26.1.2 Error Conditions . . . 442
26.1.3 Signalling Errors . . . 444
26.1.4 Standard Condition Names . . . 448
26.1.5 Errset. . . . 449
26.2 The Debugger . . . 450
26.2.1 Entering the Debugger . . . 450
26.2.2 How to Use the Debugger. . . . 451
26.2.3 Debugger Commands. . . . 452
26.2.4 Summary of Commands. . . . 455
26.3 Tracing Function Execution . . . 456
26.4 Advising a Function . . . 460
26.4.1 Designing the Advice . . . 462
26.4.2 :around Advice . . . 462
26.4.3 Advising One Function Within Another. . . . 463
26.5 Stepping Through an Evaluation . . . 464
26.6 Evalhook . . . 466
26.7 The MAR . . . 467
26.8 Variable Monitoring . . . 468

27. How to Read Assembly Language . . . 469
27.1 Introduction. . . . 469
27.2 A More Advanced Example . . . 472
27.3 The Rest of the Instructions . . . 475
27.4 Function Entry . . . 478
27.5 Special Class IV Instructions . . . 480
27.6 Estimating Run Time . . . 484

28. Querying the User . . . 487

29. Initializations 490
29.1 System Initialization Lists 492

30. Dates and Times. 493
30.1 Getting the Time 494
30.1.1 Elapsed Time in 60ths of a Second 494
30.1.2 Elapsed Time in Microseconds 494
30.2 Printing Dates and Times 495
30.3 Reading Dates and Times 496
30.4 Time Conversions. 497
30.5 Internal Functions. 497

31. Miscellaneous Useful Functions 499
31.1 Poking Around in the Lisp World 499
31.2 Utility Programs. 501
31.3 The Lisp Top Level 503
31.4 The Garbage Collector 505
31.5 Logging In 506
31.6 Dribble Files 507
31.7 Status and SStatus. 507

Concept Index. 509
Flavor Index. 512
Message Index. 513
Keyword Index 515
Object Creation Options. 518
Meter Index. 519
Variable Index. 520
Function Index 523

1. Introduction

1.1 General Information

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

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

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

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

1.2 Structure of the Manual

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

The next several chapters explain the details of the various primitive data-types of the language, and the functions that deal with them. Chapter 5 deals with conses and the higher-level structures that can be built out of them, such as trees, lists, association lists, and property lists. Chapter 6 deals with symbols, chapter 7 with the various kinds of numbers, and chapter 8 with arrays. Chapter 9 explains character strings, which are a special kind of array.

After this there are some chapters that explain more about functions, function-calling, and related matters. Chapter 10 presents all the kinds of functions in the language, explains function-specs, and tells how to manipulate definitions of functions. Chapters 11 and 12 discuss closures and stack-groups, two facilities useful for creating coroutines and other advanced control and access structures.

Next, a few lower-level issues are dealt with. Chapter 13 explains locatives, which are a kind of pointer to memory cells. Chapter 14 explains the "subprimitive" functions, which are primarily useful for implementation of the Lisp language itself and the Lisp Machine's "operating system". Chapter 15 discusses areas, which give you control over storage allocation and locality of reference.

Chapter 16 discusses the Lisp compiler, which converts Lisp programs into "machine language". Chapter 17 explains the Lisp macro facility, which allows users to write their own extensions to Lisp, extending both the interpreter and the compiler. The next two chapters go into detail about two such extensions, one that provides a powerful iteration control structure (chapter 18), and one that provides a powerful data structure facility (chapter 19).

Chapter 20 documents flavors, a language facility to provide generic functions using the paradigm used in Smalltalk and the Actor families of languages, called "object-oriented programming" or "message passing". Flavors are widely used by the system programs of the Lisp Machine, as well as being available to the user as a language feature.

Chapter 21 explains the Lisp Machine's Input/Output system, including *streams* and the *printed representation* of Lisp objects. Chapter 22 documents how to deal with pathnames (the names of files).

Chapter 23 describes the *package* system, which allows many name spaces within a single Lisp environment. Chapter 24 documents the "system" facility, which helps you create and maintain programs that reside in many files.

Chapter 25 discusses the facilities for multiple processes and how to write programs that use concurrent computation. Chapter 26 explains how exceptional conditions (errors) can be handled by programs, handled by users, and debugged. Chapter 27 explains the instruction set of the Lisp Machine, and tells you how to examine the output of the compiler. Chapter 28 documents some functions for querying the user, chapter 30 explains some functions for manipulating dates and times, and chapter 31 contains other miscellaneous functions and facilities.

1.3 Notational Conventions and Helpful Notes

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

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

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

A typical description of a Lisp function looks like this:

function-name *arg1 arg2* &optional *arg3* (*arg4* (foo 3))

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

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

Note the use of fonts (typefaces). The name of the function is in bold-face in the first line of the description, and the arguments are in italics. Within the text, printed representations of Lisp objects are in a different bold-face font, such as (+ foo 56), and argument references are italicized, such as *arg1* and *arg2*. A different, fixed-width font, such as `function-name`, is used for Lisp examples that are set off from the text.

The word "&optional" in the list of arguments tells you that all of the arguments past this point are optional. The default value can be specified explicitly, as with *arg4* whose default value is the result of evaluating the form (foo 3). If no default value is specified, it is the symbol nil. This syntax is used in lambda-lists in the language, which are explained in section 3.2, page 20. Argument lists may also contain "&rest", which is part of the same syntax.

The descriptions of special forms and macros look like this:

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

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

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

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

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

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

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

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

This ridiculous special form exhibits all of the syntactic features:

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

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

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

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

message-name *arg1 arg2* &optional *arg3* (to flavor-name)

This is the documentation of the effect of sending a message named message-name, with arguments *arg1*, *arg2*, and *arg3*, to an instance of flavor flavor-name.

Descriptions of variables ("special" or "global" variables) look like this:

typical-variable *Variable*

The variable typical-variable has a typical value....

Most numbers shown are in octal radix (base eight). Spelled out numbers and numbers followed by a decimal point are in decimal. This is because, by default, Zetalisp types out numbers in base 8; don't be surprised by this. If you wish to change it, see the documentation on the variables ibase and base (page 322).

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

There are several terms which are used widely in other references on Lisp, but are not used much in this document since they have become largely obsolete and misleading. For the benefit of those who may have seen them before, they are: "S-expression", which means a Lisp object; "Dotted pair", which means a cons; and "Atom", which means, roughly, symbols and numbers and sometimes other things, but not conses. The terms "list" and "tree" are defined in chapter 5, page 52.

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

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

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

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

The following characters also have special meanings, and may not be used in symbols without slashification. These characters are explained in detail in the section on printed-representation (section 21.2.2, page 322).

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

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

You will see various symbols that have the colon (:) character in their names. By convention, all "keyword" symbols in the Lisp Machine system have names starting with a colon. The colon character is not actually part of the print name, but is a package prefix indicating that the symbol belongs to the package with a null name, which means the user package. So, when you print such a symbol, you won't see the colon if the current package is user. However, you should always type in the colons where the manual tells you to. This is all explained in chapter 23; until you read that, just make believe that the colons are part of the names of the symbols, and don't worry that they sometimes don't get printed out for keyword symbols.

This manual documents a number of internal functions and variables, which can be identified by the "si:" prefix in their names. The "si" stands for "system internals". These functions and variables are documented here because they are things you sometimes need to know about.

However, they are considered internal to the system and their behavior is not as guaranteed as that of everything else. They may be changed in the future.

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

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

When the text speaks of "typing Control-Q" (for example), this means to hold down the CTRL key on the keyboard (either of the two), and, while holding it down, to strike the "Q" key. Similarly, to type "Meta-P", hold down either of the META keys and strike "P". To type "Control-Meta-T" hold down both CTRL and META. Unlike ASCII, there are no "control characters" in the character set; Control and Meta are merely things that can be typed on the keyboard.

Many of the functions refer to "areas". The *area* feature is only of interest to writers of large systems, and can be safely disregarded by the casual user. It is described in chapter 15.

2. Primitive Object Types

2.1 Data Types

This section enumerates some of the various different primitive types of objects in Zetalisp. The types explained below include symbols, conses, various types of numbers, two kinds of compiled code objects, locatives, arrays, stack groups, and closures. With each is given the associated symbolic name, which is returned by the function data-type (page 173).

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

The print name is a string, which may be obtained by the function get-pname (page 89). This string serves as the *printed representation* (see section 21.2.1, page 319) of the symbol. Each symbol has a *binding* (sometimes also called the "value"), which may be any Lisp object. It is also referred to sometimes as the "contents of the value cell", since internally every symbol has a cell called the *value cell* which holds the binding. It is accessed by the symeval function (page 86), and updated by the set function (page 86). (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* which holds the definition. The definition can be accessed by the fsymeval function (page 87), and updated with fset (page 87), although usually the functions fdefinition and fdefine are employed (page 149). The property list is a list of an even number of elements; it can be accessed directly by plist (page 88), and updated directly by setplist (page 88), although usually the functions get, putprop, and remprop (page 72) are used. The property list is used to associate any number of additional attributes with a symbol—attributes not used frequently enough to deserve their own cells as the value and definition do. Symbols also have a package cell, which indicates which "package" of names the symbol belongs to. This is explained further in the section on packages (chapter 23) and can be disregarded by the casual user.

The primitive function for creating symbols is make-symbol (page 90), although most symbols are created by read, intern, or fasload (which call make-symbol themselves.)

A *cons* is an object that cares about two other objects, arbitrarily named the *car* and the *cdr*. These objects can be accessed with car and cdr (page 53), and updated with rplaca and rplacd (page 61). The primitive function for creating conses is cons (page 53).

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 chapter 7, page 92 for full details of these types and the conversions between them.

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 (chapter 16, page 197), and are usually found as the definitions of symbols. The printed representation of a FEF includes its name, so that it can be identified.

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

About the only useful thing to do with any of these compiled code objects is to *apply* it to arguments. However, some functions are provided for examining such objects, for user convenience. See arglist (page 150), args-info (page 151), describe (page 500), and disassemble (page 500).

A *locative* (see chapter 13, page 170) is a kind of a pointer to a single memory cell anywhere in the system. The contents of this cell can be accessed by cdr (see page 53) and updated by rplacd (see page 62).

An *array* (see chapter 8, page 107) 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 which may contain any object, while others (numeric arrays) may only contain small positive numbers. Strings are a type of array; the elements are 8-bit unsigned numbers which encode characters.

A *list* is not a primitive data type, but rather a data structure made up out of conses and the symbol nil. See chapter 5, page 52.

2.2 Predicates

A *predicate* is a function which tests for some condition involving its arguments and returns the symbol t if the condition is true, or the symbol nil if it is not true. 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*
symbolp returns t if its argument is a symbol, otherwise nil.

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

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

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

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

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

floatp *arg*
floatp returns t if its argument is a floating-point number, i.e. a flonum or a small flonum, otherwise nil.

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

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

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

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

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

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

functionp *arg* &optional *allow-special-forms*
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 (see page 144), and symbols whose function definition is functionp. functionp is not true of objects which 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*
: **subrp** returns **t** if its argument is any compiled code object, otherwise **nil**. The Lisp Machine system doesn't use the term "subr", but the name of this function comes from Maclisp.

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

entityp *arg*
: **entityp** returns **t** if its argument is an entity, otherwise **nil**. See section 11.4, page 162 for information about "entities".

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

typep *arg* &optional *type*
: **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 which 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	*arg* is a flonum (not a small-flonum).
:small-flonum	*arg* is a small flonum.
:list	*arg* is a cons.
:locative	*arg* is a locative pointer (see chapter 13, page 170).
: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 chapter 11, page 158).
:select-method	*arg* is a select-method table (see page 144).
:stack-group	*arg* is a stack-group (see chapter 12, page 163).

: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 Named Structures, page 271, and Flavors, chapter 20, page 279.

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 chapter 20, page 279.
:entity	An entity. **typep** of one argument returns the name of the particular user-defined type of the entity, rather than **:entity**.

See also **data-type**, page 173.

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*

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

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

Note that in Zetalisp equal fixnums are **eq**; this is not true in Maclisp. Equality does not imply **eq**-ness for other types of numbers. To compare numbers, use =; see page 95.

neq *x y*

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

equal *x y*

The equal predicate returns t if its arguments are similar (isomorphic) objects. (cf. eq) Two numbers are equal if they have the same value 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 string-equal, page 128. Alphabetic case is ignored (but see alphabetic-case-affects-string-comparison, page 127). 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*
null *x*

not returns t if *x* is nil, else nil. null is the same as not; both functions are included for the sake of clarity. Use null to check whether something is nil; use not to invert the sense of a logical value. Even though Lisp uses the symbol nil to represent falseness, you shouldn't make understanding of your program depend on this fortuitously. For example, one often writes:

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

rather than

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

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

3. Evaluation

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

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

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

If *form* is a symbol, the result is the binding of *form*. If *form* is unbound, an error is signalled. The way symbols are bound is explained in section 3.1, page 14 below.

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 plain-old *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 (see chapter 6, page 86) and starts all over as if that object had been the car of the list. If the car isn't a symbol, then if it's a cons whose car is the symbol macro, then this is a macro form; if it is a "special function" (see page 141) then this is a special form; 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. All of them are documented in this manual. The internal workings of special forms are explained in more detail on page 141, but this hardly ever affects you.

If *form* is a macro form, then the macro is expanded as explained in chapter 17.

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. The way variables work and the ways in which they are manipulated, including the binding of arguments, is explained in section 3.1, page 14. A basic explanation of functions is in section 3.2, page 20. The way functions can return more than one value is explained in section 3.4, page 29. The description of all of the kinds of functions, and the means by which they are manipulated, is in chapter 10. Macros are explained in chapter 17. The evalhook facility, which lets you do something arbitrary whenever the evaluator is invoked, is explained in section 26.6, page 466. Special forms are described all over the manual; each special form is in the section on the facility it is part of.

3.1 Variables

In Zetalisp, variables are implemented using symbols. Symbols are used for many things in the language, such as naming functions, naming special forms, and being keywords; they are also useful to programs written in Lisp, as parts of data structures. But when 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 doesn't 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 non-local exit of any kind, such as *throw (see page 48) or return (see page 45), 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 page 38). 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. This is explained in the next section.

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 of 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 which are associated with symbols (the kind which are used by non-compiled 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, i.e. a new local variable.

Thus, if you compile a function, it 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 doesn't 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 can't use a local variable without first binding it. Another way to say this is that you can't ever have a free reference to a local variable. If you try to do so, the compiler will complain. In order for our 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** (see below), or by explicit compiler declarations (see page 201). 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 a variable or of many variables. The first *value* is evaluated, and the first *variable* is set to the result. Then the second *value* is evaluated, the second *variable* is set to the result, and so on for all the variable/value pairs. **setq** returns the last value, i.e. the result of the evaluation of its last subform.
Example:

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

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

psetq {*variable value*}... *Special Form*

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

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

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

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

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

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

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

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

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

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

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

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

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

let-if is a variant of let in which the binding of variables is conditional. The 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*, etc. are evaluated and then the variables *var-1*, *var-2*, etc. are bound to them. If the result is nil, the *vars* and *vals* are ignored. Finally the body forms are evaluated.

let-globally ((*var value*)...) *body*... *Special Form*

let-globally is similar in form to let (see page 16). The difference is that let-globally does not *bind* the variables; instead, it saves the old values and *sets* the variables, and sets up an unwind-protect (see page 49) to set them back. The important difference between let-globally and let is that when the current stack group (see chapter 12, page 163) co-calls some other stack group, the old values of the variables are *not* restored.

Thus **let-globally** makes the new values visible in all stack groups and processes that don't bind the variables themselves, not just the current stack group.

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*** (page 505); **break** implements this by using **progw**.

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 which 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 (see section 24.7, page 416) 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.

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.

3.2 Functions

In the description of evaluation on page 13, we said that evaluation of a function form works by applying the function to the results of evaluating the argument subforms. What is a function, and what does it mean to apply it? In Zetalisp there are many kinds of functions, and applying them may do many different kinds of things. For full details, see chapter 10, page 136. Here we will explain the most basic kinds of functions and how they work. In particular, this section explains *lambda lists* and all their important features.

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

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

The first element of the lambda-expression is the symbol lambda; the second element is a list called the *lambda list*, and the rest of the elements are called the *body*. The lambda list, in its simplest form, is just a list of variables. Assuming that this simple form is being used, here is what happens when a lambda expression is applied to some arguments. First, the number of arguments and the number of variables in the lambda list must be the same, or else an error is 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 may sound something like the description of let, above. The most important difference is that the lambda-expression is not a form at all; if you try to evaluate a lambda-expression, you will get 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 (e.g. (arg 3)). (For compatibility reasons, Zetalisp supports *lexpr*s, but they should not be used in new programs).

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

Here is the exact explanation of how this all works. When apply (the primitive function that applies functions to arguments) matches up the arguments with the parameters, it follows the following algorithm:

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

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

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

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

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

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

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

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

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

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

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

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

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

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 didn't 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 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, preceeded by the &-keyword &aux. Examples:

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

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

It is important to realize that the list of arguments to which a rest-parameter is bound is 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 (see copylist, page page 58), as you must always assume that it is one of these special lists. 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 may 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 section 10.7, page 148 for a complete list. You only need to know about &optional and &rest in order to understand this manual.

Lambda lists provide "positional" arguments: the meaning of an argument comes from its position in the lambda list. For example, the first argument to cons is the object that will be the car of the new cons. Sometimes it is desirable to use "keyword" arguments, in which the meaning of an argument comes from a "keyword" symbol that tells the callee which argument this is. While lambda lists do not provide keyword arguments directly, there is a convention for functions that want arguments passed to them in the keyword fashion. The convention is that the function takes a rest-argument, whose value is a list of alternating keyword symbols and argument values. If cons were written as a keyword-style function, then instead of saying

```
        (cons 4 (foo))
```

you could say either of

```
        (cons ':car 4 ':cdr (foo))
        or
        (cons ':cdr (foo) ':car 4)
```

assuming the keyword symbols were :car and :cdr. Keyword symbols are always in the keyword package, and so their printed representations always start with a colon; the reason for this is given in chapter 23.

This use of keyword arguments is only a convention; it is not built into the function-calling mechanism of the language. Your function must contain Lisp programming to take apart the rest parameter and make sense of the keywords and values. The special form keyword-extract (see page 42) may be useful for this.

3.3 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*

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

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

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

Also, if you are only interested in getting at the value of a symbol (that is, the contents of the symbol's value cell), then you should use the primitive function symeval (see page 86).

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 chapter 11, page 158).

apply *f arglist*

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

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

Of course, *arglist* may be nil.

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*

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

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

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

lexpr-funcall *f* &rest *args*

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**.

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*

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

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

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

Example:
```
        (call #'foo () x ':spread y '(:optional :spread) z () w)
```
The arguments passed to **foo** are the value of x, the elements of the value of y, the elements of the value of z, and the value of w. The function **foo** must be prepared to accept all the arguments which come from x and y, but if it does not want the rest, they are ignored.

quote *object* *Special Form*

(quote *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.

For 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 symbol or a list. (Note that in neither case is *f* evaluated.)

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's what the compiler does with a **function** special form whose subform *f* is not a symbol. 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 a symbol, then **function** returns the definition (contents of the function cell location) of *f*; it is like **fsymeval** except that it is a special form instead of a function, and so
```
    (function fred)  is like  (fsymeval 'fred)
```
function is the same for the compiler and the interpreter when *f* is a symbol.

Another way of explaining **function** is that it causes *f* to be treated the same way as it would as the car of a form. Evaluating the form (*f arg1 arg2*...) uses the function definition of *f* if it is a symbol, and otherwise expects *f* to be a list which is a lambda-expression.

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 (see page 457), or advised (see page 460). 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 section 10.7, page 148. 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

Takes no arguments and returns nil.

true

Takes no arguments and returns t.

ignore &rest *ignore*

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

comment *Special Form*

comment ignores its form and returns the symbol comment.
Example:

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

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

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

A problem with such comments is that they are discarded when the 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))
```

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

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

Example:
```
    (setq x (prog1 y (setq y x)))
```
interchanges the values of the variables *x* and *y*. prog1 never returns multiple values.

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

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

See also bind (page 183), 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 section 3.2, page 20).

arg *x*

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

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

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

setarg *i x*

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

listify *n*

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

```
(defun listify (n)
     (cond ((minusp n)
             (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) ))
```

3.4 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, and multiple-value-list. 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 don't 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*

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

values-list *list*

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

return and its variants can only be used within the do and prog special forms and their variants, and so they are explained on page 45.

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

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

multiple-value-list evaluates *form*, and returns a list of the values it returned. This is useful for when you don't know how many values to expect.

Example:

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

This is similar to the example of **multiple-value** above; **a** will be set to a list of three elements, the three values returned by **intern**.

Due to the syntactic structure of Lisp, it is often the case that the value of a certain form is the value of a sub-form 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 sub-form 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 sub-form 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 may change in the future or in other implementations. The reason we don't guarantee non-transmission of multiple values is because such a guarantee would not be very useful and the efficiency cost of enforcing it would be high. Even setq'ing a variable to the result of a form, then returning the value of that variable might be made to pass multiple values by an optimizing compiler which realized that the setqing 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 may or may 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**, etc., 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.

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 sub-form 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 (i.e. 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 shouldn't 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 (see page 47) 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. This is inelegant, but we don't know anything that's much better.

4. Flow of Control

Lisp provides a variety of structures for flow of control.

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

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

The performing of an action repeatedly (usually with some changes between repetitions) is called *iteration*, and is provided as a basic control structure in most languages. The *do* statement of PL/I, the *for* statement of ALGOL/60, and so on are examples of iteration primitives. Lisp provides two general iteration facilities: **do** and **loop**, as well as a variety of special-purpose iteration facilities. (**loop** is sufficiently complex that it is explained in its own chapter later in the manual; see page 233.) There is also a very general construct to allow the traditional "goto" control structure, called **prog**.

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

There are some *non-local exit* control structures, analogous to the *leave*, *exit*, and *escape* constructs in many modern languages. The general ones are ***catch** and ***throw**; there is also **return** and its variants, used for exiting iteration the constructs **do**, **loop**, and **prog**.

Zetalisp also provides a coroutine capability, explained in the section on *stack-groups* (chapter 12, page 163), and a multiple-process facility (see chapter 25, page 428). There is also a facility for generic function calling using message passing; see chapter 20, page 279.

4.1 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 consistutes 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 which is selected if its antecedent is satisfied and the antecedents of all preceding clauses were not satisfied. When a clause is selected, its consequent forms are evaluated.

cond processes its clauses in order from left to right. First, the antecedent of the current clause is evaluated. If the result is nil, cond advances to the next clause. Otherwise, the cdr of the clause is treated as a list consequent forms which are evaluated in order from left to right. After evaluating the consequents, cond returns without inspecting any remaining clauses. The value of the cond special form is the value of the last consequent evaluated, or the value of the antecedent if there were no consequents in the clause. If cond runs out of clauses, that is, if every antecedent 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.

selectq *Special Form*

selectq is a conditional which 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:

1) A symbol	If the *key* is eq to the symbol, it matches.
2) A number	If the *key* is eq to the number, it matches. Only small numbers (*fixnums*) will work.
3) A list	If the *key* is eq to one of the elements of the list, then it matches. The elements of the list should be symbols or fixnums.
4) t or otherwise	The symbols t and otherwise are special keywords which match anything. Either symbol may be used, it makes no difference; t is 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 (page 147), 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))))
```

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. Byte specifiers and ldb are explained on page 102.
Example:
```
      (princ (dispatch 0202 cat-type
                 (0 "Siamese.")
                 (1 "Persian.")
                 (2 "Alley.")
                 (3 (ferror nil
                            "~S is not a known cat type."
                            cat-type))))
```
It is not necessary to include all possible values of the byte which will be dispatched on.

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 don't use the symbol otherwise as the key.

4.2 Iteration

do *Special Form*

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

do comes in two varieties.

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, i.e. lexically *outside* of the do. At the beginning of each succeeding iteration those *vars* that have *repeat* forms get set to the values of their respective *repeat* forms. Note that all the *repeat* forms are evaluated before any of the *vars* is set.

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

Note that the *end-test* gets evaluated before the first time the body is evaluated. do first initializes the variables from the *init* forms, then it checks the *end-test*, then it processes the body, then it deals with the *repeat* forms, then it tests the *end-test* again, and so on. If the end-test returns a non-nil value the first time, then the body 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 page 45 for more details about return and its variants. go special forms (see page 45) 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 re-evaluated each time. Note that the *init* form is evaluated before *var* is bound, i.e. lexically *outside* of the do. Each time around the loop, after *var* is set, *end-test* is evaluated. If it is non-nil, the do finishes and returns nil. If the *end-test* evaluated to nil, the *body* of the loop is executed. As with the new-style do, return and go may be used in the body, and they have the same meaning.

Examples of the older variety of do:

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

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

return forms are often useful to do simple searches:

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

Examples of the new form of do:

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

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

is like `(maplist 'f x y)` (see page 50).

Also see **loop** (page 233), 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** (see page 46), 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, then it will be made "invisible" to **returns**; that is, **returns** inside that **do-named** will return to the next outermost level whose name is not t. (**return-from** t ...) will return from a **do-named** named 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 **loops** can have names just as **dos** can. Since the same functions are used to return from all of these forms, all of these names are in the same name-space; 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.

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, etc. 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 zero rather than one, 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 which 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 which 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**.)

keywords defines the symbols which are keywords to be followed by an argument. Each element of *keywords* is either the name of a local variable which 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 which 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 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*.

prog *Special Form*

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

```
(prog (var1 var2 (var3 init3) var4 (var5 init5))
 tag1
      statement1
      statement2
 tag2
      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*** (see page 45) 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. The programmer is recommended to 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 (see page 46) can be used to return from it. See do-named, page 41.

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* *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**.

go *tag* *Special Form*

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

Example:

```
(prog (x y z)
  (setq x somefrob)
loop
  do something
  (if somepredicate (go endtag))
  do something more
  (if (minusp x) (go loop))
endtag
  (return z))
```

return *value...* *Special Form*

return is used to exit from a **prog**-like special form (**prog**, **prog***, **do**, **do-named**, **dotimes**, **dolist**, **loop**, etc.) The *value* forms are evaluated, and the resulting values are returned by the **prog** as its values.

In addition, **break** (see page 504) recognizes the typed-in form (**return** *value*) specially. If this form is typed at a **break**, *value* will be evaluated and returned as the value of **break**. If not specially recognized by **break**, and not inside a **prog**-like form, **return** will cause an error.

Example:

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

Note that the **return** form is very unusual: it does not ever return a value itself, in the conventional sense. It isn't useful to write (setq a (return 3)), because when the **return** form is evaluated, the containing **do** or **prog** is immediately exited, and the **setq** never happens. A **return** form may not appear as an argument to a regular function, but only at the top level of a **prog** or **do**, or within certain special forms such as conditionals which are within a prog or do. A return as an argument to a regular function would be not only useless but possibly meaningless. The compiler does not bother to know how to compile it correctly in all cases. The same is true of go.

return can also be used with multiple arguments, to return multiple values from a prog or do. For example,

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

This function is like assq, but it returns an additional value which is the index in the table of the entry it found.

However, if you use return with only one subform, then the prog or do will return all of the values returned by that subform. That is, if you do

```
(prog ()
   ...
   (return (foo 2)))
```

and the function foo returns many values, then the prog will return all of those values. In fact, this means that

(return (values *form1 form2 form3*))
is the same as
(return *form1 form2 form3*)

It is legal to write simply (return), which will return from the prog without returning any values.

See section 3.4, page 29 for more information.

return-from *name value...* *Special Form*

The *value* forms are evaluated, and then are returned from the innermost containing prog-like special form whose name is *name*. See the description of do-named (page 41) in which named dos and progs are explained.

return-list *list*

This function is like return except that the prog returns all of the elements of *list*; if *list* has more than one element, the prog does a multiple-value return.

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

Also see defunp (page 140), a variant of defun that incorporates a prog into the function body.

4.3 Non-Local Exits

***catch** *tag body...* *Special Form*

*catch is a special form used with the *throw function to do non-local exits. First *tag* is evaluated; the result is called the "tag" of the *catch. Then the *body* forms are evaluated sequentially, and the value of the last form is returned. However, if, during the evaluation of the body, the function *throw is called with the same tag as the tag of the *catch, then the evaluation of the body is aborted, and the *catch form immediately returns the value that was the second argument to *throw without further evaluating the current *body* form or the rest of the body.

The *tag*'s are used to match up *throw's with *catch's. (*catch 'foo *form*) will catch a (*throw 'foo *form*) but not a (*throw 'bar *form*). It is an error if *throw is done when there is no suitable *catch (or catch-all; see below).

The values t and nil for *tag* are special: a *catch whose tag is one of these values will catch throws to any tag. These are only for internal use by unwind-protect and catch-all respectively. The only difference between t and nil is in the error checking; t implies that after a "cleanup handler" is executed control will be thrown again to the same tag, therefore it is an error if a specific catch for this tag does not exist higher up in the stack. With nil, the error check isn't done.

*catch returns up to four values; trailing null values are not returned for reasons of microcode simplicity, but the values not returned will default to nil if they are received with the multiple-value or multiple-value-bind special forms. If the catch completes normally, the first value is the value of *form* and the second is nil. If a *throw occurs, the first value is the second argument to *throw, and the second value is the first argument to *throw, the tag thrown to. The third and fourth values are the third and fourth arguments to *unwind-stack (see page 48) if that was used in place of *throw; otherwise these values are nil. To summarize, the four values returned by *catch are the value, the tag, the active-frame-count, and the action.
Example

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

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

Note that *catch returns its own extra values, and so it does *not* propagate multiple values back from the last form.

***throw** *tag value*

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

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

The values t, nil, and 0 for *tag* are reserved and used for internal purposes. nil may not be used, because it would cause an ambiguity in the returned values of *catch. t may only be used with *unwind-stack. 0 and nil are used internally when returning out of an unwind-protect.

See the description of *catch for further details.

catch *form tag* *Macro*
throw *form tag* *Macro*

catch and throw are provided only for Maclisp compatibility. (catch *form* tag) is the same as (*catch 'tag *form*), and (throw *form* tag) is the same as (*throw 'tag *form*). The forms of catch and throw without tags are not supported.

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

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

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

A *tag* of t invokes a special feature whereby the entire stack is unwound, and then the function *action* is called (see below). During this process unwind-protects receive control, but catch-alls do not. This feature is provided for the benefit of system programs which want to unwind a stack completely.

active-frame-count, if non-nil, is the number of frames to be unwound. The definition of a "frame" is implementation-dependent. If this counts down to zero before a suitable *catch is found, the *unwind-stack terminates and *that frame* returns *value* to whoever called it. This is similar to Maclisp's freturn function.

If *action* is non-nil, whenever the *unwind-stack would be ready to terminate (either due to *active-frame-count* or due to *tag* being caught as in *throw), instead *action* is called with one argument, *value*. If *tag* is t, meaning throw out the whole way, then the function *action* is not allowed to return. Otherwise the function *action* may return and its value will be returned instead of *value* from the *catch—or from an arbitrary function if *active-frame-count* is in use. In this case the *catch does not return multiple values as it normally does when thrown to. Note that it is often useful for *action* to be a stack-group.

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

unwind-protect *protected-form cleanup-form...* *Special Form*

Sometimes it is necessary to evaluate a form and make sure that certain side-effects take place after the form is evaluated; a typical example is:

```
(progn
   (turn-on-water-faucet)
   (hairy-function 3 nil 'foo)
   (turn-off-water-faucet))
```

The non-local exit facility of Lisp creates a situation in which the above code won't work, however: if hairy-function should do a *throw to a *catch which is outside of the progn form, then (turn-off-water-faucet) will never be evaluated (and the faucet will presumably be left running). This is particularly likely if hairy-function gets an error and the user tells the error-handler 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 which attempts to quit out of the evaluation of the unwind-protect, the (turn-off-water-faucet) form will be evaluated in between the time of the *throw and the time at which the *catch returns. If the progn returns normally, then the (turn-off-water-faucet) is evaluated, and the unwind-protect returns the result of the progn.

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. 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

catch-all *body...* *Macro*

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

4.4 Mapping

map *fcn* &rest *lists*
mapc *fcn* &rest *lists*
maplist *fcn* &rest *lists*
mapcar *fcn* &rest *lists*
mapcon *fcn* &rest *lists*
mapcan *fcn* &rest *lists*

Mapping is a type of iteration in which a function is successively applied to pieces of a list. There are several options for the way in which the pieces of the list are chosen and for what is done with the results returned by the applications of the function.

For example, mapcar operates on successive *elements* of the list. As it goes down the list, it calls the function giving it an element of the list as its one argument: first the car, then the cadr, then the caddr, etc., continuing until the end of the list is reached. The value returned by mapcar is a list of the results of the successive calls to the function. An example of the use of mapcar would be mapcar'ing the function abs over the list (1 -2 -4.5 6.0e15 -4.2), which would be written as (mapcar (function abs) '(1 -2 -4.5 6.0e15 -4.2)). The result is (1 2 4.5 6.0e15 4.2).

In general, the mapping functions take any number of arguments. For example,

```
    (mapcar f x1 x2 ... xn)
```

In this case *f* must be a function of *n* arguments. mapcar will proceed down the lists *x1*, *x2*, ..., *xn* in parallel. The first argument to *f* will come from *x1*, the second from *x2*, etc. The iteration stops as soon as any of the lists is exhausted. (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.)

There are five other mapping functions besides mapcar. maplist is like mapcar except that the function is applied to the list and successive cdr's of that list rather than to successive elements of the list. map and mapc are like maplist and mapcar respectively, except that they don't return any useful value. These functions are used when the function is being called merely for its side-effects, rather than its returned values. mapcan and mapcon are like mapcar and maplist respectively, except that they combine the results of the function using nconc instead of list. That is, mapcon could have been defined by

```
    (defun mapcon (f x y)
        (apply 'nconc (maplist f x y)))
```

Of course, this definition is less general than the real one.

Sometimes a do or a straightforward recursion is preferable to a map; however, the mapping functions should be used wherever they naturally apply because this increases the clarity of the code.

Often *f* 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
	its own second argument	map	mapc
returns	list of the function results	maplist	mapcar
	nconc of the function results	mapcon	mapcan

There are also functions (mapatoms and mapatoms-all) for mapping over all symbols in certain packages. See the explanation of packages (chapter 23, page 392).

You can also do what the mapping functions do in a different way by using **loop**. See page 233.

5. Manipulating List Structure

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

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

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

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

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

A "dotted list" is like a list except that the cdr of the last cons does not have to be nil. This name comes from the printed representation, which includes a "dot" character. 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 (see page 335) is one of these functions; if you try to print a circular list the printer will never stop producing text. You have to be careful what you do with circular lists.

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. The cdr-coding scheme is discussed in section 5.4, page 63.

5.1 Conses

car *x*

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

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

cdr *x*

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

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

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

c...r *x*

All of the compositions of up to four *car*'s and *cdr*'s are defined as functions in their own right. The names of these functions begin with "c" and end with "r", and in between is a sequence of "a"'s and "d"'s corresponding to the composition performed by the function.
Example:

```
(cddadr x) is the same as (cdr (cdr (car (cdr x))))
```

The error checking for these functions is exactly the same as for car and cdr above.

cons *x y*

cons is the primitive function to create a new *cons*, whose *car* is *x* and whose *cdr* is *y*.
Examples:

```
(cons 'a 'b) => (a . b)
(cons 'a (cons 'b (cons 'c nil))) => (a b c)
(cons 'a '(b c d)) => (a b c d)
```

ncons *x*

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

xcons *x y*

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

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

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

This function creates a *cons* in a specific *area*. (Areas are an advanced feature of storage management, explained in chapter 15; if you aren't interested in them, you can safely skip all this stuff). The first two arguments are the same as the two arguments to **cons**, and the third is the number of the area in which to create the *cons*.
Example:

```
(cons-in-area 'a 'b my-area) => (a . b)
```

ncons-in-area *x area-number*

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

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

(xcons-in-area *x y area-number*) = (cons-in-area *y x area-number*)

The backquote reader macro facility is also generally useful for creating list structure, especially mostly-constant list structure, or forms constructed by plugging variables into a template. It is documented in the chapter on macros; see chapter 17, page 208.

car-location *cons*

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

Note: there is no cdr-location function; it is difficult because of the cdr-coding scheme (see section 5.4, page 63).

5.2 Lists

length *list*

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*
second *list*
third *list*
fourth *list*
fifth *list*
sixth *list*
seventh *list*

These functions take a list as an argument, and return the first, second, etc. element of the list. first is identical to car, second is identical to cadr, and so on. The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

rest1 *list*
rest2 *list*
rest3 *list*
rest4 *list*

rest*n* returns the rest of the elements of a list, starting with element *n* (counting the first element as the zeroth). Thus rest1 is identical to cdr, rest2 is identical to cddr, and so on. The reason these names are provided is that they make more sense when you are thinking of the argument as a list rather than just as a cons.

nth *n list*

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

Examples:

```
(nth 1 '(foo bar gack)) => bar
(nth 3 '(foo bar gack)) => nil
```

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*

(nthcdr *n list*) cdrs *list n* times, and returns the result.
Examples:
```
        (nthcdr 0 '(a b c)) => (a b c)
        (nthcdr 2 '(a b c)) => (c)
```
In other words, it returns the *n*'th cdr of the list. If *n* is greater than the length of the list, nil is returned.

This is similar to InterLisp's function nth, except that the InterLisp function is one-based instead of zero-based; see the InterLisp manual for details. nthcdr could have been defined by:
```
        (defun nthcdr (n list)
            (do ((i 0 (1+ i))
                 (list list (cdr list)))
                ((= i n) list)))
```

last *list*

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

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*

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*

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

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

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

make-list *length* &rest *options*

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 (see chapter 15, page 192) the list should be created. 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 section 5.4, page 63).
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*

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*

Returns a list which is equal to *list*, but not eq. copylist does not copy any elements of the list: only the conses of the list itself. The returned list is fully cdr-coded (see section 5.4, page 63) to minimize storage. 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*

This is the same as copylist except that the last cons of the resulting list is never cdr-coded (see section 5.4, page 63). This makes for increased efficiency if you nconc something onto the list later.

copyalist *list* &optional *area*

copyalist is for copying association lists (see section 5.5, page 65). The *list* is copied, as in copylist. In addition, each element of *list* which is a cons is replaced in the copy by a new cons with the same car and cdr. You may optionally specify the area in which to create the new copy.

copytree *tree*

copytree copies all the conses of a tree and makes a new tree with the same fringe.

reverse *list*

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

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

Example:

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

nreverse could have been defined by:

```
(defun nreverse  (x)
    (cond ((null x) nil)
          ((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 (see section 5.4, page 63) lists, because it just uses **rplacd** in the straightforward way. This may be fixed someday. In the meantime **reverse** might be preferable in some cases.

append &rest *lists*

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

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

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

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

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

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

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

These definitions do not express the full functionality of **append**; the real definition minimizes storage utilization by cdr-coding (see section 5.4, page 63) 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.

To copy a list, use **copylist** (see page 58); the old practice of using **append** to copy lists is unclear and obsolete.

nconc &rest *lists*

nconc takes lists as arguments. It returns a list which is the arguments concatenated together. The arguments are changed, rather than copied. (cf. **append**, page 59)

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*

(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*

This creates and returns a list with the same elements as *list*, excepting the last element.
Examples:
```
        (butlast '(a b c d)) => (a b c)
        (butlast '((a b) (c d))) => ((a b))
        (butlast '(a)) => nil
        (butlast nil) => nil
```
The name is from the phrase "all elements but the last"

nbutlast *list*

This is the destructive version of butlast; it changes the cdr of the second-to-last cons of the list to nil. If there is no second-to-last cons (that is, if the list has fewer than two elements) it returns nil.
Examples:
```
        (setq foo '(a b c d))
        (nbutlast foo) => (a b c)
        foo => (a b c)
        (nbutlast '(a)) => nil
```

firstn *n list*

firstn returns a list of length *n*, whose elements are the first *n* elements of list. If *list* is fewer than *n* elements long, the remaining elements of the returned list will be nil.

Example:
```
(firstn 2 '(a b c d)) => (a b)
(firstn 0 '(a b c d)) => nil
(firstn 6 '(a b c d)) => (a b c d nil nil)
```

nleft *n list* &optional *tail*

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

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.
```

5.3 Alteration of List Structure

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

The structure is not copied but is physically altered; hence caution should be exercised when using these functions, as strange side-effects can occur if portions of list structure become shared unbeknownst to the programmer. The nconc, nreverse, nreconc, and nbutlast functions already described, and the delq family described later, have the same property.

rplaca *x y*

(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*

(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*

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

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

subst could have been defined by:

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

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

To copy a tree, use copytree (see page 58); 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*

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*

sublis makes substitutions for symbols in a tree. The first argument to sublis is an association list (see section 5.5, page 65). 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 it is associated with. 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 ((atom sexp)
         (let ((tem (assq sexp alist)))
           (if tem (cdr tem) 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))))))
```

nsublis *alist tree*

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)))
```

5.4 Cdr-Coding

This section explains the internal data format used to store conses inside the Lisp Machine. Casual users don't have to worry about this; you can skip this section if you want. It is only important to read this section if you require extra storage efficiency in your program.

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

Very often, conses are used to store lists. If the above representation is used, a list of *n* elements requires two times *n* words of memory: *n* to hold the pointers to the elements of the list, and *n* to point to the next cons or to nil. To optimize this particular case of using conses, the Lisp Machine uses a storage representation called "cdr coding" to store lists. The basic goal is to allow a list of *n* elements to be stored in only *n* locations, while allowing conses that are not parts of lists to be stored in the usual way.

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

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

If you examine these rules, you will find that they work fine even if you mix the two kinds of storage representation within the same list. There's 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 doesn't exist.

This problem is dealt with by the use of "invisible pointers". An invisible pointer is a special kind of pointer, recognized by its data type (Lisp Machine pointers include a data type field as well as an address field). The way they work is that when the Lisp Machine reads a word from memory, if that word is an invisible pointer then it proceeds to read the word pointed to by the invisible pointer and use that word instead of the invisible pointer itself. Similarly, when it writes to a location, it first reads the location, and if it contains an invisible pointer then it writes to the location addressed by the invisible pointer instead. (This is a somewhat simplified explanation; actually there are several kinds of invisible pointer that are interpreted in different ways at different times, used for things other than the cdr coding scheme.)

Here's how to do 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 doesn't matter what the cdr-code of this word is, since the invisible pointer data type is checked first, as we will see.)

Now, whenever any operation is done to the cons (car, cdr, rplaca, or rplacd), the initial reading of the word pointed to by the Lisp pointer that represents the cons 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 a difference in efficiency. The compact representation is more efficient in most cases. However, if the conses are going to get rplacd'ed, then invisible pointers will be created, extra memory will be allocated, and the compact representation will 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 simple-mindedly uses **rplacd**, but this will be changed.

(**copylist** *x*) is a suitable way to copy a list, converting it into compact form (see page 58).

5.5 Tables

Zetalisp includes functions which simplify the maintenance of tabular data structures of several varieties. The simplest is a plain list of items, 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 chapter 19, page 257).

Simple list-structure is very convenient, but may not be efficient enough for large data bases because it takes a long time to search a long list. Zetalisp includes hash table facilities for more efficient but more complex tables (see section 5.9, page 74), and a hashing function (sxhash) to aid users in constructing their own facilities.

5.6 Lists as Tables

memq *item list*

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

Examples:

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

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

Example:

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

memq could have been defined by:

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

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

member *item list*

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

member could have been defined by:

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

mem *predicate item list*

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

mem is usually used with equality predicates other than eq and equal, such as =, char-equal or string-equal. It can also be used with non-commutative 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 or equal to 4.

find-position-in-list *item list*

find-position-in-list looks down *list* for an element which is eq to *item*, like memq. However, it returns the numeric index in the list at which it found the first occurence of *item*, or nil if it did not find it at all. This function is sort of the complement of nth (see page 55); like nth, it is zero-based.
Examples:

```
(find-position-in-list 'a '(a b c)) => 0
(find-position-in-list 'c '(a b c)) => 2
(find-position-in-list 'e '(a b c)) => nil
```

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

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

tailp *sublist list*

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

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

delq *item list* &optional *n*

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

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

rather than

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

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

(delq *item list n*) is like (delq *item list*) except only the first *n* instances of *item* are deleted. *n* is allowed to be zero. If *n* is greater than or equal to the number of occurrences of *item* in the list, all occurrences of *item* in the list 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*
delete is the same as delq except that equal is used for the comparison instead of eq.

del *predicate item list* &optional *n*
del is the same as delq except that it takes an extra argument which should be a predicate of two arguments, which is used for the comparison instead of eq. (del 'eq a b) is the same as (delq a b). (cf. mem, page 67)

remq *item list* &optional *n*
remq is similar to delq, except that the list is not altered; rather, a new list is returned. Examples:
```
        (setq x '(a b c d e f))
        (remq 'b x) => (a c d e f)
        x => (a b c d e f)
        (remq 'b '(a b c b a b) 2) => (a c a b)
```

remove *item list* &optional *n*
remove is the same as remq except that equal is used for the comparison instead of eq.

rem *predicate item list* &optional *n*
rem is the same as remq except that it takes an extra argument which should be a predicate of two arguments, which is used for the comparison instead of eq. (rem 'eq a b) is the same as (remq a b). (cf. mem, page 67)

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

subset-not *predicate list*
rem-if *predicate list*
predicate should be a function of one argument. A new list is made by applying *predicate* to all of the elements of *list* and removing the ones for which the predicate returns non-nil. One of this function's names (rem-if) means "remove if this condition is true". The

other name (subset-not) refers to the function's action if *list* is considered to represent a mathematical set.

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

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

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

some *list predicate* &optional *step-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.

5.7 Association Lists

assq *item alist*
(assq *item alist*) looks up *item* in the association list (list of conses) *alist*. The value is the first cons whose car is eq to *x*, or nil if there is none such.
Examples:

```
(assq 'r '((a . b) (c . d) (r . x) (s . y) (r . z)))
        => (r . x)

(assq 'fooo '((foo . bar) (zoo . goo))) => nil

(assq 'b '((a b c) (b c d) (x y z))) => (b c d)
```

It is okay to rplacd the result of assq as long as it is not nil, if your intention is to "update" the "table" that was assq's second argument.
Example:

```
(setq values '((x . 100) (y . 200) (z . 50)))
(assq 'y values) => (y . 200)
(rplacd (assq 'y values) 201)
(assq 'y values) => (y . 201) now
```

A typical trick is to say (cdr (assq x y)). Since the cdr of nil is guaranteed to be nil, this yields nil if no pair is found (or if a pair is found whose cdr is nil.)

assq could have been defined by:

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

assoc *item alist*

assoc is like assq except that the comparison uses equal instead of eq.
Example:

```
(assoc '(a b) '((x . y) ((a b) . 7) ((c . d) .e)))
        => ((a b) . 7)
```

assoc could have been defined by:

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

ass *predicate item alist*

ass is the same as assq except that it takes an extra argument which should be a predicate of two arguments, which is used for the comparison instead of eq. (ass 'eq a b) is the same as (assq a b). (cf. mem, page 67) As with mem, you may use non-commutative predicates; the first argument to the predicate is *item* and the second is the key of the element of *alist*.

memass *predicate item alist*

memass searches *alist* just like ass, but returns the portion of the list beginning with the pair containing *item*, rather than the pair itself. (car (memass *x y z*)) = (ass *x y z*). (cf. mem, page 67) As with mem, you may use non-commutative predicates; the first argument to the predicate is *item* and the second is the key of the element of *alist*.

rassq *item alist*

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*

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*

rass is to rassq as ass is to assq. That is, it takes a predicate to be used instead of eq. (cf. mem, page 67) As with mem, you may use non-commutative predicates; the first argument to the predicate is *item* and the second is the cdr of the element of *alist*.

sassq *item alist fcn*

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

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

sassq and sassoc (see below) are of limited use. These are primarily leftovers from Lisp 1.5.

sassoc *item alist fcn*

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

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

pairlis *cars cdrs*

pairlis takes two lists and makes an association list which associates elements of the first list with corresponding elements of the second list.
Example:

```
(pairlis '(beef clams kitty) '(roast fried yu-shiang))
   => ((beef . roast) (clams . fried) (kitty . yu-shiang))
```

5.8 Property Lists

From time immemorial, Lisp has had a kind of tabular data structure called a *property list* (plist for short). A property list contains zero or more entries; each entry associates from a keyword symbol (called the *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 which alter the property-list rather than making a new one. An association list with no entries would be the empty list (), i.e. the symbol nil. There is only one empty list, so all empty association lists are the same object. Each empty property-list is a separate and distinct object.

The implementation of a property list is a memory cell containing a list with an even number (possibly zero) of elements. Each pair of elements constitutes a *property*; the first of the pair is the 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; i.e. 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 available to the user to store something.

2. The system associates a property list with every symbol (see section 6.3, page 88). 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 chapter 13, page 170). Such locatives are typically created with the locf special form (see page 230).

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 which is being used by a program which deals with blocks:

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

There are three properties, and so the list has six elements. The first property's indicator is the symbol color, and its value is the symbol blue. One says that "the value of b1's color property is blue", or, informally, that "b1's color property is blue." The program is probably representing the information that the block represented by b1 is 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*

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*

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*

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 doesn't 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*

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.]

5.9 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. Hashing is explained in section 5.9.4, page 78.

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. In other words, there are hash tables which hash on Lisp *objects* (using eq) and there are hash tables which hash on trees (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 with the function make-hash-table, which takes various options. New entries are added to hash tables with the puthash function. To look up a key and find the associated value, the gethash function is used. To remove an entry, use remhash. Here is a simple example.

```
(setq a (make-hash-table))

(puthash 'color 'brown a)

(puthash 'name 'fred a)

(gethash 'color a) => brown

(gethash 'name a) => fred
```

In this example, the symbols color and name are being used as keys, and the symbols brown and fred are being used as the associated values. The hash table 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. This means that keys are really objects, but it means that it is not reasonable to use numbers other than fixnums as keys.

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 (see page 500) prints a variety of useful information when applied to a hash table.

This hash table facility is similar to the hasharray facility of Interlisp, and some of the function names are the same. However, it is *not* compatible. The exact details and the order of arguments are designed to be consistent with the rest of Zetalisp rather than with Interlisp. For instance, the order of arguments to maphash is different, we do not have the Interlisp "system hash table", and we do not have the Interlisp restriction that keys and values may not be nil. Note, however, that the order of arguments to gethash, puthash, and remhash is not consistent with the Zetalisp's get, putprop, and remprop, either. This is an unfortunate result of the haphazard historical development of Lisp.

If the calling program is using multiprocessing, it must be careful to make sure that there are never two processes both referencing the hash table at the same time. There is no locking built into hash tables; if you have two processes that both want to reference the same hash table, you must arrange mutual exclusion yourself by using a lock or some other means. Even two processes just doing gethash on the same hash table must synchronize themselves, because gethash may be forced by garbage collection to rehash the table. Don't worry about this if you don't use multiprocessing; but if you do use multiprocessing, you will have a lot of trouble if you don't understand this.

Hash tables are implemented with a special kind of array. arrayp of a hash table will return t. However, it is illegal to use normal array operations on a hash table, and in general they will not work. Hash tables should be manipulated only with the functions described below.

5.9.1 Hashing on Eq

This section documents the functions for eq hash tables, which use *objects* as keys and associate other objects with them.

make-hash-table &rest *options*

This creates a new hash table. Valid option keywords are:

:size
: Sets the initial size of the hash table, in entries, as a fixnum. The default is 100 (octal). The actual size is rounded up from the size you specify to the next size that is "good" for the hashing algorithm. You won't necessarily be able to store this many entries into the table before it overflows and becomes bigger; but except in the case of extreme bad luck you will be able to store almost this many.

:area
: Specifies the area in which the hash table should be created. This is just like the :area option to make-array (see page 111). Defaults to nil (i.e. default-cons-area).

:rehash-function
: Specifies the function to be used for rehashing when the table becomes full. Defaults to the internal rehashing function that does the usual thing. If you want to write your own rehashing function, you will have to understand all the internals of how hash tables work. These internals are not documented here, as the best way to learn them is to read the source

code.

:rehash-size — Specifies how much to increase the size of the hash table when it becomes full. This can be a fixnum which is the number of entries to add, or it can be a flonum which is the ratio of the new size to the old size. The default is 1.3, which causes the table to be made 30% bigger each time it has to grow.

gethash *key hash-table*
Find the entry in *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.

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

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

swaphash *key value hash-table*
This does the same thing as puthash, but returns different values. If there was already an entry in *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.

maphash *function hash-table*
For each entry in *hash-table*, call *function* on two arguments: the key of the entry and the value of the entry.

clrhash *hash-table*
Remove all the entries from *hash-table*. Returns the hash table itself.

5.9.2 Hashing on Equal

This section documents the functions for equal hash tables, which use trees as keys and associate objects with them. The function to make one is slightly different from make-hash-table because the implementations of the two kinds of hash table differ, but analogous operations are provided.

make-equal-hash-table &rest *options*
This creates a new hash table of the equal kind. Valid option keywords are:

:size — Sets the initial size of the hash table, in entries, as a fixnum. The default is 100 (octal). The actual size is rounded up from the size you specify to the next "good" size. You won't necessarily be able to store this many entries into the table before it overflows and becomes bigger; but except in the case of extreme bad luck you will be able to store almost this many.

:area
: Specifies the area in which the hash table should be created. This is just like the :area option to **make-array** (see page 111). Defaults to **nil** (i.e. **default-cons-area**).

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

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

gethash-equal *key hash-table*
Find the entry in *hash-table* whose key is **equal** to *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.

puthash-equal *key value hash-table*
Create an entry associating *key* to *value*; if there is already an entry for *key*, then replace the value of that entry with *value*. Returns *value*. If adding an entry to the hash table exceeds its rehash threshold, it is grown and rehashed so that searching does not become too slow.

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

swaphash-equal *key value hash-table*
This does the same thing as **puthash-equal**, but returns different values. If there was already an entry in *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**.

maphash-equal *function hash-table*
For each entry in *hash-table*, call *function* on two arguments: the key of the entry and the value of the entry.

clrhash-equal *hash-table*
Remove all the entries from *hash-table*. Returns the hash table itself.

5.9.3 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 gethash will rehash the table based on the new object addresses.

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

5.9.4 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.

sxhash *tree*

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

Here is an example of how to use sxhash in maintaining hash tables of trees:

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

To write an "intern" for trees, one could

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

sxhash provides what is called "hashing on equal"; that is, two objects that are equal are considered to be "the same" by sxhash. In particular, if two strings differ only in alphabetic case, sxhash will return the same thing for both of them because they are equal. The value returned by sxhash does not depend on the value of alphabetic-case-affects-string-comparison (see page 127).

Therefore, sxhash is useful for retrieving data when two keys that are not the same object but are equal are considered the same. If you consider two such keys to be different, then you need "hashing on eq", where two different objects are always considered different. In some Lisp implementations, there is an easy way to create a hash function that hashes on eq, namely, by returning the virtual address of the storage associated with the object. But in other implementations, of which Zetalisp is one, this doesn't work, because the address associated with an object can be changed by the relocating garbage collector. The hash tables created by make-hash-table deal with this problem by using the appropriate subprimitives so that they interface correctly with the garbage collector. If you need a hash table that hashes on eq, it is already provided; if you need an eq hash function for some other reason, you must build it yourself, either using the provided eq hash table facility or carefully using subprimitives.

5.10 Sorting

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

After sorting, the argument (be it list or array) has been rearranged internally so as to be completely ordered. In the case of an array argument, this is accomplished by permuting the elements of the array, while in the list case, the list is reordered by rplacd's in the same manner as nreverse. Thus if the argument should not be clobbered, the user must sort a copy of the argument, obtainable by fillarray or 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 section 5.4, page 63 for an explanation of compact lists, and A. I. Memo 587 by Guy L. Steele Jr. for an explanation of the sorting algorithm.

sort *table predicate*

The first argument to sort is an array or a list. The second is a predicate, which must be applicable to all the objects in the array or list. The predicate should take two arguments, and return non-nil if and only if the first argument is strictly less than the second (in some appropriate sense).

The sort function proceeds to sort the contents of the array or list under the ordering imposed by the predicate, and returns the array or list modified into sorted order. 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 copylist, page 58).

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.

sortcar *x predicate*

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*

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

stable-sortcar *x predicate*

stable-sortcar is like sortcar, but if two elements of *x* are equal, i.e. *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*

sort-grouped-array considers its array argument to be composed of records of *group-size* elements each. These records are considered as units, and are sorted with respect to one another. The *predicate* is applied to the first element of each record; so the first elements act as the keys on which the records are sorted.

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

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

5.11 Resources

Storage allocation is handled differently by different computer systems. In many languages, the programmer must spend a lot of time thinking about when variables and storage units are allocated and deallocated. In Lisp, freeing of allocated storage is normally done automatically by the Lisp system; when an object is no longer accessible to the Lisp environment, it is garbage collected. This relieves the programmer of a great burden, and makes writing programs much easier.

However, automatic freeing of storage incurs an expense: more computer resources must be devoted to the garbage collector. If a program is designed to allocate temporary storage, which is then left as garbage, more of the computer must be devoted to the collection of garbage; this expense can be high. In some cases, the programmer may decide that it is worth putting up with the inconvenience of having to free storage under program control, rather than letting the system do it automatically, in order to prevent a great deal of overhead from the garbage collector.

It usually is not worth worrying about freeing of storage when the units of storage are very small things such as conses or small arrays. Numbers are not a problem, either; fixnums and small flonums do not occupy storage, and the system has a special way of garbage-collecting the other kinds of numbers with low overhead. But when a program allocates and then gives up very large objects at a high rate (or large objects at a very high rate), it can be very worthwhile to keep track of that one kind of object manually. Within the Lisp Machine system, there are several programs that are in this position. The Chaosnet software allocates and frees "packets", which are moderately large, at a very high rate. The window system allocates and frees certain kinds of windows, which are very large, moderately often. Both of these programs manage their objects manually, keeping track of when they are no longer used.

When we say that a program "manually frees" storage, it does not really mean that the storage is freed in the same sense that the garbage collector frees storage. Instead, a list of unused objects is kept. When a new object is desired, the program first looks on the list to see if there is one around already, and if there is it uses it. Only if the list is empty does it actually allocate a new one. When the program is finished with the object, it returns it to this list.

The functions and special forms in this section perform the above function. The set of objects forming each such list is called a "resource"; for example, there might be a Chaosnet packet resource. defresource defines a new resource; allocate-resource allocates one of the objects; deallocate-resource frees one of the objects (putting it back on the list); and using-resource temporarily allocates an object and then frees it.

defresource *Special Form*

The defresource special form is used to define a new resource. The form looks like this:

```
(defresource name parameters
    keyword value
    keyword value
    ...)
```

name should be a symbol; it is the name of the resource and gets a defresource property of the internal data structure representing the resource.

parameters is a lambda-list giving names and default values (if &optional is used) of parameters to an object of this type. For example, if one had a resource of two-dimensional arrays to be used as temporary storage in a calculation, the resource would typically have two parameters, the number of rows and the number of columns. In the simplest case *parameters* is ().

The keyword options control how the objects of the resource are made and kept track of. The following keywords are allowed:

:constructor The *value* is either a form or the name of a function. It is responsible for making an object, and will be used when someone tries to allocate an object from the resource and no suitable free objects exist. If the *value* is a form, it may access the parameters as variables. If it is a function, it is given the internal data structure for the resource and any supplied parameters as its arguments; it will need to default any unsupplied optional parameters. This keyword is required.

:initial-copies The *value* is a number (or nil which means 0). This many objects will be made as part of the evaluation of the defresource; thus is useful to set up a pool of free objects during loading of a program. The default is to make no initial copies.

If initial copies are made and there are *parameters*, all the parameters must be &optional and the initial copies will have the default values of the parameters.

:finder The *value* is a form or a function as with :constructor and sees the same arguments. If this option is specified, the resource system does not keep track of the objects. Instead, the finder must do so. It will be called inside a without-interrupts and must find a usable object somehow and return it.

:matcher The *value* is a form or a function as with :constructor. In addition to the parameters, a form here may access the variable object (in the current package). A function gets the object as its second argument, after the data structure and before the parameters. The job of the matcher is to make sure that the object matches the specified parameters. If no matcher is supplied, the system will remember the values of the parameters (including optional ones that defaulted) that were used to construct the object, and will assume that it matches those particular values for all time. The comparison is done with equal (not eq). The matcher is called inside a without-interrupts.

:checker The *value* is a form or a function, as above. In addition to the parameters, a form here may access the variables object and in-use-p (in the current package). A function receives these as its second and third arguments, after the data structure and before the parameters. The job of the checker is to determine whether the object is safe to allocate. If no checker is supplied, the default checker looks only at in-use-p; if the object has been allocated and not freed it is not safe to allocate, otherwise it is. The checker is called inside a without-interrupts.

If these options are used with forms (rather than functions), the forms get compiled into functions as part of the expansion of defresource. These functions are given names like (:property *resource-name* si:resource-constructor); these names are not guaranteed not to change in the future.

Most of the options are not used in typical cases. Here is an example:

```
    (defresource two-dimensional-array (rows columns)
            :constructor (make-array (list rows columns)))
```

Suppose the array was usually going to be 100 by 100, and you wanted to preallocate one during loading of the program so that the first time you needed an array you wouldn't have to spend the time to create one. You might simply put

```
(using-resource (foo two-dimensional-array 100 100)
           )
```
after your defresource, which would allocate a 100 by 100 array and then immediately free it. Alternatively you could:
```
(defresource two-dimensional-array
                        (&optional (rows 100) (columns 100))
        :constructor (make-array (list rows columns))
        :initial-copies 1)
```

Here is an example of how you might use the :matcher option. Suppose you wanted to have a resource of two-dimensional arrays, as above, except that when you allocate one you don't care about the exact size, as long as it is big enough. Furthermore you realize that you are going to have a lot of different sizes and if you always allocated one of exactly the right size, you would allocate a lot of different arrays and would not reuse a pre-existing array very often. So you might:
```
(defresource sloppy-two-dimensional-array (rows columns)
    :constructor (make-array (list rows columns))
    :matcher (and (≥ (array-dimension-n 1 object) rows)
                  (≥ (array-dimension-n 2 object) columns)))
```

allocate-resource *name* &rest *parameters*

Allocate an object from the resource specified by *name*. The various forms and/or functions given as options to defresource, together with any *parameters* given to allocate-resource, control how a suitable object is found and whether a new one has to be constructed or an old one can be reused.

Note that the using-resource special form is usually what you want to use, rather than allocate-resource itself; see below.

deallocate-resource *name resource*

Free the object *resource*, returning it to the free-object list of the resource specified by *name*.

using-resource (*variable resource parameters...*) *body...* *Special Form*

The *body* forms are evaluated sequentially with *variable* bound to an object allocated from the resource named *resource*, using the given *parameters*. The *parameters* (if any) are evaluated, but *resource* is not.

using-resource is often more convenient than calling allocate-resource and deallocate-resource. Furthermore it is careful to free the object when the body is exited, whether it returns normally or via *throw. This is done by using unwind-protect; see page 49.

Here is an example of the use of resources:
```
      (defresource huge-16b-array (&optional (size 1000))
        :constructor (make-array size ':type 'art-16b))

      (defun do-complex-computation (x y)
        (using-resource (temp-array huge-16b-array)
          ...                                          ;Within the body, the array can be used.
          (aset 5 temp-array i)
          ...))                                        ;The array is returned at the end.
```

6. Symbols

6.1 The Value Cell

Each symbol has associated with it a *value cell*, which refers to one Lisp object. This object is called the symbol's *binding* or *value*, since it is what you get when you evaluate the symbol. The binding of symbols to values allows symbols to be used as the implementation of *variables* in programs.

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

Symbols are often used as special variables. Variables and how they work are described in section 3.1, page 14. 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 won't work if you try to use them on local variables.

set *symbol value*

set is the primitive for assignment of symbols. The *symbol*'s value is changed to *value*; *value* may be any Lisp object. set returns *value*.
Example:
```
(set (cond ((eq a b) 'c)
           (t 'd))
     'foo)
```
will either set c to foo or set d to foo.

symeval *sym*

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.

boundp *sym*

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

makunbound *sym*

makunbound causes *sym* to become unbound.
Example:
```
(setq a 1)
a => 1
(makunbound 'a)
a => causes an error.
```
makunbound returns its argument.

value-cell-location *sym*

value-cell-location returns a locative pointer to *sym*'s value cell. See the section on locatives (chapter 13, page 170). 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. For details, see the section on closures (chapter 11, page 158).

Note: the function value-cell-location works on symbols that get converted to local variables (see section 3.1, page 14); 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.

6.2 The Function Cell

Every symbol also has associated with it a *function cell*. The *function* cell is similar to the *value* cell; it refers to a Lisp object. When a function is referred to by name, that is, when a symbol is *applied* or appears as the car of a form to be evaluated, that symbol's function cell is used to find its *definition*, the functional object which is to be applied. For example, when evaluating (+ 5 6), the evaluator looks in +'s function cell to find the definition of +, in this case a *FEF* containing a compiled program, to apply to 5 and 6.

Maclisp does not have function cells; instead, it looks for special properties on the property list. This is one of the major incompatibilities between the two dialects.

Like the value cell, a function cell can be empty, and it can be bound or assigned. (However, to bind a function cell you must use the bind subprimitive; see page 183.) The following functions are analogous to the value-cell-related functions in the previous section.

fsymeval *sym*

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

fset *sym definition*

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

fboundp *sym*

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

fmakunbound *sym*

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

function-cell-location *sym*

function-cell-location returns a locative pointer to *sym*'s function cell. See the section on locatives (chapter 13, page 170). 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. Refer to chapter 10 for details.

6.3 The Property List

Every symbol has an associated property list. See section 5.8, page 71 for documentation of property lists. When a symbol is created, its property list is initially empty.

The Lisp language itself does not use a symbol's property list for anything. (This was not true in older Lisp implementations, where the print-name, value-cell, and function-cell of a symbol were kept on its property list.) However, various system programs use the property list to associate information with the symbol. For instance, the editor uses the property list of a symbol which is the name of a function to remember where it has the source code for that function, and the compiler uses the property list of a symbol which is the name of a special form to remember how to compile that special form.

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

plist *sym*

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*

This sets the list which 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*

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.

6.4 The Print Name

Every symbol has an associated string called the *print-name*, or *pname* for short. This string is used as the external representation of the symbol: if the string is typed in to read, it is read as a reference to that symbol (if it is interned), and if the symbol is printed, print types out the print-name. For more information, see the sections on the *reader* (see section 21.2.2, page 322) and *printer* (see section 21.2.1, page 319).

get-pname *sym*

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

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

samepnamep *sym1 sym2*

This predicate returns t if the two symbols *sym1* and *sym2* have equal print-names; that is, if their printed representation is the same. Upper and lower case letters are normally considered the same. If either or both of the arguments is a string instead of a symbol, then that string is used in place of the print-name. samepnamep is useful for determining if two symbols would be the same except that they are in different packages (see chapter 23, page 392).
Examples:

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

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

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

This is the same function as string-equal (see page 128). 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.

6.5 The Package Cell

Every symbol has a *package cell* which is used, for interned symbols, to point to the package which the symbol belongs to. For an uninterned symbol, the package cell contains nil. For information about packages in general, see the chapter on packages, chapter 23, page 392. For information about package cells, see page 399.

6.6 Creating Symbols

The functions in this section are primitives for creating symbols. However, before discussing them, it is important to point out that most symbols are created by a higher-level mechanism, namely the reader and the intern function. Nearly all symbols in Lisp are created by virtue of the reader's having seen a sequence of input characters that looked like the printed representation of a symbol. When the reader sees such a p.r., it calls intern (see page 399), which looks up the sequence of characters in a big table and sees whether any symbol with this print-name already exists. If it does, read uses the already-existing symbol. If it does not, then intern creates a new symbol and puts it into the table, and read uses that new symbol.

A symbol that has been put into such a table is called an *interned* symbol. Interned symbols are normally created automatically; the first time 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. For more information, turn to the chapter on packages (chapter 23, page 392).

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*

This creates a new uninterned symbol, whose print-name is the string *pname*. The value and function bindings will be unbound and the property list will be 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:

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

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

copysymbol *sym copy-props*

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*

gensym invents a print-name, and creates a new symbol with that print-name. It returns the new, uninterned symbol.

The invented print-name is a character prefix (the value of si:*gensym-prefix) followed by the decimal representation of a number (the value of si:*gensym-counter), e.g. "g0001". The number is increased by one every time gensym is called.

If the argument *x* is present and is a fixnum, then si:*gensym-counter is set to *x*. If *x* is a string or a symbol, then si:*gensym-prefix is set to the first character of the string or of the 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 which should not normally be seen by the user, and whose print-name is unimportant, except to allow easy distinction by eye between two such symbols. The optional argument is rarely supplied. The name comes from "generate symbol", and the symbols produced by it are often called "gensyms".

7. 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*. In Maclisp, there are generic numeric functions (like **plus**) and there are 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	Flonums are floating-point numbers. They have a mantissa of 32 bits and an exponent of 11 bits, providing a precision of about 9 digits and a range of about 10↑300. Stable rounding is employed.
small-flonum	Small flonums are another form of floating-point number, with a mantissa of 18 bits and an exponent of 7 bits, providing a precision of about 5 digits and a range of about 10↑19. Stable rounding is employed. Small flonums are useful because, like fixnums, and unlike flonums, they don't require any storage. Computing with small flonums is more efficient than with regular flonums because the operations are faster and consing overhead is eliminated.

Generally, Lisp objects have a unique identity; each exists, independent of any other, and you can use the eq predicate to determine whether two references are to the same object or not. Numbers are the exception to this rule; they don't work this way. The following function may return either t or nil. Its behavior is considered undefined, 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, in order to gain efficiency, and on the grounds that identity testing of numbers is not really an interesting thing to do. So, the rule is that the result of applying eq to numbers is undefined, and may return either t or nil at will. If you want to compare the values of two numbers, use = (see page 95).

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

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

When an arithmetic function of more than one argument is given arguments of different numeric types, uniform *coercion rules* are followed to convert the arguments to a common type, which is also the type of the result (for functions which return a number). When an integer meets a 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.

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 which 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.

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. If a number has an exponent separated by "s", it is a small flonum. Small flonums require a special indicator so that naive users will not 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 examples on page 323, in the description of what the reader understands.

7.1 Numeric Predicates

zerop *x*

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

plusp *x*

Returns t if its argument is a positive number, strictly greater than zero. Otherwise it returns nil. If *x* is not a number, plusp causes an error.

minusp *x*

Returns t if its argument is a negative number, strictly less than zero. Otherwise it returns nil. If *x* is not a number, minusp causes an error.

oddp *number*

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

evenp *number*

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

signp *test x* *Special Form*

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

l	$x < 0$
le	$x \le 0$
e	$x = 0$
n	$x \ne 0$
ge	$x \ge 0$

```
g   x > 0
```

Examples:

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

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

7.2 Numeric Comparisons

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

= *x y*

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

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

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

Examples:

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

>= *x y* &rest *more-args*
≥ *x y* &rest *more-args*

≥ 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**.

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

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

Examples:

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

<= *x y* &rest *more-args*
≤ *x y* &rest *more-args*

≤ 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.

≠ *x y*

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

max &rest *args*

max returns the largest of its arguments.
Example:

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

max requires at least one argument.

min &rest *args*

min returns the smallest of its arguments.
Example:

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

min requires at least one argument.

7.3 Arithmetic

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

plus &rest *args*
+ &rest *args*
+$ &rest *args*

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

difference *arg* &rest *args*

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

minus *x*

Returns the negative of *x*.
Examples:

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

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

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

abs *x*

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

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

times &rest *args*
***** &rest *args*
***$** &rest *args*

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

quotient *arg* &rest *args*

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

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

The name of this function is written // rather than / because / is the quoting character in Lisp syntax and must be doubled. With more than one argument, // is the same as quotient; it returns the first argument divided by all of the rest of its arguments. With only one argument, (// *x*) is the same as (// 1 *x*). The exact rules for the meaning of the quotient and remainder of two integers are given on page 93; 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
```

remainder *x y*
\ *x y*

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 on page 93.

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

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

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

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

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

gcd *x y* &rest *args*
\ *x y* &rest *args*

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

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

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*))).

sqrt *x*

Returns the square root of *x*.

isqrt *x*

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

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

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

7.4 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 [currently this is not true of most of them, but it will be fixed in the future].

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

log *x*
Returns the natural logarithm of *x*.

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

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

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

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

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

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

7.5 Numeric Type Conversions

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

fix *x*
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*
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*
Converts any kind of number to a flonum.

small-float *x*
Converts any kind of number to a small flonum.

7.6 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 &rest *args*
Returns the bit-wise logical *inclusive or* of its arguments. At least one argument is required.
Example:
```
(logior 4002 67) => 4067
```

logxor &rest *args*
Returns the bit-wise logical *exclusive or* of its arguments. At least one argument is required.
Example:
```
(logxor 2531 7777) => 5246
```

logand &rest *args*
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*
Returns the logical complement of *number*. This is the same as logxor'ing *number* with -1.
Example:
```
(lognot 3456) => -3457
```

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

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

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

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

With two arguments, the result of boole is simply its second argument. 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*

bit-test is a predicate which returns t if any of the bits designated by the 1's in *x* are 1's in *y*. bit-test is implemented as a macro which expands as follows:

```
(bit-test x y) ==> (not (zerop (logand x y)))
```

lsh *x y*

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

Examples:

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

ash *x y*

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

rot *x y*

Returns *x* rotated left *y* bits if *y* is positive or zero, or *x* rotated right |*y*| bits if *y* is negative. The rotation considers *x* as a 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*

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*

Returns the high *n* bits of the binary representation of |*x*|, or the low -*n* bits if *n* is negative. *x* may be a fixnum or a bignum; its sign is ignored. haipart could have been defined by:
```
        (defun haipart (x n)
          (setq x (abs x))
          (if (minusp n)
              (logand x (1- (ash 1 (- n))))
              (ash x (min (- n (haulong x))
                          0))))
```

7.7 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 (i.e. 10 octal) refers to the lowest eight bits of a word, and the byte-specifier 1010 refers to the next eight bits. These byte-specifiers will be stylized below as *ppss*. The maximum value of the *ss* digits is 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*

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*
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*
ldb-test is a predicate which returns t if any of the bits designated by the byte specifier *ppss* are 1's in *y*. That is, it returns t if the designated field is non-zero. ldb-test is implemented as a macro which expands as follows:
```
        (ldb-test ppss y) ==> (not (zerop (ldb ppss y)))
```

mask-field *ppss num*
This is similar to ldb; however, the specified byte of *num* is returned as a number in position *pp* of the returned word, instead of position 0 as with ldb. *num* must be a fixnum.
Example:
```
        (mask-field 0306 4567) => 560
```

dpb *byte ppss num*
Returns a number which is the same as *num* except in the bits specified by *ppss*. The low *ss* bits of *byte* are placed in those bits. *byte* is interpreted as being right-justified, as if it were the result of ldb. *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*
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*
This is like dpb, except that *byte* is not taken to be left-justified; the *ppss* bits of *byte* are used for the *ppss* bits of the result, with the rest of the bits taken from *num*. *num* must be a fixnum.
Example:
```
        (deposit-field 230 0306 4567) => 4237
```

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*

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

%logdpb *byte ppss fixnum*

%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.

7.8 Random Numbers

The functions in this section provide a pseudo-random number generator facility. The basic function you use is random, which returns a new pseudo-random number each time it is called. 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 pseudo-random 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*

(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 (see below). Otherwise, the default random-array is used (and is created if it doesn't already exist). The algorithm is executed inside a without-interrupts (see page 430) so two processes can use the same random-array without colliding.

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)

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*

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).

7.9 24-Bit Numbers

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

%24-bit-plus *x y*

Returns the sum of *x* and *y* modulo 2↑24. Both arguments must be fixnums.

%24-bit-difference *x y*

Returns the difference of *x* and *y* modulo 2↑24. Both arguments must be fixnums.

%24-bit-times *x y*

Returns the product of *x* and *y* modulo 2↑24. Both arguments must be fixnums.

7.10 Double-Precision Arithmetic

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

%multiply-fractions *num1 num2*

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

%divide-double *dividend[24:46] dividend[0:23] divisor*
: Divides the double-precision number given by the first two arguments by the third argument, and returns the single-precision quotient. Causes an error if division by zero or if the quotient won't fit in single precision.

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

%float-double *high24 low24*
: *high24* and *low24*, which must be fixnums, are concatenated to produce a 48-bit unsigned positive integer. A flonum containing the same value is constructed and returned. Note that only the 31 most-significant bits are retained (after removal of leading zeroes.) This function is mainly for the benefit of **read**.

8. 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 which the array has) is the number of subscripts used to refer to one of the elements of the array. The dimensionality may be any integer from one to seven, inclusively.

The lowest value for any subscript is zero; the highest value is a property of the array. Each dimension has a size, which is the lowest number which is too great to be used as a subscript. For example, in a one-dimensional array of five elements, the size of the one and only dimension is five, and the acceptable values of the subscript are zero, one, two, three, and four.

The most basic primitive functions for handling arrays are: make-array, which is used for the creation of arrays, aref, which is used for examining the contents of arrays, and aset, which is used for storing into arrays.

An array is a regular Lisp object, and it is common for an array to be the binding of a symbol, or the car or cdr of a cons, or, in fact, an element of an array. There are many functions, described in this chapter, which take arrays as arguments and perform useful operations on them.

Another way of handling arrays, inherited from Maclisp, is to treat them as functions. In this case each array has a name, which is a symbol whose function definition is the array. Zetalisp supports this style by allowing an array to be *applied* to arguments, as if it were a function. The arguments are treated as subscripts and the array is referenced appropriately. The store special form (see page 125) is also supported. This kind of array referencing is considered to be obsolete, and is slower than the usual kind. It should not be used in new programs.

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 will take an art-q-list array and return 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 will change, and if you store into the array, the corresponding element of the list will change the same way. An attempt to rplacd the list will cause 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 non-negative fixnum, and only the least significant *n* bits are remembered in the array; all of the others are discarded. Thus **art-1b** arrays store only 0 and 1, and if you store a 5 into an **art-2b** array and look at it later, you will find a 1 rather than a 5.

These arrays are used when it is known beforehand that the fixnums which will be stored are non-negative and limited in size to a certain number of bits. Their advantage over the **art-q** array is that they occupy less storage, because more than one element of the array is kept in a single machine word. (For example, 32 elements of an **art-1b** array or 2 elements of an **art-16b** array will fit into one word).

There are also **art-32b** arrays which have 32 bits per element. Since fixnums only have 24 bits anyway, these are the same as **art-q** arrays except that they only hold fixnums. They 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*, and chapter 9 of this manual deals with functions that manipulate them.

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 (see page 100). 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 pdp11/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 pdp11. If the magnitude of the number is too large, the largest valid floating-point number will be stored. If the magnitude is too small, zero 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 which exist only for the implementation of *stack groups*; these types are called **art-stack-group-head**, **art-special-pdl**, and **art-reg-pdl**. Their elements may be any Lisp object; their use is explained in the section on stack groups (see chapter 12, page 163).

Currently, multi-dimensional 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 multi-dimensional 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*
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 (see page 69) which associates each array type symbol with the number of array elements stored in one word, for an array of that type. If the value is negative, it is instead the number of words per array element, for arrays whose elements are more than one word long.

array-elements-per-q *array-type-code*
Given the internal array-type code number, returns the number of array elements stored in one word, for an array of that type. If the value is negative, it is instead the number of words per array element, for arrays whose elements are more than one word long.

array-bits-per-element *Variable*
The value of **array-bits-per-element** is an association list (see page 69) which associates each array type symbol with the number of bits of unsigned number it can hold, or **nil** if it can hold Lisp objects. This can be used to tell whether an array can hold Lisp objects or not.

array-bits-per-element *array-type-code*
Given the internal array-type code numbers, returns the number of bits per cell for unsigned numeric arrays, or **nil** for a type of array that can contain Lisp objects.

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

8.1 Extra Features of Arrays

Any array may have an *array leader*. An array leader is like a one-dimensional art-q array which is attached to the main array. So an array which has a leader acts like two arrays joined together. The leader can be stored into and examined by a special set of functions, different from those used for the main array: array-leader and store-array-leader. The leader is always one-dimensional, and always can hold any kind of Lisp object, regardless of the type or 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 non-homogeneous pieces of data. In this case the leader is not used like an array; each slot is used differently from the others. Explicit numeric subscripts should not be used for the leader elements of such an array; instead the leader should be described by a defstruct (see page 259).

By convention, element 0 of the array leader of an array is used to hold the number of elements in the array that are "active" 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 five, then only elements zero through four of the string are considered to be "active"; the string's printed representation will be five characters long, string-searching functions will stop after the fifth element, etc.

The system does not provide a way to turn off the fill-pointer convention; any array that has a leader must reserve element 0 for the fill pointer or avoid using many of the array functions.

Leader element 1 is used in conjunction with the "named structure" feature to associate a "data type" with the array; see page 271. Element 1 is only treated specially if the array is flagged as a named structure.

The following explanation of *displaced arrays* is probably not of interest to a beginner; the section may be passed over without losing the continuity of the manual.

Normally, an array is represented as a small amount of header information, followed by the contents of the array. However, sometimes it is desirable to have the header information removed from the actual contents. One such occasion is when the contents of the array must be located in a special part of the Lisp Machine's address space, such as the area used for the control of input/output devices, or the bitmap memory which generates the TV image. Displaced arrays are also used to reference certain special system tables, which are at fixed addresses so the microcode can access them easily.

If you give make-array a fixnum or a locative as the value of the :displaced-to option, it 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 which was indirected to a second, two-dimensional array of three elements by three, then the elements could be accessed in either a one-dimensional or a two-dimensional manner. Weird effects can be produced if the new array is of a different type than the old array; this is not generally recommended. Indirecting an art-*m*b array to an art-*n*b array 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 (see page 129) creates such arrays. When using index offsets with multi-dimensional arrays, there is only one index offset; it is added in to the "linearized" subscript which is the result of multiplying each subscript by an appropriate coefficient and adding them together.

8.2 Basic Array Functions

make-array *dimensions* &rest *options*

This is the primitive function for making arrays. *dimensions* should be a list of fixnums which are the dimensions of the array; the length of the list 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 (see chapter 15, page 192) the list should be created. It should be either an area number (a fixnum), or nil to mean the default area.
:type	The value should be a symbolic name of an array type; the most common of these is art-q, which is the default. The elements of the array are initialized according to the type: if the array is of a type whose elements may only be fixnums or flonums, then every element of the array will initially be 0 or 0.0; otherwise, every element will initially be nil. See the description of array types on page 107. The value of the option may also be the value of a symbol which is an array type name (that is, an internal numeric array type code).

: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 which refers to the specified section of virtual address space. If the value is an array, make-array will create an indirect array (see page 111).

: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 (see below).

:leader-list The value should be a list. Call the number of elements in the list *n*. The first *n* elements of the leader 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.

:displaced-index-offset
If this is present, the value of the :displaced-to option should be an array, and the value should be a non-negative fixnum; it is made to be the index-offset of the cr' ated indirect array. (See page 111.)

: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 page 271.) 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, i.e. the **%structure-total-size** of the array.

When make-array was originally implemented, it took its arguments in the following fixed pattern:

(make-array *area type dimensions*
&optional *displaced-to leader*
displaced-index-offset
named-structure-symbol)

leader was a combination of the :leader-length and :leader-list options, and the list was in reverse order. This obsolete form is still supported so that old programs will continue to work, but the new keyword-argument form is preferred.

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

ar-1 *array i*
ar-2 *array i j*
ar-3 *array i j k*
These are obsolete versions of aref that only work for one, two, or three dimensional arrays, respectively. There is no reason ever to use them.

aset *x array* &rest *subscripts*
Stores *x* into the element of *array* selected by the *subscripts*. The *subscripts* must be fixnums and their number must match the dimensionality of *array*. The returned value is *x*.

as-1 *x array i*
as-2 *x array i j*
as-3 *x array i j k*
These are obsolete versions of aset that only work for one, two, or three dimensional arrays, respectively. There is no reason ever to use them.

aloc *array* &rest *subscripts*
Returns a locative pointer to the element-cell of *array* selected by the *subscripts*. The *subscripts* must be fixnums and their number must match the dimensionality of *array*. See the explanation of locatives in chapter 13, page 170.

ap-1 *array i*
ap-2 *array i j*
ap-3 *array i j k*
These are obsolete versions of aloc that only work for one, two, or three dimensional arrays, respectively. There is no reason ever to use them.

The compiler turns aref into ar-1, ar-2, etc. according to the number of subscripts specified, turns aset into as-1, as-2, etc., and turns aloc into ap-1, ap-2, etc. For arrays with more than 3 dimensions the compiler uses the slightly less efficient form since the special routines only exist for 1, 2, and 3 dimensions. There is no reason for any program to call ar-1, as-1, ar-2, etc. explicitly; they are documented because there used to be such a reason, and many old programs use these functions. New programs should use aref, aset, and aloc.

A related function, provided only for Maclisp compatibility, is arraycall (page 125).

array-leader *array i*
array should be an array with a leader, and *i* should be a fixnum. This returns the *i*'th element of *array*'s leader. This is analogous to aref.

store-array-leader *x array i*
array should be an array with a leader, and *i* should be a fixnum. *x* may be any object. *x* is stored in the *i*'th element of *array*'s leader. store-array-leader returns *x*. This is analogous to aset.

ap-leader *array i*
array should be an array with a leader, and *i* should be a fixnum. This returns a locative pointer to the *i*'th element of *array*'s leader. See the explanation of locatives, chapter 13, page 170. This is analogous to aloc.

8.3 Getting Information About an Array

array-type *array*
Returns the symbolic type of *array*.
Example:
```
        (setq a (make-array '(3 5)))
        (array-type a) => art-q
```

array-length *array*
array may be any array. This returns the total number of elements in *array*. For a one-dimensional array, this is one greater than the maximum allowable subscript. (But if fill pointers are being used, you may want to use array-active-length.)

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

array-active-length *array*
If *array* does not have a fill pointer, then this returns whatever (array-length *array*) would have. If *array* does have a fill pointer, array-active-length returns it. See the general explanation of the use of fill pointers, on page 110.

array-#-dims *array*
Returns the dimensionality of *array*. Note that the name of the function includes a "#", which must be slashified if you want to be able to read your program in Maclisp. (It doesn't need to be slashified for the Zetalisp reader, which is smarter.)
Example:
```
        (array-#-dims (make-array '(3 5))) => 2
```

array-dimension-n *n array*
array may be any kind of array, and *n* should be a fixnum. If *n* is between 1 and the dimensionality of *array*, this returns the *n*'th dimension of *array*. If *n* is 0, 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*
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*
array may be any array; it also may be a symbol whose function cell contains an array, for Maclisp compatibility (see section 8.10, page 124). 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*

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*

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*

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

array-indexed-p *array*

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

array-has-leader-p *array*

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

array-leader-length *array*

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

8.4 Changing the Size of an Array

adjust-array-size *array new-size*

If *array* is a one-dimensional array, its size is changed to be *new-size*. If *array* has more than one dimension, its size (**array-length**) is changed to *new-size* by changing only the last dimension.

If *array* is made smaller, the extra elements are lost; if *array* is made bigger, the new elements are initialized in the same fashion as **make-array** (see page 111) would initialize them: either to **nil** or **0**, depending on the type of array.

Example:
```
(setq a (make-array 5))
(aset 'foo a 4)
(aref a 4) => foo
(adjust-array-size a 2)
(aref a 4) => an error occurs
```

If the size of the array is being increased, **adjust-array-size** may have to allocate a new array somewhere. In that case, it alters *array* so that references to it will be made to the new array instead, by means of "invisible pointers" (see **structure-forward**, page 175). **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 which are not the same (i.e. not **eq**), but which share the same contents.

array-grow *array* &rest *dimensions*

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 multi-dimensional 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*

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, i.e. if it was not the most recently allocated non-list 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.

8.5 Arrays Overlaid With Lists

These functions manipulate art-q-list arrays, which were introduced on page 107.

g-l-p *array*

array should be an art-q-list array. This returns a list which shares the storage of *array*. Example:

```
(setq a (make-array 4 ':type 'art-q-list))
(aref a 0) => nil
(setq b (g-l-p a)) => (nil nil nil nil)
(rplaca b t)
b => (t nil nil nil)
(aref a 0) => t
(aset 30 a 2)
b => (t nil 30 nil)
```

The following two functions work strangely, in the same way that store does, and should not be used in new programs.

get-list-pointer-into-array *array-ref*

The argument *array-ref* is ignored, but should be a reference to an **art-q-list** array by applying the array to subscripts (rather than by **aref**). This returns a list object which is a portion of the "list" of the array, beginning with the last element of the last array which has been called as a function.

get-locative-pointer-into-array *array-ref*

get-locative-pointer-into-array is similar to **get-list-pointer-into-array**, except that it returns a locative, and doesn't require the array to be **art-q-list**. Use **aloc** instead of this function in new programs.

8.6 Adding to the End of an Array

array-push *array x*

array must be a one-dimensional array which has a fill pointer, and *x* may be any object. **array-push** attempts to store *x* in the element of the array designated by the fill pointer, and increase the fill pointer by one. If the fill pointer does not designate an element of the array (specifically, when it gets too big), it is unaffected and **array-push** returns **nil**; otherwise, the two actions (storing and incrementing) happen uninterruptibly, and **array-push** returns the *former* value of the fill pointer, i.e. the array index in which it stored *x*. If the array is of type **art-q-list**, an operation similar to **nconc** has taken place, in that the element has been added to the list by changing the cdr of the formerly last element. The cdr coding is updated to ensure this.

array-push-extend *array x* &optional *extension*

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

array must be a one-dimensional array which has a fill pointer. The fill pointer is decreased by one, and the array element designated by the new value of the fill pointer is returned. If the new value does not designate any element of the array (specifically, if it had already reached zero), an error is caused. The two operations (decrementing and array referencing) happen uninterruptibly. If the array is of type **art-q-list**, an operation similar to **nbutlast** has taken place. The cdr coding is updated to ensure this.

8.7 Copying an Array

fillarray *array x*

array may be any type of array, or, for Maclisp compatibility, a symbol whose function cell contains an array. There are two forms of this function, depending on the type of *x*.

If *x* is a list, then fillarray fills up *array* with the elements of *list*. If *x* is too short to fill up all of *array*, then the last element of *x* is used to fill the remaining elements of *array*. If *x* is too long, the extra elements are ignored. If *x* is nil (the empty list), *array* is filled with the default initial value for its array type (nil or 0).

If *x* is an array (or, for Maclisp compatibility, a symbol whose function cell contains an array), then the elements of *array* are filled up from the elements of *x*. If *x* is too small, then the extra elements of *array* are not affected.

If *array* is multi-dimensional, the elements are accessed in row-major order: the last subscript varies the most quickly. The same is true of *x* if it is an array.

fillarray returns *array*.

listarray *array* &optional *limit*

array may be any type of array, or, for Maclisp compatibility, a symbol whose function cell contains an array. listarray creates and returns a list whose elements are those of *array*. If *limit* is present, it should be a fixnum, and only the first *limit* (if there are more than that many) elements of *array* are used, and so the maximum length of the returned list is *limit*.

If *array* is multi-dimensional, the elements are accessed in row-major order: the last subscript varies the most quickly.

list-array-leader *array* &optional *limit*

array may be any type of array, or, for Maclisp compatibility, a symbol whose function cell contains an array. list-array-leader creates and returns a list whose elements are those of *array*'s leader. If *limit* is present, it should be a fixnum, and only the first *limit* (if there are more than that many) elements of *array*'s leader are used, and so the maximum length of the returned list is *limit*. If *array* has no leader, nil is returned.

copy-array-contents *from to*

from and *to* must be arrays. The contents of *from* is copied into the contents of *to*, element by element. If *to* is shorter than *from*, the rest of *from* is ignored. If *from* is shorter than *to*, the rest of *to* is filled with nil 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 multi-dimensional arrays. *from* and *to* are "linearized" subscripts, and column-major order is used, *i.e. the first subscript varies fastest (opposite from* fillarray*).*

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

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*

The portion of the array *from-array* with indices greater than or equal to *from-start* and less than *from-end* is copied into the portion of the array *to-array* with indices greater than or equal to *to-start* and less than *to-end*, element by element. If there are more elements in the selected portion of *to-array* than in the selected portion of *from-array*, the extra elements are filled with the default value as by copy-array-contents. If there are more elements in the selected portion of *from-array*, the extra ones are ignored. Multi-dimensional arrays are treated the same way as copy-array-contents treats them. This function always returns t.

bitblt *alu width height from-array from-x from-y to-array to-x to-y*

from-array and *to-array* must be two-dimensional arrays of bits or bytes (art-1b, art-2b, art-4b, art-8b, art-16b, or art-32b). bitblt copies a rectangular portion of *from-array* into a rectangular portion of *to-array*. The value stored can be a Boolean function of the new value and the value already there, under the control of *alu* (see below). This function is most commonly used in connection with raster images for TV displays.

The top-left corner of the source rectangle is (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*.

If bitblt goes outside the bounds of the source array, it wraps around. This allows such operations as the replication of a small stipple pattern through a large array. If bitblt goes outside the bounds of the destination array, it signals an error.

If *src* is an element of the source rectangle, and *dst* is the corresponding element of the destination rectangle, then bitblt changes the value of *dst* to (boole *alu src dst*). See the boole function (page 100). There are symbolic names for some of the most useful *alu* functions; they are tv:alu-seta (plain copy), tv:alu-ior (inclusive or), tv:alu-xor (exclusive or), and tv:alu-andca (and with complement of source).

bitblt is written in highly-optimized microcode and goes very much faster than the same thing written with ordinary **aref** and **aset** operations would. Unfortunately this causes **bitblt** to have a couple of strange restrictions. Wrap-around does not work correctly if *from-array* is an indirect array with an index-offset. **bitblt** 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., etc.

8.8 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*

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*

Computes the inverse of *matrix*. If *into-matrix* is supplied, stores the result into it and returns it; otherwise it creates an array to hold the result, and returns that. *matrix* must be two-dimensional and square. The Gauss-Jordan algorithm with partial pivoting is used. Note: if you want to solve a set of simultaneous equations, you should not use this function; use **math:decompose** and **math:solve** (see below).

math:transpose-matrix *matrix* &optional *into-matrix*

Transposes *matrix*. If *into-matrix* is supplied, stores the result into it and returns it; otherwise it creates an array to hold the result, and returns that. *matrix* must be a two-dimensional array. *into-matrix*, if provided, must be two-dimensional and have sufficient dimensions to hold the transpose of *matrix*.

math:determinant *matrix*

Returns the determinant of *matrix*. *matrix* must be a two-dimensional square matrix.

The next two functions are used to solve sets of simultaneous linear equations. **math:decompose** takes a matrix holding the coefficients of the equations and produces the LU decomposition; this decomposition can then be passed to **math:solve** along with a vector of right-hand sides to get the values of the variables. If you want to solve the same equations for many different sets of right-hand side values, you only need to call **math:decompose** once. In terms of the argument names used below, these two functions exist to solve the vector equation $A\ x = b$ for x. A is a matrix. b and x are vectors.

math:decompose *a* &optional *lu ps*

Computes the LU decomposition of matrix *a*. If *lu* is non-nil, stores the result into it and returns it; otherwise it creates an array to hold the result, and returns that. The lower triangle of *lu*, with ones added along the diagonal, is L, and the upper triangle of *lu* is U, such that the product of L and U is *a*. Gaussian elimination with partial pivoting is used. The *lu* array is permuted by rows according to the permutation array *ps*, which is also produced by this function; if the argument *ps* is supplied, the permutation array is stored into it; otherwise, an array is created to hold it. This function returns two values: the LU decomposition and the permutation array.

math:solve *lu ps b* &optional *x*

This function takes the LU decomposition and associated permutation array produced by math:decompose, and solves the set of simultaneous equations defined by the original matrix *a* and the right-hand sides in the vector *b*. If *x* is supplied, the solutions are stored into it and it is returned; otherwise, an array is created to hold the solutions and that is returned. *b* must be a one-dimensional array.

math:list-2d-array *array*

Returns a list of lists containing the values in *array*, which must be a two-dimensional array. There is one element for each row; each element is a list of the values in that row.

math:fill-2d-array *array list*

This is the opposite of math:list-2d-array. *list* should be a list of lists, with each element being a list corresponding to a row. *array*'s elements are stored from the list. Unlike fillarray (see page 119), if *list* is not long enough, math:fill-2d-array "wraps around", starting over at the beginning. The lists which are elements of *list* also work this way.

8.9 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 can't 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 don't work on planes. You can use make-plane to create a plane, plane-aref or plane-ref to get the value of a component, and plane-aset or plane-store to store into a component. array-#-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 which 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*

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 (e.g. art-1b) specifying the type of the array out of which the plane is made.
:default-value	The default component value as explained above.
:extension	The amount by which to extend the plane, as explained above.

Example:

```
        (make-plane 2 ':type 'art-4b ':default-value 3)
```

creates a two-dimensional plane of type art-4b, with default value 3.

plane-origin *plane*

A list of numbers, giving the lowest coordinate values actually stored.

plane-default *plane*

This is the contents of the infinite number of plane elements which are not actually stored.

plane-extension *plane*

The amount to extend the plane by in any direction when plane-store is done outside of the currently-stored portion.

plane-aref *plane* &rest *subscripts*
plane-ref *plane subscripts*

These two functions return the contents of a specified element of a plane. They differ only in the way they take their arguments; plane-aref wants the subscripts as arguments, while plane-ref wants a list of subscripts.

plane-aset *datum plane* &rest *subscripts*
plane-store *datum plane subscripts*

These two functions store *datum* into the specified element of a plane, extending it if necessary, and return *datum*. They differ only in the way they take their arguments; plane-aset wants the subscripts as arguments, while plane-store wants a list of subscripts.

8.10 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. See the descriptions of **read** (page 332) and **intern** (page 399) for information about readtables and obarrays (packages). There are no "dead" arrays, nor are Multics "external" arrays provided.

The **arraycall** function exists for compatibility but should not be used (see **aref**, page 113.)

Subscripts are always checked for validity, regardless of the value of ***rset** and whether the code is compiled or not. However, in a multi-dimensional array, an error is only caused if the subscripts would have resulted in a reference to storage outside of the array. 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, multi-dimensional arrays are stored in column-major order rather than row-major order as in Maclisp. See chapter 8, page 109 for further discussion of this issue.

loadarrays and **dumparrays** are not provided. However, arrays can be put into "QFASL" files; see section 16.8, page 206.

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 are provided, and when arrays are applied to arguments, the arguments are treated as subscripts and **apply** returns the corresponding element of the array.

array &quote *symbol type* &eval &rest *dims*
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*
This is just like **array**, except that all of the arguments are evaluated.

store *array-reference x* *Special Form*

store stores *x* into the specified array element. *array-ref* should be a form which references an array by calling it as a function (**aref** forms are not acceptable). First *x* is evaluated, then *array-ref* is evaluated, and then the value of *x* is stored into the array cell last referenced by a function call, presumably the one in *array-ref*.

xstore *x array-ref*

This is just like **store**, but it is not a special form; this is because the arguments are in the other order. This function only exists for the compiler to compile the **store** special form into, and should never be used by programs.

arraycall *ignored array* &rest *subscripts*

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

9. Strings

Strings are a type of array which 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.

The way characters work, including multiple fonts and the extra bits from the keyboard, is explained in section 21.1, page 314. 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 page 325 for details of this syntax.

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 which 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 (see page 110) so that you can change the length of the string as well as the contents. The length of a string is always computed using array-active-length, so that if a string has a fill-pointer, its value will be used as the length.

9.1 Characters

character *x*

character coerces *x* to a single character, represented as a fixnum. If *x* is a number, it is returned. If *x* is a string or an array, its first element is returned. If *x* is a symbol, the first character of its pname is returned. Otherwise, an error occurs. The way characters are represented as fixnums is explained in section 21.1, page 314.

char-equal *ch1 ch2*

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

char-lessp *ch1 ch2*

This is the primitive for comparing characters for order; many of the string functions call it. *ch1* and *ch2* must be fixnums. The result is t if *ch1* comes before *ch2* ignoring case and font, otherwise nil. Details of the ordering of characters are in section 21.1, page 314.

9.2 Upper and Lower Case 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 upper-case and lower-case letters. If it is nil, lower-case characters behave as if they were the same character but in upper-case. 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 *ch*

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

char-downcase *ch*

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

string-upcase *string*

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

string-downcase *string*

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

9.3 Basic String Operations

string *x*

string coerces *x* into a string. Most of the string functions apply this to their string arguments. If *x* is a string (or any array), it is returned. If *x* is a symbol, its pname is returned. If *x* is a non-negative fixnum less than 400 octal, a one-character-long string containing it is created and returned. If *x* is a pathname (see chapter 22, page 376), the "string for printing" is returned. Otherwise, an error is signalled.

string-length *string*

string-length returns the number of characters in *string*. This is 1 if *string* is a number, the array-active-length (see page 115) if *string* is an array, or the array-active-length of the pname if *string* is a symbol.

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

string-equal compares two strings, returning t if they are equal and nil if they are not. The comparison ignores the extra "font" bits in 16-bit strings 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 idx1 string2 idx2 count*

%string-equal is the microcode primitive which 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 which require high efficiency and are willing to pay the price of less generality may want to use %string-equal in place of string-equal.

Examples:

To compare the two strings *foo* and *bar*:
```
(%string-equal foo 0 bar 0 nil)
```
To see if the string *foo* starts with the characters "bar":
```
(%string-equal foo 0 "bar" 0 3)
```

string-lessp *string1 string2*

string-lessp compares two strings using dictionary 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.

substring *string start* &optional *end area*

This extracts a substring of *string*, starting at the character specified by *start* and going up to but not including the character specified by *end*. *start* and *end* are 0-origin indices. The length of the returned string is *end* minus *start*. If *end* is not specified it defaults to the length of *string*. The area in which the result is to be consed may be optionally specified.

Example:
```
(substring "Nebuchadnezzar" 4 8) => "chad"
```

nsubstring *string start* &optional *end area*

nsubstring is the same as substring except that the substring is not copied; instead an indirect array (see page 111) is created which shares part of the argument *string*. Modifying one string will modify the other.

Note that nsubstring does not necessarily use less storage than substring; an nsubstring of any length uses at least as much storage as a substring 12 characters long. So you shouldn't use this just "for efficiency"; it is intended for uses in which it is important to have a substring which, if modified, will cause the original string to be modified too.

string-append &rest *strings*

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

Example:
```
(string-append #/! "foo" #/!) => "!foo!"
```

string-nconc *modified-string* &rest *strings*

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 (page 118). The value of string-nconc is *modified-string* or a new, longer copy of it; in the latter case the original copy is forwarded to the new copy (see adjust-array-size, page 116). 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*

This returns a substring of *string*, with all characters in *char-set* stripped off of 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*

This returns a substring of *string*, with all characters in *char-set* stripped off of 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*

This returns a substring of *string*, with all characters in *char-set* stripped off of 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*

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

string-nreverse *string*

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

string-pluralize *string*

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.

9.4 String Searching

string-search-char *char string* &optional (*from 0*) *to*

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

%string-search-char is the microcode primitive which 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*

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*

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*

string-search-set searches through *string* looking for a character which is in *char-set*. The search begins at the index *from*, which defaults to the beginning. It returns the index of the first character which is char-equal to some element of *char-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*

string-search-not-set searches through *string* looking for a character which 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 which 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)

string-reverse-search-char searches through *string* in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, and returns the index of the first character which is char-equal to *char*, or nil if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. 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)

string-reverse-search-not-char searches through *string* in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, and returns the index of the first character which is *not* char-equal to *char*, or nil if none is found. Note that the index returned is from the beginning of the string, although the search starts from the end. 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)

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)

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

string-reverse-search-not-set searches through *string* in reverse order, starting from the index one less than *from*, which defaults to the length of *string*, and returns the index of the first character which is not char-equal to any element of *char-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 (page 399), which given a string will return "the" symbol with that print name.

9.5 I/O to Strings

The special forms in this section allow you to create I/O streams which input from or output to a string rather than a real I/O device. See section 21.5.1, page 338 for documentation of I/O streams.

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 which 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 can't use its value within the body to see how far the reading has gotten.

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 which saves the characters output to it in a string. The value of the special form is the string.

```
(with-output-to-string (var string)
  body)
```

will append its output to the string which is the value of the form *string*. (This is like the `string-nconc` function; see page 129.) 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 which 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 can't use its value within the body to see how far the writing has gotten. The presence of *index* means that *string* is not required to have a fill-pointer; if 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.

It is OK 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 (see page 346).

9.6 Maclisp-Compatible Functions

The following functions are provided primarily for Maclisp compatibility.

alphalessp *string1 string2*

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

getchar *string index*

Returns the *index*'th character of *string* as a symbol. Note that 1-origin indexing is used. This function is mainly for Maclisp compatibility; aref should be used to index into strings (however, aref will not coerce symbols or numbers into strings).

getcharn *string index*

Returns the *index*'th character of *string* as a fixnum. Note that 1-origin indexing is used. This function is mainly for Maclisp compatibility; aref should be used to index into strings (however, aref will not coerce symbols or numbers into strings).

ascii *x*

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

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

The symbol returned is interned in the current package (see chapter 23, page 392).

maknam *char-list*

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

```
(maknam '(a b #/0 d)) => ab0d
```

implode *char-list*

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

The samepnamep function is also provided; see page 89.

10. Functions

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

This chapter contains internal details intended for those writing programs to manipulate programs as well as material suitable for the beginner. Feel free to skip sections that look complicated or uninteresting when reading this for the first time.

10.1 What Is a Function?

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

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

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

10.2 Function Specs

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

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

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

Some kinds of functions remember their own names, and some don't. The "name" remembered by a function can be any kind of function spec, although it is usually a symbol. In the examples of functions in the previous section, the one starting with the symbol named-lambda, the one whose printed representation included dtp-fef-pointer, and the dtp-u-entry remembered names (the function specs foo, append, and last respectively). The others didn't remember their names.

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

To *define a function* means to create a new function, and define a given function spec as that new function. This is what the defun special form does. Several other special forms, such as defmethod (page 293) and defselect (page 147) do this too.

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

A function spec's definition in general consists of a *basic definition* surrounded by *encapsulations*. Both the basic definition and the encapsulations are functions, but of recognizably different kinds. What defun creates is a basic definition, and usually that is all there is. Encapsulations are made by function-altering functions such as trace and advise. When the function is called, the entire definition, which includes the tracing and advice, is used. If the function is "redefined" with defun, only the basic definition is changed; the encapsulations are left in place. See the section on encapsulations, section 10.10, page 153.

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

a symbol
: The function is remembered in the function cell of the symbol. See page 87 for an explanation of function cells and the primitive functions to manipulate them.

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

(:method *flavor-name message*)
(:method *flavor-name method-type message*)
: The function is remembered inside internal data structures of the flavor system, and in the flavor-method-symbol of the function. See the chapter on flavors (chapter 20, page 279)

for details.

(:location *pointer*)

The function is stored in the cdr of *pointer*, which may be a locative or a list. This is for pointing at an arbitrary place which there is no other way to describe. This form of function spec isn't useful in defun (and related special forms) because the reader has no printed representation for locative pointers and always creates new lists; these function specs are intended for programs that manipulate functions (see section 10.8, page 149).

(:within *within-function function-to-affect*)

This refers to the meaning of the symbol *function-to-affect*, but only where it occurs in the text of the definition of *within-function*. If you define this function spec as anything but the symbol *function-to-affect* itself, then that symbol is replaced throughout the definition of *within-function* by a new symbol which is then defined as you specify. See the section on function encapsulation (section 10.10, page 153) for more information.

(:internal *function-spec number*)

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

(*symbol property*)

If *symbol* is not recognized as one of the keywords above, this function spec is the same as (:property *symbol property*). This is provided for compatibility with Maclisp, which allows this syntax in certain places. This form should be avoided since *symbol* might conflict with some existing or future keyword.

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

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

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

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

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

10.3 Simple Function Definitions

defun *Special Form*

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

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

name is the function spec you wish to define as a function. The *lambda-list* is a list of the names to give to the arguments of the function. Actually, it is a little more general than that; it can contain *lambda-list keywords* such as &optional and &rest. (These keywords are explained in section 3.2, page 20 and other keywords are explained in section 10.7, page 148.) See page 145 for some additional syntactic features of defun.

defun creates a list which looks like

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

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

Examples:

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

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

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

A declaration (a list starting with declare) can appear as the first element of the body. It is equivalent to a local-declare (see page 201) surrounding the entire defun form. For example,

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

is equivalent to and preferable to

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

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

A documentation string can also appear as the first element of the body (following the declaration, if there is one). (It shouldn't be the only thing in the body; otherwise it is the value returned by the function and so is not interpreted as documentation. A string as an element of a body other than the last element is only evaluated for side-effect, and since evaluation of strings has no side effects, they aren't useful in this position to do any computation, so they are interpreted as documentation.) This documentation string becomes part of the function's debugging info and can be obtained with the function documentation (see page 150). The first line of the string should be a complete sentence which makes sense read by itself, since there are two editor commands to get at the documentation, one of which is "brief" and prints only the first line. Example:

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

defunp *Macro*

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

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

expands into

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

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

For more information on defining functions, and other ways of doing so, see section 10.6, page 145.

10.4 Operations the User Can Perform on Functions

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

To print out the definition of the function spec with indentation to make it legible, use grindef (see page 360). This works only for interpreted functions. If the definition is a compiled function, it can't be printed out as Lisp code, but its compiled code can be printed by the disassemble function (see page 500).

To find out about how to call the function, you can ask to see its documentation, or its argument names. (The argument names are usually chosen to have mnemonic significance for the caller). Use arglist (page 150) to see the argument names and documentation (page 150) to see the documentation string. There are also editor commands for doing these things: the CTRL/SHIFT/D and META/SHIFT/D commands are for looking at a function's documentation, and CTRL/SHIFT/A is for looking at an argument list. CTRL/SHIFT/A does not ask for the function name; it acts on the function which is called by the innermost expression which the cursor is inside. Usually this is the function which will be called by the form you are in the process of writing.

You can see the function's debugging info alist by means of the function debugging-info (see page 150).

When you are debugging, you can use trace (see page 457) to obtain a printout or a break loop whenever the function is called. You can customize the definition of the function, either temporarily or permanently, using advise (see page 460).

10.5 Kinds of Functions

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

Before we start classifying the functions, we'll first discuss something about how the evaluator works. As we said in the basic description of evaluation on page 13, when the evaluator is given a list whose first element is a symbol, the form may be a function form, a special form, or a macro form. If the definition of the symbol is a function, then the function is just applied to the result of evaluating the rest of the subforms. If the definition is a cons whose car is macro, then it is a macro form; these are explained in chapter 17, page 208. What about special forms?

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

Special functions are written like regular functions except that the keywords &quote and &eval (see section 10.7, page 148) are used to make some of the arguments be "quoted" arguments. The evaluator looks at the pattern in which arguments to the special function are "quoted" or not, and it calls the special function in a special way: for each regular argument, it passes the result of evaluating the corresponding subform, but for each "quoted" argument, it passes the subform itself without evaluating it first. For example, cond works by having a special function that takes a "quoted" &rest argument; when this function is called it is passed a list of cond clauses as its argument.

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

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

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

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

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

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

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

10.5.1 Interpreted Functions

An interpreted function is a piece of list structure which represents a program according to the rules of the Lisp interpreter. Unlike other kinds of functions, an interpreted function can be printed out and read back in (it has a printed representation that the reader understands), can be pretty-printed (see page 360), and can be opened up and examined with the usual functions for list-structure manipulation.

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

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

The symbol lambda identifies this list as a lambda function. *lambda-list* is a description of what arguments the function takes; see section 3.2, page 20 for details. The *forms* make up the body of the function. When the function is called, the argument variables are bound to the values of the arguments as described by *lambda-list*, and then the forms in the body are evaluated, one by one. The value of the function is the value of its last form.

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

```
    (named-lambda name lambda-list bodyforms...)
```

If the *name* slot contains a symbol, it is the function's name. Otherwise it is a list whose car is the name and whose cdr is the function's debugging information alist. See debugging-info, page 150. Note that the name need not be a symbol; it can be any function spec. For example,

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

will give foo a bar property whose value is

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

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

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

The difference between a subst and a lambda is the way they are handled by the compiler. A call to a normal function is compiled as a *closed subroutine*; the compiler generates code to compute the values of the arguments and then apply the function to those values. A call to a subst is compiled as an *open subroutine*; the compiler incorporates the body forms of the subst into the function being compiled, substituting the argument forms for references to the variables in the subst's *lambda-list*. This is a simple-minded but useful facility for *open* or *in-line coded* functions. It is simple-minded because the argument forms can be evaluated multiple times or out of order, and so the semantics of a subst may not be the same in the interpreter and the compiler. substs are described more fully on page 215, with the explanation of defsubst.

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

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

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

10.5.2 Compiled Functions

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

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

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

The printed representation of a microcoded function looks like:

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

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

10.5.3 Other Kinds of Functions

A closure is a kind of function which contains another function and a set of special variable bindings. When the closure is applied, it puts the bindings into effect and then applies the other function. When that returns, the closure bindings are removed. Closures are made with the function closure. See chapter 11, page 158 for more information. Entities are slightly different from closures; see section 11.4, page 162.

A select-method (dtp-select-method) is an a-list of symbols and functions. When one is called the first argument is looked up in the a-list to find the particular function to be called. This function is applied to the rest of the arguments. The a-list may have a list of symbols in place of a symbol, in which case the associated function is called if the first argument is any of the symbols on the list. If cdr of last of the a-list is not nil, it is a *default handler* function, which gets called if the message key is not found in the a-list. Select-methods can be created with the defselect special form (see page 147).

An instance is a message-receiving object which has some state and a table of message-handling functions (called *methods*). Refer to the chapter on flavors (chapter 20, page 279) for further information.

An array can be used as a function. The arguments to the array are the indices and the value is the contents of the element of the array. This works this way for Maclisp compatibility and is not recommended usage. Use aref (page 113) instead.

A stack group can be called as a function. This is one way to pass control to another stack group. See chapter 12, page 163.

10.6 Function-Defining Special Forms

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

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

These special forms are used in writing programs because the function names and bodies are constants. Programs that define functions usually want to compute the functions and their names, so they use fdefine. See page 149.

All of these function-defining special forms alter only the basic definition of the function spec. Encapsulations are preserved. See section 10.10, page 153.

The special forms only create interpreted functions. There is no special way of defining a compiled function. Compiled functions are made by compiling interpreted ones. The same special form which defines the interpreted function, when processed by the compiler, yields the compiled function. See chapter 16, page 197 for details.

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

The defun special form (and the defunp macro which expands into a defun) are used for creating ordinary interpreted functions (see page 139).

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

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

If *lambda-list* is a non-nil symbol instead of a list, the function is recognized as a Maclisp *lexpr* and it is converted in such a way that the arg, setarg, and listify functions can be used to access its arguments (see page 29).

The defsubst special form is used to create substitutible functions. It is used just like defun but produces a list starting with named-subst instead of one starting with named-lambda. The named-subst function acts just like the corresponding named-lambda function when applied, but it can also be open-coded (incorporated into its callers) by the compiler. See page 215 for full information.

The macro special form is the primitive means of creating a macro. It gives a function spec a definition which is a macro definition rather than a actual function. A macro is not a function because it cannot be applied, but it *can* appear as the car of a form to be evaluated. Most macros are created with the more powerful defmacro special form. See chapter 17, page 208.

The defselect special form defines a select-method function. See page 147.

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

def *Special Form*

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

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

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

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

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

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

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

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

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

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

@define *Macro*

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

defun-compatibility *x*

This function is used by defun and the compiler to convert Maclisp-style lexpr, fexpr, and macro defuns to Zetalisp definitions. *x* should be the cdr of a (defun ...) form. defun-compatibility will return a corresponding (defun ...) or (macro ...) form, in the usual Zetalisp format. You shouldn't ever need to call this yourself.

defselect *Special Form*

defselect defines a function which is a select-method. This function contains a table of subfunctions; when it is called, the first argument, a keyword symbol, is looked up in the table to determine which subfunction to call. Each subfunction can take a different number of arguments, and have a different pattern of &optional and &rest arguments. defselect is useful for a variety of "dispatching" jobs. By analogy with the more general message passing facilities described in chapter 20, page 279, the subfunctions are sometimes called *methods* and the first argument is sometimes called a *message*.

The special form looks like

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

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

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

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

A method subform can instead look like:

```
  (keyword   symbol)
```

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

10.7 Lambda-List Keywords

This section documents all the keywords that may appear in the "lambda-list" (argument list) (see section 3.2, page 20) of a function, a macro, or a special form. Some of them are allowed everywhere, while others are only allowed in one of these contexts; those are so indicated.

lambda-list-keywords *Variable*

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

&optional	Separates the required arguments of a function from the optional arguments. See section 3.2, page 20.
&rest	Separates the required and optional arguments of a function from the rest argument. There may be only one rest argument. See page 21 for full information about rest arguments. See section 3.2, page 20.
&aux	Separates the arguments of a function from the auxiliary variables. Following &aux you can put entries of the form (*variable initial-value-form*) or just *variable* if you want it initialized to nil or don't care what the initial value is.
&special	Declares the following arguments and/or auxiliary variables to be special within the scope of this function.
&local	Turns off a preceding &special for the variables which follow.
&functional	Preceding an argument, tells the compiler that the value of this argument will be a function. When a caller of this function is compiled, if it passes a quoted constant argument which looks like a function (a list beginning with the symbol lambda) the compiler will know that it is intended to be a function rather than a list that happens to start with that symbol, and will compile it.
&quote	Declares that the following arguments are not to be evaluated. This is how you create a special function. See the caveats about special forms, on page 142.
&eval	Turns off a preceding &quote for the arguments which follow.
&list-of	This is for macros defined by defmacro only. Refer to page 227.
&body	This is for macros defined by defmacro only. It is similar to &rest, but declares to grindef and the code-formatting module of the editor that the body forms of a special form follow and should be indented accordingly. Refer to page 227.

10.8 How Programs Manipulate Definitions

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

This is the primitive which defun and everything else in the system uses to change the definition of a function spec. If *carefully* is non-nil, which it usually should be, then only the basic definition is changed, the previous basic definition is saved if possible (see undefun, page 150), and any encapsulations of the function such as tracing and advice are carried over from the old definition to the new definition. *carefully* also causes the user to be queried if the function spec is being redefined by a file different from the one that defined it originally, or if *function-spec* belongs to a package other than the current one. However, these warnings are suppressed if either argument *no-query* is non-nil, or if the global variable inhibit-fdefine-warnings is t.

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

If *function-spec* is a symbol, and it was already defined as a function, and *carefully* is non-nil, the symbol's :previous-definition property is used to save the previous definition. If the previous definition is an interpreted function, it is also saved on the :previous-expr-definition property. These properties are used by the undefun function (page 150), which restores the previous definition, and the uncompile function (page 197), which restores the previous interpreted definition. These things are also done for :method function specs, using the property list of the flavor-method-symbol (see page 294).

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

inhibit-fdefine-warnings *Variable*

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

sys:fdefine-file-pathname *Variable*

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

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

This function is obsolete. It is equivalent to

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

fdefinedp *function-spec*

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

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

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

si:fdefinition-symbol-or-location *function-spec*
This attempts to return a symbol which is equivalent as a function spec to the one supplied. The symbol is not created specially so that si:fdefinition-symbol-or-location can return it. Rather, some kinds of function specs are implemented in such a way that a symbol is already part of the data structure and used to hold the function. si:fdefinition-symbol-or-location is the way to get that symbol. Supplying that symbol as a function spec is equivalent to supplying *function-spec*; in addition, the previous definitions of *function-spec* are stored as properties on that symbol.

For those types of function specs which do not use a symbol's function cell to point to the definition, si:fdefinition-symbol-or-location returns a locative to the cell which is used. Don't do get or putprop on this!

undefun *function-spec*
If *function-spec* has a saved previous basic definition, this interchanges the current and previous basic definitions, leaving the encapsulations alone. This undoes the effect of a defun, compile, etc. See also uncompile (page 197).

10.9 How Programs Examine Functions

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

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

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

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

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

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

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

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

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

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

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

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

args-info *function*

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

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

Here are the fields:

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

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

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

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

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

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

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

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

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

10.10 Encapsulations

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

Only advanced users should ever need to use encapsulations directly via the primitives explained in this section. The most common things to do with encapsulations are provided as higher-level, easier-to-use features: trace (see page 457) and advise (see page 460).

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

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

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

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

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

An encapsulation is created with the macro si:encapsulate.

si:encapsulate *Macro*

A call to si:encapsulate looks like

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

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

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

type evaluates to a symbol which identifies the purpose of the encapsulation; it says what the application is. For example, it could be **advise** or **trace**. The list of possible types is defined by the system because encapsulations are supposed to be kept in an order according to their type (see **si:encapsulation-standard-order**, page 155). *type* should have an **si:encapsulation-grind-function** property which tells **grindef** what to do with an encapsulation of this type.

body-form is a form which evaluates to the body of the encapsulation-definition, the code to be executed when it is called. Backquote is typically used for this expression; see section 17.2.2, page 211. **si:encapsulate** is a macro because, while *body* is being evaluated, the variable **si:encapsulated-function** is bound to a list of the form **(function** *uninterned-symbol***)**, referring to the uninterned symbol used to hold the prior definition of *function-spec*. If **si:encapsulate** were a function, *body-form* would just get evaluated normally by the evaluator before **si:encapsulate** ever got invoked, and so there would be no opportunity to bind **si:encapsulated-function**. The form *body-form* should contain **(apply ,si:encapsulated-function arglist)** somewhere if the encapsulation is to live up to its name and truly serve to encapsulate the original definition. (The variable **arglist** is bound by some of the code which the **si:encapsulate** macro produces automatically. When the body of the encapsulation is run **arglist**'s value will be the list of the arguments which the encapsulation received.)

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

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

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

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

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

si:encapsulation-standard-order *Variable*

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

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

advise encapsulations are used to hold advice (see page 460). trace encapsulations are used for implementing tracing (see page 457). si:rename-within encapsulations are used to record the fact that function specs of the form (:within *within-function altered-function*) have been defined. The encapsulation goes on *within-function* (see section 10.10.1, page 156 for more information).

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

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

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

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

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

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

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

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

returns the basic definition of foo, and

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

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

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

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

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

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

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

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

10.10.1 Rename-Within Encapsulations

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

How is this used?

If you define, advise, or trace (:within foo bar), then bar gets renamed to altered-bar-within-foo wherever it is called from foo, and foo gets a si:rename-within encapsulation to record the fact. The purpose of the encapsulation is to enable various parts of the system to do what seems natural to the user. For example, grindef (see page 360) notices the encapsulation, and so knows to print bar instead of altered-bar-within-foo, when grinding the definition of foo.

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

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

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

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

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

11. Closures

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

11.1 What a Closure Is

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

For example, consider the following sequence of Lisp forms:

```
(setq a 3)

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

(print a)
```

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

The form (**closure** *var-list function*), where *var-list* is a list of variables and *function* is any function, creates and returns a closure. When this closure is applied to some arguments, all of the value cells of the variables on *var-list* are saved away, and the value cells that those variables had *at the time* **closure** *was called* (that is, at the time the closure was created) are made to be the value cells of the symbols. Then *function* is applied to the argument. (This paragraph is somewhat complex, but it completely describes the operation of closures; if you don't understand it, come back and read it again after reading the next two paragraphs.)

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

Now, if we evaluate the form

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

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

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

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

Because of the way closures are implemented, the variables to be closed over must not get turned into "local variables" by the compiler. Therefore, all such variables must be declared special. This can be done with an explicit declare (see page 200), with a special form such as defvar (page 19), or with let-closed (page 161). In simple cases, a local-declare around the binding will do the job. Usually the compiler can tell when a special declaration is missing, but in the case of making a closure the compiler detects this after already acting on the assumption that the variable is local, by which time it is too late to fix things. The compiler will warn you if this happens.

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

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

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

11.2 Examples of the Use of Closures

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

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

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

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

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

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

```
(declare (special a b))

(defun foo ()
   (setq a 5))

(defun bar ()
   (cons a b))

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

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

11.3 Closure-Manipulating Functions

closure *var-list function*

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

To test whether an object is a closure, use the **closurep** predicate (see page 10). The **typep** function will return the symbol **closure** if given a closure. **(typep** *x* **'closure)** is equivalent to **(closurep** *x***)**.

symeval-in-closure *closure symbol*

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

set-in-closure *closure symbol x*

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

locate-in-closure *closure symbol*

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

closure-alist *closure*

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

closure-function *closure*

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

let-closed ((*variable value*)...) *function* *Special Form*

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

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

macro-expands into

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

11.4 Entities

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

To some degree, entities are made obsolete by flavors (see chapter 20, page 279). The use of entities as message-receiving objects is explained in section 20.14, page 311.

entity *variable-list function*
Returns a newly constructed entity. This function is just like the function closure except that it returns an entity instead of a closure.

To test whether an object is an entity, use the entityp predicate (see page 10). The functions symeval-in-closure, closure-alist, closure-function, etc. also operate on entities.

12. Stack Groups

A *stack group* (usually abbreviated "SG") is a type of Lisp object useful for implementation of certain advanced control structures such as coroutines and generators. Processes, which are a kind of coroutine, are built on top of stack groups (see chapter 25, page 428). A stack group represents a computation and its internal state, including the Lisp stack.

At any time, the computation being performed by the Lisp Machine is associated with one stack group, called the *current* or *running* stack group. The operation of making some stack group be the current stack group is called a *resumption* or a *stack group switch*; the previously running stack group is said to have *resumed* the new stack group. The *resume* operation has two parts: first, the state of the running computation is saved away inside the current stack group, and secondly the state saved in the new stack group is restored, and the new stack group is made current. Then the computation of the new stack group resumes its course.

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

When the state of the current stack group is saved away, all of its bindings are undone, and when the state is restored, the bindings are put back. Note that although bindings are temporarily undone, unwind-protect handlers are *not* run by a stack-group switch (see **let-globally**, page 17).

Each stack group is a separate environment for purposes of function calling, throwing, dynamic variable binding, and condition signalling. All stack groups run in the same address space, thus they share the same Lisp data and the same global (not lambda-bound) variables.

When a new stack group is created, it is empty: it doesn't contain the state of any computation, so it can't be resumed. In order to get things going, the stack group must be set to an initial state. This is done by "presetting" the stack group. To preset a stack group, you supply a function and a set of arguments. The stack group is placed in such a state that when it is first resumed, this function will call those arguments. The function is called the "initial" function of the stack group.

12.1 Resuming of Stack Groups

The interesting thing that happens to stack groups is that they resume each other. When one stack group resumes a second stack group, the current state of Lisp execution is saved away in the first stack group, and is restored from the second stack group. Resuming is also called "switching stack groups".

At any time, there is one stack group associated with the current computation; it is called the current stack group. The computations associated with other stack groups have their states saved away in memory, and they are not computing. So the only stack group that can do anything at all, in particular resuming other stack groups, is the current one.

You can look at things from the point of view of one computation. Suppose it is running along, and it resumes some stack group. Its state is saved away into the current stack group, and the computation associated with the one it called starts up. The original computation lies dormant in the original stack group, while other computations go around resuming each other, until finally the original stack group is resumed by someone. Then the computation is restored from the stack group and gets to run again.

There are several ways that the current stack group can resume other stack groups. This section describes all of them.

Associated with each stack group is a *resumer*. The resumer is nil or another stack group. Some forms of resuming examine and alter the resumer of some stack groups.

Resuming has another ability: it can transmit a Lisp object from the old stack group to the new stack group. Each stack group specifies a value to transmit whenever it resumes another stack group; whenever a stack group is resumed, it receives a value.

In the descriptions below, let *c* stand for the current stack group, *s* stand for some other stack group, and *x* stand for any arbitrary Lisp object.

Stack groups can be used as functions. They accept one argument. If *c* calls *s* as a function with one argument *x*, then *s* is resumed, and the object transmitted is *x*. When *c* is resumed (usually—but not necessarily—by *s*), the object transmitted by that resumption will be returned as the value of the call to *s*. This is one of the simple ways to resume a stack group: call it as a function. The value you transmit is the argument to the function, and the value you receive is the value returned from the function. Furthermore, this form of resuming sets *s*'s resumer to be *c*.

Another way to resume a stack group is to use stack-group-return. Rather than allowing you to specify which stack group to resume, this function always resumes the resumer of the current stack group. Thus, this is a good way to resume whoever it was who resumed *you*, assuming he did it by function-calling. stack-group-return takes one argument which is the object to transmit. It returns when someone resumes the current stack group, and returns one value, the object that was transmitted by that resumption. stack-group-return does not affect the resumer of any stack group.

The most fundamental way to do resuming is with stack-group-resume, which takes two arguments: the stack group, and a value to transmit. It returns when someone resumes the current stack group, returning the value that was transmitted by that resumption, and does not affect any stack group's resumer.

If the initial function of *c* attempts to return a value *x*, the regular kind of Lisp function return cannot take place, since the function did not have any caller (it got there when the stack group was initialized). So instead of normal function returning, a "stack group return" happens. *c*'s resumer is resumed, and the value transmitted is *x*. *c* is left in a state ("exhausted") from which it cannot be resumed again; any attempt to resume it will signal an error. Presetting it will make it work again.

Those are the "voluntary" forms of stack group switch; a resumption happens because the computation said it should. There are also two "involuntary" forms, in which another stack group is resumed without the explicit request of the running program.

If an error occurs, the current stack group resumes the error handler stack group. The value transmitted is partially descriptive of the error, and the error handler looks inside the saved state of the erring stack group to get the rest of the information. The error handler recovers from the error by changing the saved state of the erring stack group and then resuming it.

When certain events occur, typically a 1-second clock tick, a *sequence break* occurs. This forces the current stack group to resume a special stack group called the *scheduler* (see section 25.1, page 429). The scheduler implements processes by resuming, one after another, the stack group of each process that is ready to run.

sys:%current-stack-group-previous-stack-group *Variable*

The binding of this variable is the resumer of the current stack group.

sys:%current-stack-group *Variable*

The value of sys:%current-stack-group is the stack group which is currently running. A program can use this variable to get its hands on its own stack group.

12.2 Stack Group States

A stack group has a *state*, which controls what it will do when it is resumed. The code number for the state is returned by the function sys:sg-current-state. This number will be the value of one of the following symbols. Only the states actually used by the current system are documented here; some other codes are defined but not used.

sys:sg-state-active
: The stack group is the current one.

sys:sg-state-resumable
: The stack group is waiting to be resumed, at which time it will pick up its saved machine state and continue doing what it was doing before.

sys:sg-state-awaiting-return
: The stack group called some other stack group as a function. When it is resumed, it will return from that function call.

sys:sg-state-awaiting-initial-call
: The stack group has been preset (see below) but has never been called. When it is resumed, it will call its initial function with the preset arguments.

sys:sg-state-exhausted
: The stack group's initial function has returned. It cannot be resumed.

sys:sg-state-awaiting-error-recovery
: When a stack group gets an error it goes into this state, which prevents anything from happening to it until the error handler has looked at it. In the meantime it cannot be resumed.

sys:sg-state-invoke-call-on-return
: When the stack group is resumed, it will call a function. The function and arguments are already set up on the stack. The debugger uses this to force the stack group being debugged to do things.

12.3 Stack Group Functions

make-stack-group *name* &optional *options*

This creates and returns a new stack group. *name* may be any symbol or string; it is used in the stack group's printed representation. *options* is a list of alternating keywords and values. The options are not too useful; most calls to make-stack-group don't need any options at all. The options are:

:sg-area
: The area in which to create the stack group structure itself. Defaults to the default area (the value of default-cons-area).

:regular-pdl-area
: The area in which to create the regular PDL. Note that this may not be any area; only certain areas will do, because regular PDLs are cached in a hardware device called the *pdl buffer*. The default is sys:pdl-area.

:special-pdl-area
: The area in which to create the special PDL. Defaults to the default area (the value of default-cons-area).

:regular-pdl-size
: Length of the regular PDL to be created. Defaults to 3000.

:special-pdl-size
: Length of the special PDL to be created. Defaults to 2000.

:swap-sv-on-call-out
:swap-sv-of-sg-that-calls-me
: These flags default to 1. If these are 0, the system does not maintain separate binding environments for each stack group. You do not want to use this feature.

:trap-enable
: This determines what to do if a microcode error occurs. If it is 1 the system tries to handle the error; if it is 0 the machine halts. Defaults to 1.

:safe If this flag is 1 (the default), a strict call-return discipline among stack-groups is enforced. If 0, no restriction on stack-group switching is imposed.

stack-group-preset *stack-group function* &rest *arguments*
This sets up *stack-group* so that when it is resumed, *function* will be applied to *arguments* within the stack group. Both stacks are made empty; all saved state in the stack group is destroyed. stack-group-preset is typically used to initialize a stack group just after it is made, but it may be done to any stack group at any time. Doing this to a stack group which is not exhausted will destroy its present state without properly cleaning up by running unwind-protects.

stack-group-resume *s x*
Resumes *s*, transmitting the value *x*. No stack group's resumer is affected.

stack-group-return *x*
Resumes the current stack group's resumer, transmitting the value *x*. No stack group's resumer is affected.

symeval-in-stack-group *symbol sg*
Evaluates the variable *symbol* in the binding environment of *sg*. If *sg* is the current stack group, this is just symeval. Otherwise it looks inside *sg* to see if *symbol* is bound there; if so, the binding is returned; if not, the global value is returned. If the variable has no value this will get an unbound-variable error.

There are a large number of functions in the sys: and eh: packages for manipulating the internal details of stack groups. These are not documented here as they are not necessary for most users or even system programmers to know about.

12.4 Input/Output in Stack Groups

Because each stack group has its own set of dynamic bindings, a stack group will not inherit its creator's value of terminal-io (see page 343), nor its caller's, unless you make special provision for this. The terminal-io a stack group gets by default is a "background" stream which does not normally expect to be used. If it is used, it will turn into a "background window" which will request the user's attention. Usually this is because an error printout is trying to be printed on the stream. [This will all be explained in the window system documentation.]

If you write a program that uses multiple stack groups, and you want them all to do input and output to the terminal, you should pass the value of terminal-io to the top-level function of each stack group as part of the stack-group-preset, and that function should bind the variable terminal-io.

Another technique is to use a closure as the top-level function of a stack group. This closure can bind terminal-io and any other variables that are desired to be shared between the stack group and its creator.

12.5 An Example of Stack Groups

The canonical coroutine example is the so-called samefringe problem: Given two trees, determine whether they contain the same atoms in the same order, ignoring parenthesis structure. A better way of saying this is, given two binary trees built out of conses, determine whether the sequence of atoms on the fringes of the trees is the same, ignoring differences in the arrangement of the internal skeletons of the two trees. Following the usual rule for trees, nil in the cdr of a cons is to be ignored.

One way of solving this problem is to use *generator* coroutines. We make a generator for each tree. Each time the generator is called it returns the next element of the fringe of its tree. After the generator has examined the entire tree, it returns a special "exhausted" flag. The generator is most naturally written as a recursive function. The use of coroutines, i.e. stack groups, allows the two generators to recurse separately on two different control stacks without having to coordinate with each other.

The program is very simple. Constructing it in the usual bottom-up style, we first write a recursive function which takes a tree and stack-group-returns each element of its fringe. The stack-group-return is how the generator coroutine delivers its output. We could easily test this function by changing stack-group-return to print and trying it on some examples.

```
(defun fringe (tree)
  (cond ((atom tree) (stack-group-return tree))
        (t (fringe (car tree))
           (if (not (null (cdr tree)))
               (fringe (cdr tree))))))
```

Now we package this function inside another, which takes care of returning the special "exhausted" flag.

```
(defun fringe1 (tree exhausted)
  (fringe tree)
  exhausted)
```

The samefringe function takes the two trees as arguments and returns t or nil. It creates two stack groups to act as the two generator coroutines, presets them to run the fringe1 function, then goes into a loop comparing the two fringes. The value is nil if a difference is discovered, or t if they are still the same when the end is reached.

```
(defun samefringe (tree1 tree2)
  (let ((sg1 (make-stack-group "samefringe1"))
        (sg2 (make-stack-group "samefringe2"))
        (exhausted (ncons nil)))
    (stack-group-preset sg1 #'fringe1 tree1 exhausted)
    (stack-group-preset sg2 #'fringe1 tree2 exhausted)
    (do ((v1) (v2)) (nil)
      (setq v1 (funcall sg1 nil)
            v2 (funcall sg2 nil))
      (cond ((neq v1 v2) (return nil))
            ((eq v1 exhausted) (return t))))))
```

Now we test it on a couple of examples.
```
(samefringe '(a b c) '(a (b c))) => t
(samefringe '(a b c) '(a b c d)) => nil
```

The problem with this is that a stack group is quite a large object, and we make two of them every time we compare two fringes. This is a lot of unnecessary overhead. It can easily be eliminated with a modest amount of explicit storage allocation, using the resource facility (see page 82). While we're at it, we can avoid making the exhausted flag fresh each time; its only important property is that it not be an atom.
```
(defresource samefringe-coroutine ()
   :constructor (make-stack-group "for-samefringe"))

(defvar exhausted-flag (ncons nil))

(defun samefringe (tree1 tree2)
  (using-resource (sg1 samefringe-coroutine)
    (using-resource (sg2 samefringe-coroutine)
      (stack-group-preset sg1 #'fringe1 tree1 exhausted-flag)
      (stack-group-preset sg2 #'fringe1 tree2 exhausted-flag)
      (do ((v1) (v2)) (nil)
        (setq v1 (funcall sg1 nil)
              v2 (funcall sg2 nil))
        (cond ((neq v1 v2) (return nil))
              ((eq v1 exhausted-flag) (return t)))))))
```

Now we can compare the fringes of two trees with no allocation of memory whatsoever.

13. Locatives

13.1 Cells and Locatives

A *locative* is a type of Lisp object used as a *pointer* to a *cell*. Locatives are inherently a more "low level" construct than most Lisp objects; they require some knowledge of the nature of the Lisp implementation. Most programmers will never need them.

A *cell* is a machine word which 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 which create locatives to cells; the functions are documented with the kind of object to which they create a pointer. See ap-1, ap-leader, car-location, value-cell-location, etc. The macro locf (see page 230) can be used to convert a form which accesses a cell to one which creates a locative pointer to that cell: for example,

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

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.

13.2 Functions Which Operate on Locatives

Either of the functions car and cdr (see page 53) may be given a locative, and will return the contents of the cell at which the locative points.

For example,

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

is the same as

```
(symeval x)
```

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

For example,

```
(rplaca (value-cell-location x) y)
```

is the same as

```
(set x y)
```

If you mix locatives and lists, then it matters whether you use car and rplaca or cdr and rplacd, and care is required. For example, 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 non-special 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.

14. Subprimitives

Subprimitives are functions which are not intended to be used by the average program, only by "system programs". They allow one to manipulate the environment at a level lower than normal Lisp. They are described in this chapter. Subprimitives usually have names which start with a % character. The "primitives" described in other sections of the manual typically use subprimitives to accomplish their work. The subprimitives take the place of machine language in other systems, to some extent. Subprimitives are normally hand-coded in microcode.

There is plenty of stuff in this chapter that is not fully explained; there are terms that are undefined, there are forward references, and so on. Furthermore, most of what is in here is considered subject to change without notice. In fact, this chapter does not exactly belong in this manual, but in some other more low-level manual. Since the latter manual does not exist, it is here for the interim.

Subprimitives by their very nature cannot do full checking. Improper use of subprimitives can destroy the environment. Subprimitives come in varying degrees of dangerousness. Those without a % sign in their name cannot destroy the environment, but are dependent on "internal" details of the Lisp implementation. The ones whose names start with a % sign can violate system conventions if used improperly. The subprimitives are documented here since they need to be documented somewhere, but this manual does not document all the things you need to know in order to use them. Still other subprimitives are not documented here because they are very specialized. Most of these are never used explicitly by a programmer; the compiler inserts them into the program to perform operations which are expressed differently in the source code.

The most common problem you can cause using subprimitives, though by no means the only one, is to create illegal pointers: pointers that are, for one reason or another, according to storage conventions, not allowed to exist. The storage conventions are not documented; as we said, you have to be an expert to correctly use a lot of the functions in this chapter. If you create such an illegal pointer, it probably will not be detected immediately, but later on parts of the system may see it, notice that it is illegal, and (probably) halt the Lisp Machine.

In a certain sense car, cdr, rplaca, and rplacd are subprimitives. If these are given a locative instead of a list, they will access or modify the cell addressed by the locative without regard to what object the cell is inside. Subprimitives can be used to create locatives to strange places.

14.1 Data Types

data-type *arg*

data-type returns a symbol which is the name for the internal data-type of the "pointer" which represents *arg*. Note that some types as seen by the user are not distinguished from each other at this level, and some user types may be represented by more than one internal type. For example, dtp-extended-number is the symbol that data-type would return for either a flonum or a bignum, even though those two types are quite different. The typep function (page 10) is a higher-level primitive which is more useful in most cases; normal programs should always use typep rather than data-type. Some of these type codes are internal tag fields that are never used in pointers that represent Lisp objects at all, but they are documented here anyway.

dtp-symbol	The object is a symbol.
dtp-fix	The object is a fixnum; the numeric value is contained in the address field of the pointer.
dtp-small-flonum	The object is a small flonum; the numeric value is contained in the address field of the pointer.
dtp-extended-number	The object is a flonum or a bignum. This value will also be used for future numeric types.
dtp-list	The object is a cons.
dtp-locative	The object is a locative pointer.
dtp-array-pointer	The object is an array.
dtp-fef-pointer	The object is a compiled function.
dtp-u-entry	The object is a microcode entry.
dtp-closure	The object is a closure; see chapter 11, page 158.
dtp-stack-group	The object is a stack-group; see chapter 12, page 163.
dtp-instance	The object is an instance of a flavor, i.e. an "active object". See chapter 20, page 279.
dtp-entity	The object is an entity; see section 11.4, page 162.
dtp-select-method	The object is a "select-method"; see page 144.
dtp-header	An internal type used to mark the first word of a multi-word structure.
dtp-array-header	An internal type used in arrays.
dtp-symbol-header	An internal type used to mark the first word of a symbol.
dtp-instance-header	An internal type used to mark the first word of an instance.
dtp-null	Nothing to do with nil. This is used in unbound value and function cells.
dtp-trap	The zero data-type, which is not used. This hopes to detect microcode bugs.

dtp-free	This type is used to fill free storage, to catch wild references.
dtp-external-value-cell-pointer	An "invisible pointer" used for external value cells, which are part of the closure mechanism (see chapter 11, page 158), and used by compiled code to address value and function cells.
dtp-header-forward	An "invisible pointer" used to indicate that the structure containing it has been moved elsewhere. The "header word" of the structure is replaced by one of these invisible pointers. See the function structure-forward (page 175).
dtp-body-forward	An "invisible pointer" used to indicate that the structure containing it has been moved elsewhere. This points to the word containing the header-forward, which points to the new copy of the structure.
dtp-one-q-forward	An "invisible pointer" used to indicate that the single cell containing it has been moved elsewhere.
dtp-gc-forward	This is used by the copying garbage collector to flag the obsolete copy of an object; it points to the new copy.

q-data-types *Variable*

The value of q-data-types is a list of all of the symbolic names for data types described above under data-type. These are the symbols whose print names begin with "dtp-". The values of these symbols are the internal numeric data-type codes for the various types.

q-data-types *type-code*

Given the internal numeric data-type code, returns the corresponding symbolic name. This "function" is actually an array.

14.2 Forwarding

An *invisible pointer* is a kind of pointer that does not represent a Lisp object, but just resides in memory. There are several kinds of invisible pointer, and there are various rules about where they may or may not appear. The basic property of an invisible pointer is that if the Lisp Machine reads a word of memory and finds an invisible pointer there, instead of seeing the invisible pointer as the result of the read, it does a second read, at the location addressed by the invisible pointer, and returns that as the result instead. Writing behaves in a similar fashion. When the Lisp Machine writes a word of memory it first checks to see if that word contains an invisible pointer; if so it goes to the location pointed to by the invisible pointer and tries to write there instead. Many subprimitives that read and write memory do not do this checking.

The simplest kind of invisible pointer has the data type code dtp-one-q-forward. It is used to forward a single word of memory to someplace else. The invisible pointers with data types dtp-header-forward and dtp-body-forward are used for moving whole Lisp objects (such as cons cells or arrays) somewhere else. The dtp-external-value-cell-pointer is very similar to the dtp-one-q-forward; the difference is that it is not "invisible" to the operation of binding. If the (internal) value cell of a symbol contains a dtp-external-value-cell-pointer that points to some other word (the external value cell), then symeval or set operations on the symbol will consider the pointer to be invisible and use the external value cell, but binding the symbol will save away the **dtp-external-value-cell-pointer** itself, and store the new value into the internal value cell of the symbol. This is how closures are implemented.

dtp-gc-forward is not an invisible pointer at all; it only appears in "old space" and will never be seen by any program other than the garbage collector. When an object is found not to be garbage, and the garbage collector moves it from "old space" to "new space", a **dtp-gc-forward** is left behind to point to the new copy of the object. This ensures that other references to the same object get the same new copy.

structure-forward *old-object new-object*

This causes references to *old-object* to actually reference *new-object*, by storing invisible pointers in *old-object*. It returns *old-object*.

An example of the use of **structure-forward** is **adjust-array-size**. If the array is being made bigger and cannot be expanded in place, a new array is allocated, the contents are copied, and the old array is structure-forwarded to the new one. This forwarding ensures that pointers to the old array, or to cells within it, continue to work. When the garbage collector goes to copy the old array, it notices the forwarding and uses the new array as the copy; thus the overhead of forwarding disappears eventually if garbage collection is in use.

follow-structure-forwarding *object*

Normally returns *object*, but if *object* has been **structure-forward**'ed, returns the object at the end of the chain of forwardings. If *object* is not exactly an object, but a locative to a cell in the middle of an object, a locative to the corresponding cell in the latest copy of the object will be returned.

forward-value-cell *from-symbol to-symbol*

This alters *from-symbol* so that it always has the same value as *to-symbol*, by sharing its value cell. A **dtp-one-q-forward** invisible pointer is stored into *from-symbol*'s value cell. Do not do this while *from-symbol* is **lambda**-bound, as the microcode does not bother to check for that case and something bad will happen when *from-symbol* gets unbound. The microcode check is omitted to speed up binding and unbinding.

To forward one arbitrary cell to another (rather than specifically one value cell to another), given two locatives do

```
(%p-store-tag-and-pointer locative1 dtp-one-q-forward locative2)
```

follow-cell-forwarding *loc evcp-p*

loc is a locative to a cell. Normally *loc* is returned, but if the cell has been forwarded, this follows the chain of forwardings and returns a locative to the final cell. If the cell is part of a structure which has been forwarded, the chain of structure forwardings is followed, too. If *evcp-p* is **t**, external value cell pointers are followed; if it is **nil** they are not.

14.3 Pointer Manipulation

It should again be emphasized that improper use of these functions can damage or destroy the Lisp environment. It is possible to create pointers with illegal data-type, pointers to non-existent objects, and pointers to untyped storage which will completely confuse the garbage collector.

%data-type *x*

Returns the data-type field of *x*, as a fixnum.

%pointer *x*

Returns the pointer field of *x*, as a fixnum. For most types, this is dangerous since the garbage collector can copy the object and change its address.

%make-pointer *data-type pointer*

This makes up a pointer, with *data-type* in the data-type field and *pointer* in the pointer field, and returns it. *data-type* should be an internal numeric data-type code; these are the values of the symbols that start with dtp-. *pointer* may be any object; its pointer field is used. This is most commonly used for changing the type of a pointer. Do not use this to make pointers which are not allowed to be in the machine, such as dtp-null, invisible pointers, etc.

%make-pointer-offset *data-type pointer offset*

This returns a pointer with *data-type* in the data-type field, and *pointer* plus *offset* in the pointer field. The *data-type* and *pointer* arguments are like those of %make-pointer; *offset* may be any object but is usually a fixnum. The types of the arguments are not checked; their pointer fields are simply added together. This is useful for constructing locative pointers into the middle of an object. However, note that it is illegal to have a pointer to untyped data, such as the inside of a FEF or a numeric array.

%pointer-difference *pointer-1 pointer-2*

Returns a fixnum which is *pointer-1* minus *pointer-2*. No type checks are made. For the result to be meaningful, the two pointers must point into the same object, so that their difference cannot change as a result of garbage collection.

14.4 Analyzing Structures

%find-structure-header *pointer*

This subprimitive finds the structure into which *pointer* points, by searching backward for a header. It is a basic low-level function used by such things as the garbage collector. *pointer* is normally a locative, but its data-type is ignored. Note that it is illegal to point into an "unboxed" portion of a structure, for instance the middle of a numeric array.

In structure space, the "containing structure" of a pointer is well-defined by system storage conventions. In list space, it is considered to be the contiguous, cdr-coded segment of list surrounding the location pointed to. If a cons of the list has been copied out by rplacd, the contiguous list includes that pair and ends at that point.

%find-structure-leader *pointer*

This is identical to %find-structure-header, except that if the structure is an array with a leader, this returns a locative pointer to the leader-header, rather than returning the array-pointer itself. Thus the result of %find-structure-leader is always the lowest address in the structure. This is the one used internally by the garbage collector.

%structure-boxed-size *object*

Returns the number of "boxed Q's" in *object*. This is the number of words at the front of the structure which contain normal Lisp objects. Some structures, for example FEFs and numeric arrays, contain additional "unboxed Q's" following their "boxed Q's". Note that the boxed size of a PDL (either regular or special) does not include Q's above the current top of the PDL. Those locations are boxed but their contents is considered garbage, and is not protected by the garbage collector.

%structure-total-size *object*

Returns the total number of words occupied by the representation of *object*, including boxed Q's, unboxed Q's, and garbage Q's off the ends of PDLs.

14.5 Creating Objects

%allocate-and-initialize *data-type header-type header second-word area size*

This is the subprimitive for creating most structured-type objects. *area* is the area in which it is to be created, as a fixnum or a symbol. *size* is the number of words to be allocated. The value returned points to the first word allocated, and has data-type *data-type*. Uninterruptibly, the words allocated are initialized so that storage conventions are preserved at all times. The first word, the header, is initialized to have *header-type* in its data-type field and *header* in its pointer field. The second word is initialized to *second-word*. The remaining words are initialized to nil. The flag bits of all words are set to 0. The cdr codes of all words except the last are set to cdr-next; the cdr code of the last word is set to cdr-nil. It is probably a bad idea to rely on this.

The basic functions for creating list-type objects are cons and make-list; no special subprimitive is needed. Closures, entities, and select-methods are based on lists, but there is no primitive for creating them. To create one, create a list and then use %make-pointer to change the data type from dtp-list to the desired type.

%allocate-and-initialize-array *header data-length leader-length area size*

This is the subprimitive for creating arrays, called only by make-array. It is different from %allocate-and-initialize because arrays have a more complicated header structure.

14.6 Locking Subprimitive

%store-conditional *pointer old new*

This is the basic locking primitive. *pointer* is a locative to a cell which is uninterruptibly read and written. If the contents of the cell is eq to *old*, then it is replaced by *new* and t is returned. Otherwise, nil is returned and the contents of the cell is not changed.

14.7 I/O Device Subprimitives

%unibus-read *address*

Returns the contents of the register at the specified Unibus address, as a fixnum. You must specify a full 18-bit address. This is guaranteed to read the location only once. Since the Lisp Machine Unibus does not support byte operations, this always references a 16-bit word, and so *address* will normally be an even number.

%unibus-write *address data*

Writes the 16-bit number *data* at the specified Unibus address, exactly once.

%xbus-read *io-offset*

Returns the contents of the register at the specified Xbus address. *io-offset* is an offset into the I/O portion of Xbus physical address space. This is guaranteed to read the location exactly once. The returned value can be either a fixnum or a bignum.

%xbus-write *io-offset data*

Writes *data*, which can be a fixnum or a bignum, into the register at the specified Xbus address. *io-offset* is an offset into the I/O portion of Xbus physical address space. This is guaranteed to write the location exactly once.

sys:%xbus-write-sync *w-loc w-data delay sync-loc sync-mask sync-value*

Does (%xbus-write *w-loc w-data*), but first synchronizes to within about one microsecond of a certain condition. The synchronization is achieved by looping until

(= (logand (%xbus-read *sync-loc*) *sync-mask*) *sync-value*)

is false, then looping until it is true, then looping *delay* times. Thus the write happens a specified delay after the leading edge of the synchronization condition. The number of microseconds of delay is roughly one third of *delay*.

sys:%halt

Stops the machine.

14.8 Special Memory Referencing

%p-contents-offset *base-pointer offset*

This checks the cell pointed to by *base-pointer* for a forwarding pointer. Having followed forwarding pointers to the real structure pointed to, it adds *offset* to the resulting forwarded *base-pointer* and returns the contents of that location.

There is no %p-contents, since car performs that operation.

%p-contents-as-locative *pointer*

Given a pointer to a memory location containing a pointer which isn't allowed to be "in the machine" (typically an invisible pointer) this function returns the contents of the location as a dtp-locative. It changes the disallowed data type to dtp-locative so that you can safely look at it and see what it points to.

%p-contents-as-locative-offset *base-pointer offset*

This checks the cell pointed to by *base-pointer* for a forwarding pointer. Having followed forwarding pointers to the real structure pointed to, it adds *offset* to the resulting forwarded *base-pointer*, fetches the contents of that location, and returns it with the data type changed to dtp-locative in case it was a type which isn't allowed to be "in the machine" (typically an invisible pointer). This can be used, for example, to analyze the dtp-external-value-cell-pointer pointers in a FEF, which are used by the compiled code to reference value cells and function cells of symbols.

%p-store-contents *pointer value*

value is stored into the data-type and pointer fields of the location addressed by *pointer*. The cdr-code and flag-bit fields remain unchanged. *value* is returned.

%p-store-contents-offset *value base-pointer offset*

This checks the cell pointed to by *base-pointer* for a forwarding pointer. Having followed forwarding pointers to the real structure pointed to, it adds *offset* to the resulting forwarded *base-pointer*, and stores *value* into the data-type and pointer fields of that location. The cdr-code and flag-bit fields remain unchanged. *value* is returned.

%p-store-tag-and-pointer *pointer miscfields pntrfield*

Creates a *Q* by taking 8 bits from *miscfields* and 24 bits from *pntrfield*, and stores that into the location addressed by *pointer*. The low 5 bits of *miscfields* become the data-type, the next bit becomes the flag-bit, and the top two bits become the cdr-code. This is a good way to store a forwarding pointer from one structure to another (for example).

%p-ldb *ppss pointer*

This is like ldb but gets a byte from the location addressed by *pointer*. Note that you can load bytes out of the data type etc. bits, not just the pointer field, and that the word loaded out of need not be a fixnum. The result returned is always a fixnum.

%p-ldb-offset *ppss base-pointer offset*
This checks the cell pointed to by *base-pointer* for a forwarding pointer. Having followed forwarding pointers to the real structure pointed to, the byte specified by *ppss* is loaded from the contents of the location addressed by the forwarded *base-pointer* plus *offset*, and returned as a fixnum. This is the way to reference byte fields within a structure without violating system storage conventions.

%p-dpb *value ppss pointer*
The *value*, a fixnum, is stored into the byte selected by *ppss* in the word addressed by *pointer*. nil is returned. You can use this to alter data types, cdr codes, etc.

%p-dpb-offset *value ppss base-pointer offset*
This checks the cell pointed to by *base-pointer* for a forwarding pointer. Having followed forwarding pointers to the real structure pointed to, the *value* is stored into the byte specified by *ppss* in the location addressed by the forwarded *base-pointer* plus *offset*. nil is returned. This is the way to alter unboxed data within a structure without violating system storage conventions.

%p-mask-field *ppss pointer*
This is similar to %p-ldb, except that the selected byte is returned in its original position within the word instead of right-aligned.

%p-mask-field-offset *ppss base-pointer offset*
This is similar to %p-ldb-offset, except that the selected byte is returned in its original position within the word instead of right-aligned.

%p-deposit-field *value ppss pointer*
This is similar to %p-dpb, except that the selected byte is stored from the corresponding bits of *value* rather than the right-aligned bits.

%p-deposit-field-offset *value ppss base-pointer offset*
This is similar to %p-dpb-offset, except that the selected byte is stored from the corresponding bits of *value* rather than the right-aligned bits.

%p-pointer *pointer*
Extracts the pointer field of the contents of the location addressed by *pointer* and returns it as a fixnum.

%p-data-type *pointer*
Extracts the data-type field of the contents of the location addressed by *pointer* and returns it as a fixnum.

%p-cdr-code *pointer*
Extracts the cdr-code field of the contents of the location addressed by *pointer* and returns it as a fixnum.

%p-flag-bit *pointer*
: Extracts the flag-bit field of the contents of the location addressed by *pointer* and returns it as a fixnum.

%p-store-pointer *pointer value*
: Clobbers the pointer field of the location addressed by *pointer* to *value*, and returns *value*.

%p-store-data-type *pointer value*
: Clobbers the data-type field of the location addressed by *pointer* to *value*, and returns *value*.

%p-store-cdr-code *pointer value*
: Clobbers the cdr-code field of the location addressed by *pointer* to *value*, and returns *value*.

%p-store-flag-bit *pointer value*
: Clobbers the flag-bit field of the location addressed by *pointer* to *value*, and returns *value*.

%stack-frame-pointer
: Returns a locative pointer to its caller's stack frame. This function is not defined in the interpreted Lisp environment; it only works in compiled code. Since it turns into a "misc" instruction, the "caller's stack frame" really means "the frame for the FEF that executed the %stack-frame-pointer instruction".

14.9 Storage Layout Definitions

The following special variables have values which define the most important attributes of the way Lisp data structures are laid out in storage. In addition to the variables documented here, there are many others which are more specialized. They are not documented in this manual since they are in the system package rather than the global package. The variables whose names start with %% are byte specifiers, intended to be used with subprimitives such as %p-ldb. If you change the value of any of these variables, you will probably bring the machine to a crashing halt.

%%q-cdr-code *Variable*
: The field of a memory word which contains the cdr-code. See section 5.4, page 63.

%%q-flag-bit *Variable*
: The field of a memory word which contains the flag-bit. In most data structures this bit is not used by the system and is available for the user.

%%q-data-type *Variable*
: The field of a memory word which contains the data-type code. See page 173.

%%q-pointer *Variable*
The field of a memory which contains the pointer address, or immediate data.

%%q-pointer-within-page *Variable*
The field of a memory word which contains the part of the address that lies within a single page.

%%q-typed-pointer *Variable*
The concatenation of the %%q-data-type and %%q-pointer fields.

%%q-all-but-typed-pointer *Variable*
The field of a memory word which contains the tag fields, %%q-cdr-code and %%q-flag-bit.

%%q-all-but-pointer *Variable*
The concatenation of all fields of a memory word except for %%q-pointer.

%%q-all-but-cdr-code *Variable*
The concatenation of all fields of a memory word except for %%q-cdr-code.

%%q-high-half *Variable*
%%q-low-half *Variable*
The two halves of a memory word. These fields are only used in storing compiled code.

cdr-normal *Variable*
cdr-next *Variable*
cdr-nil *Variable*
cdr-error *Variable*
The values of these four variables are the numeric values which go in the cdr-code field of a memory word. See section 5.4, page 63 for the details of cdr-coding.

14.10 Function-Calling Subprimitives

These subprimitives can be used (carefully!) to call a function with the number of arguments variable at run time. They only work in compiled code and are not defined in the interpreted Lisp environment. The preferred higher-level primitive is lexpr-funcall (page 24).

%open-call-block *function n-adi-pairs destination*
Starts a call to *function*. *n-adi-pairs* is the number of pairs of additional information words already %push'ed; normally this should be 0. *destination* is where to put the result; the useful values are 0 for the value to be ignored, 1 for the value to go onto the stack, 3 for the value to be the last argument to the previous open call block, and 4 for the value to be returned from this frame.

%push *value*
Pushes *value* onto the stack. Use this to push the arguments.

%activate-open-call-block
Causes the call to happen.

%pop
Pops the top value off of the stack and returns it as its value. Use this to recover the result from a call made by %open-call-block with a destination of 1.

%assure-pdl-room *n-words*
Call this before doing a sequence of %push's or %open-call-blocks which will add *n-words* to the current frame. This subprimitive checks that the frame will not exceed the maximum legal frame size, which is 255 words including all overhead. This limit is dictated by the way stack frames are linked together. If the frame is going to exceed the legal limit, %assure-pdl-room will signal an error.

14.11 Lambda-Binding Subprimitive

bind *locative value*
Binds the cell pointed to by *locative* to *x*, in the caller's environment. This function is not defined in the interpreted Lisp environment; it only works from compiled code. Since it turns into an instruction, the "caller's environment" really means "the binding block for the stack frame that executed the bind instruction". The preferred higher-level primitives which turn into this are let (page 16), let-if (page 17), and progv (page 18).
[This will be renamed to %bind in the future.]

14.12 The Paging System

[Someday this may discuss how it works.]

si:wire-page *address* &optional (*wire-p* t)
If *wire-p* is t, the page containing *address* is *wired-down*; that is, it cannot be paged-out. If *wire-p* is nil, the page ceases to be wired-down.

si:unwire-page *address*
(si:unwire-page *address*) is the same as (si:wire-page *address* nil).

sys:page-in-structure *object*
Makes sure that the storage which represents *object* is in main memory. Any pages which have been swapped out to disk are read in, using as few disk operations as possible. Consecutive disk pages are transferred together, taking advantage of the full speed of the disk. If *object* is large, this will be much faster than bringing the pages in one at a time on demand. The storage occupied by *object* is defined by the %find-structure-leader and %structure-total-size subprimitives.

sys:page-in-array *array* &optional *from to*
This is a version of sys:page-in-structure which can bring in a portion of an array. *from* and *to* are lists of subscripts; if they are shorter than the dimensionality of *array*, the remaining subscripts are assumed to be zero.

sys:page-in-words *address n-words*
Any pages in the range of address space starting at *address* and continuing for *n-words* which have been swapped out to disk are read in with as few disk operations as possible.

sys:page-in-area *area-number*
sys:page-in-region *region-number*
All swapped-out pages of the specified region or area are brought into main memory.

sys:page-out-structure *object*
sys:page-out-array *array* &optional *from to*
sys:page-out-words *address n-words*
sys:page-out-area *area-number*
sys:page-out-region *region-number*
These are similar to the above, except that take pages out of main memory rather than bringing them in. Any modified pages are written to disk, using as few disk operations as possible. The pages are then made flushable; if they are not touched again soon their memory will be reclaimed for other pages. Use these operations when you are done with a large object, to make the virtual memory system prefer reclaiming that object's memory over swapping something else out.

sys:%change-page-status *virtual-address swap-status access-status-and-meta-bits*
The page hash table entry for the page containing *virtual-address* is found and altered as specified. t is returned if it was found, nil if it was not (presumably the page is swapped out.) *swap-status* and *access-status-and-meta-bits* can be nil if those fields are not to be changed. This doesn't make any error checks; you can really screw things up if you call it with the wrong arguments.

sys:%compute-page-hash *virtual-address*
This makes the hashing function for the page hash table available to the user.

sys:%create-physical-page *physical-address*
This is used when adjusting the size of real memory available to the machine. It adds an entry for the page frame at *physical-address* to the page hash table, with virtual address -1, swap status flushable, and map status 120 (read only). This doesn't make error checks; you can really screw things up if you call it with the wrong arguments.

sys:%delete-physical-page *physical-address*
If there is a page in the page frame at *physical-address*, it is swapped out and its entry is deleted from the page hash table, making that page frame unavailable for swapping in of pages in the future. This doesn't make error checks; you can really screw things up if you call it with the wrong arguments.

sys:%disk-restore *high-16-bits low-16-bits*

Loads virtual memory from the partition named by the concatenation of the two 16-bit arguments, and starts executing it. The name 0 refers to the default load (the one the machine loads when it is started up). This is the primitive used by disk-restore (see page 423).

sys:%disk-save *physical-mem-size high-16-bits low-16-bits*

Copies virtual memory into the partition named by the concatenation of the two 16-bit arguments (0 means the default), then restarts the world, as if it had just been restored. The *physical-mem-size* argument should come from %sys-com-memory-size in system-communication-area. This is the primitive used by disk-save (see page 424).

14.13 Closure Subprimitives

These functions deal with things like what closures deal with: the distinction between internal and external value cells and control over how they work.

sys:%binding-instances *list-of-symbols*

This is the primitive that could be used by closure. First, if any of the symbols in *list-of-symbols* has no external value cell, a new external value cell is created for it, with the contents of the internal value cell. Then a list of locatives, twice as long as *list-of-symbols*, is created and returned. The elements are grouped in pairs: pointers to the internal and external value cells, respectively, of each of the symbols. closure could have been defined by:

```
(defun closure (variables function)
  (%make-pointer dtp-closure
    (cons function (sys:%binding-instances variables))))
```

sys:%using-binding-instances *instance-list*

This function is the primitive operation that invocation of closures could use. It takes a list such as sys:%binding-instances returns, and for each pair of elements in the list, it "adds" a binding to the current stack frame, in the same manner that the bind function (which should be called %bind) does. These bindings remain in effect until the frame returns or is unwound.

sys:%using-binding-instances checks for redundant bindings and ignores them. (A binding is redundant if the symbol is already bound to the desired external value cell). This check avoids excessive growth of the special pdl in some cases and is also made by the microcode which invokes closures, entities, and instances.

sys:%internal-value-cell *symbol*

Returns the contents of the internal value cell of *symbol*. dtp-one-q-forward pointers are considered invisible, as usual, but dtp-external-value-cell-pointers are *not*; this function can return a dtp-external-value-cell-pointer. Such pointers will be considered invisible as soon as they leave the "inside of the machine", meaning internal registers and the stack.

14.14 Microcode Variables

The following variables' values actually reside in the scratchpad memory of the processor. They are put there by dtp-one-q-forward invisible pointers. The values of these variables are used by the microcode. Many of these variables are highly internal and you shouldn't expect to understand them.

%microcode-version-number *Variable*
This is the version number of the currently-loaded microcode, obtained from the version number of the microcode source file.

sys:%number-of-micro-entries *Variable*
Size of micro-code-entry-area and related areas.

default-cons-area is documented on page 193.

sys:number-cons-area *Variable*
The area number of the area where bignums and flonums are consed. Normally this variable contains the value of sys:extra-pdl-area, which enables the "temporary storage" feature for numbers, saving garbage collection overhead.

sys:%current-stack-group and sys:%current-stack-group-previous-stack-group are documented on page 165.

sys:%current-stack-group-state *Variable*
The sg-state of the currently-running stack group.

sys:%current-stack-group-calling-args-pointer *Variable*
The argument list of the currently-running stack group.

sys:%current-stack-group-calling-args-number *Variable*
The number of arguments to the currently-running stack group.

sys:%trap-micro-pc *Variable*
The microcode address of the most recent error trap.

sys:%initial-fef *Variable*
The function which is called when the machine starts up. Normally this is the definition of si:lisp-top-level.

sys:%initial-stack-group *Variable*
The stack group in which the machine starts up.

sys:%error-handler-stack-group *Variable*
The stack group which receives control when a microcode-detected error occurs. This stack group cleans up, signals the appropriate condition, or assigns a stack group to run the debugger on the erring stack group.

sys:%scheduler-stack-group *Variable*
The stack group which receives control when a sequence break occurs.

sys:%chaos-csr-address *Variable*
A fixnum which is the virtual address which maps to the Unibus location of the Chaosnet interface.

%mar-low *Variable*
A fixnum which is the inclusive lower bound of the region of virtual memory subject to the MAR feature (see section 26.7, page 467).

%mar-high *Variable*
A fixnum which is the inclusive upper bound of the region of virtual memory subject to the MAR feature (see section 26.7, page 467).

sys:%inhibit-read-only *Variable*
If non-nil, you can write into read-only areas. This is used by fasload.

self is documented on page 297.

inhibit-scheduling-flag is documented on page 430.

inhibit-scavenging-flag *Variable*
If non-nil, the scavenger is turned off. The scavenger is the quasi-asynchronous portion of the garbage collector, which normally runs during consing operations.

sys:%region-cons-alarm *Variable*
Incremented whenever a new region is allocated.

sys:%page-cons-alarm *Variable*
Increments whenever a new page is allocated.

sys:%gc-flip-ready *Variable*
t while the scavenger is running, nil when there are no pointers to oldspace.

sys:%gc-generation-number *Variable*
A fixnum which is incremented whenever the garbage collector flips, converting one or more regions from newspace to oldspace. If this number has changed, the %pointer of an object may have changed.

sys:%disk-run-light *Variable*
A fixnum which is the virtual address of the TV buffer location of the run-light which lights up when the disk is active. This plus 2 is the address of the run-light for the processor. This minus 2 is the address of the run-light for the garbage collector.

`sys:%loaded-band` *Variable*
: A fixnum which contains the high 24 bits of the name of the disk partition from which virtual memory was booted. Used to create the greeting message.

`sys:%disk-blocks-per-track` *Variable*
`sys:%disk-blocks-per-cylinder` *Variable*
: Configuration of the disk being used for paging. Don't change these!

`sys:%read-compare-enables` *Variable*
: A fixnum which controls extra disk error-checking. Bit 0 enables read-compare after a read, bit 1 enables read-compare after a write. Normally this is 0.

`sys:currently-prepared-sheet` *Variable*
: Used for communication between the window system and the microcoded graphics primitives.

The next four have to do with a metering system which is not yet documented in this manual.

`sys:%meter-global-enable` *Variable*
: t if the metering system is turned on for all stack-groups.

`sys:%meter-buffer-pointer` *Variable*
: A temporary buffer used by the metering system.

`sys:%meter-disk-address` *Variable*
: Where the metering system writes its next block of results on the disk.

`sys:%meter-disk-count` *Variable*
: The number of disk blocks remaining for recording of metering information.

`sys:a-memory-location-names` *Variable*
: A list of all of the above symbols (and any others added after this documentation was written).

14.15 Meters

`read-meter` *name*
: Returns the contents of the microcode meter named *name*, which can be a fixnum or a bignum. *name* must be one the symbols listed below.

`write-meter` *name value*
: Writes *value*, a fixnum or a bignum, into the microcode meter named *name*. *name* must be one the symbols listed below.

The microcode meters are as follows:

sys:%count-chaos-transmit-aborts *Meter*
The number of times transmission on the Chaosnet was aborted, either by a collision or because the receiver was busy.

sys:%count-cons-work *Meter*
sys:%count-scavenger-work *Meter*
Internal state of the garbage collection algorithm.

sys:%tv-clock-rate *Meter*
The number of TV frames per clock sequence break. The default value is 67 which causes clock sequence breaks to happen about once per second.

sys:%count-first-level-map-reloads *Meter*
The number of times the first-level virtual-memory map was invalid and had to be reloaded from the page hash table.

sys:%count-second-level-map-reloads *Meter*
The number of times the second-level virtual-memory map was invalid and had to be reloaded from the page hash table.

sys:%count-meta-bits-map-reloads *Meter*
The number of times the virtual address map was reloaded to contain only "meta bits", not an actual physical address.

sys:%count-pdl-buffer-read-faults *Meter*
The number of read references to the pdl buffer which were virtual memory references that trapped.

sys:%count-pdl-buffer-write-faults *Meter*
The number of write references to the pdl buffer which were virtual memory references that trapped.

sys:%count-pdl-buffer-memory-faults *Meter*
The number of virtual memory references which trapped in case they should have gone to the pdl buffer, but turned out to be real memory references after all (and therefore were needlessly slowed down.)

sys:%count-disk-page-reads *Meter*
The number of pages read from the disk.

sys:%count-disk-page-writes *Meter*
The number of pages written to the disk.

sys:%count-fresh-pages *Meter*
The number of fresh (newly-consed) pages created in core, which would have otherwise been read from the disk.

sys:%count-disk-page-read-operations *Meter*
The number of paging read operations; this can be smaller than the number of disk pages read when more than one page at a time is read.

sys:%count-disk-page-write-operations *Meter*
The number of paging write operations; this can be smaller than the number of disk pages written when more than one page at a time is written.

sys:%count-disk-prepages-used *Meter*
The number of times a page was used after being read in before it was needed.

sys:%count-disk-prepages-not-used *Meter*
The number of times a page was read in before it was needed, but got evicted before it was ever used.

sys:%count-disk-page-write-waits *Meter*
The number of times the machine waited for a page to finish being written out in order to evict the page.

sys:%count-disk-page-write-busys *Meter*
The number of times the machine waited for a page to finish being written out in order to do something else with the disk.

sys:%disk-wait-time *Meter*
The time spent waiting for the disk, in microseconds. This can be used to distinguish paging time from running time when measuring and optimizing the' performance of programs.

sys:%count-disk-errors *Meter*
The number of recoverable disk errors.

sys:%count-disk-recalibrates *Meter*
The number of times the disk seek mechanism was recalibrated, usually as part of error recovery.

sys:%count-disk-ecc-corrected-errors *Meter*
The number of disk errors which were corrected through the error correcting code.

sys:%count-disk-read-compare-differences *Meter*
The number of times a read compare was done, no disk error occurred, but the data on disk did not match the data in memory.

sys:%count-disk-read-compare-rereads *Meter*
The number of times a disk read was done over because after the read a read compare was done and did not succeed (either it got an error or the data on disk did not match the data in memory).

sys:%count-disk-read-compare-rewrites *Meter*
The number of times a disk write was done over because after the write a read compare was done and did not succeed (either it got an error or the data on disk did not match the data in memory).

sys:%disk-error-log-pointer *Meter*
Address of the next entry to be written in the disk error log. The function **si:print-disk-error-log** (see page 503) prints this log.

sys:%count-aged-pages *Meter*
The number of times the page ager set an age trap on a page, to determine whether it was being referenced.

sys:%count-age-flushed-pages *Meter*
The number of times the page ager saw that a page still had an age trap and hence made it "flushable", a candidate for eviction from main memory.

sys:%aging-depth *Meter*
A number from 0 to 3 which controls how long a page must remain unreferenced before it becomes a candidate for eviction from main memory.

sys:%count-findcore-steps *Meter*
The number of pages inspected by the page replacement algorithm.

sys:%count-findcore-emergencies *Meter*
The number of times no evictable page was found and extra aging had to be done.

sys:a-memory-counter-block-names *Variable*
A list of all of the above symbols (and any others added after this documentation was written).

15. Areas

Storage in the Lisp Machine is divided into *areas*. Each area contains related objects, of any type. Areas are intended to give the user control over the paging behavior of his program, among other things. By putting related data together, locality can be greatly increased. Whenever a new object is created the area to be used can optionally be specified. For example, instead of using cons you can use cons-in-area (see page 54). Object-creating functions which take keyword arguments generally accept a :area argument. You can also control which area is used by binding default-cons-area (see page 193); most functions that allocate storage use the value of this variable, by default, to specify the area to use.

There is a default Working Storage area which collects those objects which the user has not chosen to control explicitly.

Areas also give the user a handle to control the garbage collector. Some areas can be declared to be "static", which means that they change slowly and the garbage collector should not attempt to reclaim any space in them. This can eliminate a lot of useless copying. A "static" area can be explicitly garbage-collected at infrequent intervals when it is believed that that might be worthwhile.

Each area can potentially have a different storage discipline, a different paging algorithm, and even a different data representation. The microcode will dispatch on an attribute of the area at the appropriate times. The structure of the machine makes the performance cost of these features negligible; information about areas is stored in extra bits in the memory mapping hardware where it can be quickly dispatched on by the microcode; these dispatches usually have to be done anyway to make the garbage collector work, and to implement invisible pointers. This feature is not currently used by the system, except for the list/structure distinction described below.

Each area has a name and a number. The name is a symbol whose value is the number. The number is an index into various internal tables. Normally the name is treated as a special variable, so the number is what is given as an argument to a function that takes an area as an argument. Thus, areas are not Lisp objects; you cannot pass an area itself as an argument to a function; you just pass its number. There is a maximum number of areas (set at cold-load generation time); you can only have that many areas before the various internal tables overflow. Currently (as this manual is written) the limit is 256. areas, of which 64. already exist when you start.

The storage of an area consists of one or more *regions*. Each region is a contiguous section of address space with certain homogeneous properties. The most important of these is the *data representation type*. A given region can only store one type. The two types that exist now are *list* and *structure*. A list is anything made out of conses (a closure for instance). A structure is anything made out of a block of memory with a header at the front; symbols, strings, arrays, instances, compiled functions, etc. Since lists and structures cannot be stored in the same region, they cannot be on the same page. It is necessary to know about this when using areas to increase locality of reference.

When you create an area, one region is created initially. When you try to allocate memory to hold an object in some area, the system tries to find a region that has the right data representation type to hold this object, and that has enough room for it to fit. If there isn't any such region, it makes a new one (or signals an error; see the :size option to make-area, below). The size of the new region is an attribute of the area (controllable by the :region-size option to make-area). If regions are too large, memory may get taken up by a region and never used. If regions are too small, the system may run out of regions because regions, like areas, are defined by internal tables that have a fixed size (set at cold-load generation time). Currently (as this manual is written) the limit is 256. regions, of which about 90. already exist when you start. (If you're wondering why the limit on regions isn't higher than the limit on areas, as it clearly ought to be, it's just because both limits have to be multiples of 256. for internal reasons, and 256. regions seem to be enough.)

15.1 Area Functions and Variables

default-cons-area *Variable*

The value of this variable is the number of the area in which objects are created by default. It is initially the number of working-storage-area. Giving nil where an area is required uses the value of default-cons-area. Note that to put objects into an area other than working-storage-area you can either bind this variable or use functions such as cons-in-area (see page 54) which take the area as an explicit argument.

make-area &rest *keywords*

Creates a new area, whose name and attributes are specified by the keywords. You must specify a symbol as a name; the symbol will be setq'ed to the area-number of the new area, and that number will also be returned, so that you can use make-area as the initialization of a defvar. The arguments are taken in pairs, the first being a keyword and the second a "value" for that keyword. The last three keywords documented herein are in the nature of subprimitives; like the stuff in chapter 14, their meaning is system-dependent and is not documented here. The following keywords exist:

:name	A symbol which will be the name of the area. This item is required.
:size	The maximum allowed size of the area, in words. Defaults to infinite. If the number of words allocated to the area reaches this size, attempting to cons an object in the area will signal an error.
:region-size	The approximate size, in words, for regions within this area. The default is the area size if a :size argument was given, otherwise a suitable medium size. Note that if you specify :size and not :region-size, the area will have exactly one region. When making an area which will be very big, it is desirable to make the region size larger than the default region size to avoid creating very many regions and possibly overflowing the system's fixed-size region tables.
:representation	The type of object to be contained in the area's initial region. The argument to this keyword can be :list, :structure, or a numeric code. :structure is the default. If you are only going to cons lists in your area, you should specify :list so you don't get a useless structure region.

:gc
: The type of garbage-collection to be employed. The choices are :dynamic (which is the default) and :static. :static means that the area will not be copied by the garbage collector, and nothing in the area or pointed to by the area will ever be reclaimed, unless a garbage collection of this area is manually requested.

:read-only
: With an argument of t, causes the area to be made read-only. Defaults to nil. If an area is read-only, then any attempt to change anything in it (altering a data object in the area, or creating a new object in the area) will signal an error unless sys:%inhibit-read-only (see page 187) is bound to a non-nil value.

:pdl
: With an argument of t, makes the area suitable for storing regular-pdls of stack-groups. This is a special attribute due to the pdl-buffer hardware. Defaults to nil. Areas for which this is nil may *not* be used to store regular-pdls. Areas for which this is t are relatively slow to access; all references to pages in the area will take page faults to check whether the referenced location is really in the pdl-buffer.

sys:%%region-map-bits
: Lets you specify the *map bits* explicitly, overriding the specification from the other keywords. This is for special hacks only.

sys:%%region-space-type
: Lets you specify the *space type* explicitly, overriding the specification from the other keywords. This is for special hacks only.

sys:%%region-scavenge-enable
: Lets you override the scavenge-enable bit explicitly. This is an internal flag related to the garbage collector. Don't mess with this!

:room
: With an argument of t, adds this area to the list of areas which are displayed by default by the room function (see page 500).

Example:
```
        (make-area ':name 'foo-area
                   ':gc ':dynamic
                   ':representation ':list)
```

describe-area *area*
area may be the name or the number of an area. Various attributes of the area are printed.

area-list *Variable*
The value of area-list is a list of the names of all existing areas. This list shares storage with the internal area name table, so you should not change it.

%area-number *pointer*

Returns the number of the area to which *pointer* points, or nil if it does not point within any known area. The data-type of *pointer* is ignored.

%region-number *pointer*

Returns the number of the region to which *pointer* points, or nil if it does not point within any known region. The data-type of *pointer* is ignored. (This information is generally not very interesting to users; it is important only inside the system.)

area-name *number*

Given an area number, returns the name. This "function" is actually an array.

See also cons-in-area (page 54), list-in-area (page 57), and room (page 500).

15.2 Interesting Areas

This section lists the names of some of the areas and tells what they are for. Only the ones of the most interest to a user are listed; there are many others.

working-storage-area *Variable*

This is the normal value of default-cons-area. Most working data are consed in this area.

permanent-storage-area *Variable*

This area is to be used for "permanent" data, which will (almost) never become garbage. Unlike working-storage-area, the contents of this area are not continually copied by the garbage collector; it is a static area.

sys:p-n-string *Variable*

Print-names of symbols are stored in this area.

sys:nr-sym *Variable*

This area contains most of the symbols in the Lisp world, except t and nil, which are in a different place for historical reasons.

sys:pkg-area *Variable*

This area contains packages, principally the hash tables with which intern keeps track of symbols.

macro-compiled-program *Variable*

FEFs (compiled functions) are put here by the compiler and by fasload.

sys:property-list-area *Variable*

This area holds the property lists of symbols.

sys:init-list-area *Variable*
sys:fasl-constants-area *Variable*
These two areas contain constants used by compiled programs.

16. The Compiler

16.1 The Basic Operations of the Compiler

The purpose of the Lisp compiler is to convert Lisp functions into programs in the Lisp Machine's instruction set, so that they will run more quickly and take up less storage. Compiled functions are represented in Lisp by FEFs (Function Entry Frames), which contain machine code as well as various other information. The printed representation of a FEF is

`#<DTP-FEF-POINTER` *address name*`>`

If you want to understand the output of the compiler, refer to chapter 27, page 469.

There are three ways to invoke the compiler from the Lisp Machine. First, you may have an interpreted function in the Lisp environment which you would like to compile. The function **compile** is used to do this. Second, you may have code in an editor buffer which you would like to compile. The Zwei editor has commands to read code into Lisp and compile it. Third, you may have a program (a group of function definitions and other forms) written in a file on the file system. The compiler can translate this file into a QFASL file. Loading in the QFASL file is almost the same as reading in the source file; the difference is that the functions defined in the file will be defined as compiled functions instead of interpreted functions. The **qc-file** function is used for translating source files into QFASL files.

16.2 How to Invoke the Compiler

compile *function-spec* &optional *definition*

If *definition* is supplied, it should be a **lambda**-expression. Otherwise *function-spec* (this is usually a symbol, but see section 10.2, page 136 for details) should be defined as an interpreted function and its definition will be used as the **lambda**-expression to be compiled. The compiler converts the **lambda**-expression into a FEF, saves the **lambda**-expression as the **:previous-expr-definition** and **:previous-definition** properties of *function-spec* if it is a symbol, and changes *function-spec*'s definition to be the FEF. (See **fdefine**, page 149). (Actually, if *function-spec* is not defined as a **lambda**-expression, and *function-spec* is a symbol, **compile** will try to find a **lambda**-expression in the **:previous-expr-definition** property of *function-spec* and use that instead.)

uncompile *symbol*

If *symbol* is not defined as an interpreted function and it has a **:previous-expr-definition** property, then **uncompile** will restore the function cell from the value of the property. (Otherwise, **uncompile** does nothing and returns "Not compiled".) This "undoes" the effect of **compile**. See also **undefun**, page 150.

qc-file *filename* &optional *output-file load-flag in-core-flag package functions-defined file-local-declarations dont-set-default-p read-then-process-flag*

This function takes a formidable number of arguments, but normally only one argument is supplied. The file *filename* is given to the compiler, and the output of the compiler is written to a file whose name is *filename* except with a file type of "QFASL". The input format for files to the compiler is described on section 16.3, page 198. Macro definitions, subst definitions, and special declarations created during the compilation are undone when the compilation is finished.

The optional arguments allow certain modifications to the standard procedure. *output-file* lets you change where the output is written. *package* lets you specify in what package the source file is to be read. Normally the system knows, or asks interactively, and you need not supply this argument. *load-flag* and *in-core-flag* are incomprehensible; you don't want to use them. *functions-defined* and *file-local-declarations* are for compiling multiple files as if they were one. *dont-set-default-p* suppresses the changing of the default file name to *filename* that normally occurs.

Normally, a form is read from the file and processed and then another form is read and processed, and so on. But if *read-then-process-flag* is non-nil, the whole source file is read before any of it is processed. This is not done by default; it has the problem that compile-time reader-macros defined in the file will not work properly.

qc-file-load *filename* &optional *output-file load-flag in-core-flag package functions-defined file-local-declarations dont-set-default-p read-then-process-flag*

qc-file-load compiles a file and then loads in the resulting QFASL file.

See also the disassemble function (page 500), which lists the instructions of a compiled function in symbolic form.

16.3 Input to the Compiler

The purpose of qc-file is to take a file and produce a translated version which does the same thing as the original except that the functions are compiled. qc-file reads through the input file, processing the forms in it one by one. For each form, suitable binary output is sent to the QFASL file so that when the QFASL file is loaded the effect of that source form will be reproduced. The differences between source files and QFASL files are that QFASL files are in a compressed binary form which reads much faster (but cannot be edited), and that function definitions in QFASL files have been translated from Lisp forms to FEFs.

So, if the source contains a (defun ...) form at top level, then when the QFASL file is loaded, the function will be defined as a compiled function. If the source file contains a form which is not of a type known specially to the compiler, then that form (encoded in QFASL format) will be output "directly" into the QFASL file, so that when the QFASL file is loaded that form will be evaluated. Thus, if the source file contains (setq x 3), then the compiler will put in the QFASL file instructions to set x to 3 at load time (that is, when the QFASL file is loaded into the Lisp environment). It happens that QFASL files have a specific way to setq a symbol. For a more general form, the QFASL file would contain instructions to recreate the list structure of a form and then call eval on it.

Sometimes we want to put things in the file that are not merely meant to be translated into QFASL form. One such occasion is top level macro definitions; the macros must actually get defined within the compiler in order for the compiler to be able to expand them at compile time. So when a macro form is seen, it should (sometimes) be evaluated at compile time, and should (sometimes) be put into the QFASL file.

Another thing we sometimes want to put in a file is compiler declarations. These are forms which should be evaluated at compile time to tell the compiler something. They should not be put into the QFASL file, unless they are useful for working incrementally on the functions in the file, compiling them one by one from the editor.

Therefore, a facility exists to allow the user to tell the compiler just what to do with a form. One might want a form to be:

Put into the QFASL file (compiled, of course), or not.

Evaluated within the compiler, or not.

Evaluated if the file is read directly into Lisp, or not.

Two forms are recognized by the compiler to allow this. The less general, old-fashioned one is **declare**; the completely general one is **eval-when**.

An **eval-when** form looks like

```
(eval-when times-list
           form1
           form2
           ...)
```

The *times-list* may contain one or more of the symbols **load**, **compile**, or **eval**. If **load** is present, the *forms* are written into the QFASL file to be evaluated when the QFASL file is loaded (except that **defun** forms will put the compiled definition into the QFASL file instead). If **compile** is present, the *forms* are evaluated in the compiler. If **eval** is present, the *forms* are evaluated when read into Lisp; this is because **eval-when** is defined as a special form in Lisp. (The compiler ignores **eval** in the *times-list*.) For example,

```
(eval-when (compile eval) (macro foo (x) (cadr x)))
```

would define **foo** as a macro in the compiler and when the file is read in interpreted, but not when the QFASL file is fasloaded.

For the rest of this section, we will use lists such as are given to **eval-when**, e.g. **(load eval)**, **(load compile)**, etc. to describe when forms are evaluated.

A **declare** form looks like **(declare** *form1 form2* ...**)**. **declare** is defined in Lisp as a special form which does nothing; so the forms within a **declare** are not evaluated at **eval** time. The compiler does the following upon finding *form* within a **declare**: if *form* is a call to either **special** or **unspecial**, *form* is treated as **(load compile)**; otherwise it is treated as **(compile)**.

If a form is not enclosed in an **eval-when** nor a **declare**, then the times at which it will be evaluated depend on the form. The following table summarizes at what times evaluation will take place for any given form seen at top level by the compiler.

(eval-when *times-list form1* ...**)**
times-list

(declare (special ...)) or **(declare (unspecial ...))**
(load compile)

(declare *anything-else*)
: (compile)

(special ...) or (unspecial ...)
: (load compile eval)

(macro ...) or (defmacro ...) or (defsubst ...)
: (load compile eval)

(comment ...) Ignored at all times.

(compiler-let ((*var val*) ...) *body*...)
: Processes the *body* in its normal fashion, but at (compile eval) time, the indicated variable bindings are in effect. These variables will typically affect the operation of the compiler or of macros. See section 17.4.6, page 223.

(local-declare (*decl decl*...) *body*...)
: Processes the *body* in its normal fashion, with the indicated declarations added to the front of the list which is the value of local-declarations.

(defflavor ...) or (defstruct ...)
: (load compile eval)

(defun ...) or (defmethod ...) or (defselect ...)
: (load eval), but at load time what is processed is not this form itself, but the result of compiling it.

anything-else (load eval)

Sometimes a macro wants to return more than one form for the compiler top level to see (and to be evaluated). The following facility is provided for such macros. If a form

```
    (progn (quote compile) form1 form2 ...)
```

is seen at the compiler top level, all of the *forms* are processed as if they had been at compiler top level. (Of course, in the interpreter they will all be evaluated, and the (quote compile) will harmlessly evaluate to the symbol compile and be ignored.) See section 17.4.3, page 219 for additional discussion of this.

eval-when (*time*...) *body*... *Special Form*

When seen by the interpreter, if one of the *times* is the symbol eval then the *body* forms are evaluated; otherwise eval-when does nothing.

But when seen by the compiler, this special form does the special things described above.

declare *declaration*... *Special Form*

declare does nothing, and returns the symbol declare.

But when seen by the compiler, this special form does the special things described above.

16.4 Compiler Declarations

This section describes functions meant to be called during compilation, and variables meant to be set or bound during compilation, by using declare or local-declare.

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

A local-declare form looks like

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

Each *decl* is consed onto the list local-declarations while the *forms* are being evaluated (in the interpreter) or compiled (in the compiler). There are two uses for this. First, it can be used to pass information from outer macros to inner macros. Secondly, the compiler will specially interpret certain *decls* as local declarations, which only apply to the compilations of the *forms*. It understands the following forms:

(special *var1 var2*...)
: The variables *var1*, *var2*, etc. will be treated as special variables during the compilation of the *forms*.

(unspecial *var1 var2*...)
: The variables *var1*, *var2*, etc. will be treated as local variables during the compilation of the *forms*.

(arglist . *arglist*)
: Putting this local declaration around a defun saves *arglist* as the argument list of the function, to be used instead of its lambda-list if anyone asks what its arguments are. This is purely documentation.

(return-list . *values*)
: Putting this local declaration around a defun saves *values* as the return values list of the function, to be used if anyone asks what values it returns. This is purely documentation.

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

special *variable...* *Special Form*

Declares each *variable* to be "special" for the compiler.

unspecial *variable...* *Special Form*

Removes any "special" declarations of the *variables* for the compiler.

The next three declarations are primarily for Maclisp compatibility.

***expr** *symbol...* *Special Form*

Declares each *symbol* to be the name of a function. In addition it prevents these functions from appearing in the list of functions referenced but not defined printed at the end of the compilation.

***lexpr** *symbol...* *Special Form*

Declares each *symbol* to be the name of a function. In addition it prevents these functions from appearing in the list of functions referenced but not defined printed at the end of the compilation.

***fexpr** *symbol...* *Special Form*

Declares each *symbol* to be the name of a special form. In addition it prevents these names from appearing in the list of functions referenced but not defined printed at the end of the compilation.

There are some advertised variables whose compile-time values affect the operation of the compiler. The user may set these variables by including in his file forms such as

```
(declare (setq open-code-map-switch t))
```

run-in-maclisp-switch *Variable*

If this variable is non-nil, the compiler will try to warn the user about any constructs which will not work in Maclisp. By no means will all Lisp Machine system functions not built in to Maclisp be cause for warnings; only those which could not be written by the user in Maclisp (for example, make-array, value-cell-location, etc.). Also, lambda-list keywords such as &optional and initialized prog variables will be mentioned. This switch also inhibits the warnings for obsolete Maclisp functions. The default value of this variable is nil.

obsolete-function-warning-switch *Variable*

If this variable is non-nil, the compiler will try to warn the user whenever an "obsolete" Maclisp-compatibility function such as maknam or samepnamep is used. The default value is t.

allow-variables-in-function-position-switch *Variable*

If this variable is non-nil, the compiler allows the use of the name of a variable in function position to mean that the variable's value should be funcall'd. This is for compatibility with old Maclisp programs. The default value of this variable is nil.

open-code-map-switch *Variable*

If this variable is non-nil, the compiler will attempt to produce inline code for the mapping functions (mapc, mapcar, etc., but not mapatoms) if the function being mapped is an anonymous lambda-expression. This allows that function to reference the local variables of the enclosing function without the need for special declarations. The generated code is also more efficient. The default value is t.

all-special-switch *Variable*

If this variable is non-nil, the compiler regards all variables as special, regardless of how they were declared. This provides compatibility with the interpreter at the cost of efficiency. The default is nil.

inhibit-style-warnings-switch *Variable*

If this variable is non-nil, all compiler style-checking is turned off. Style checking is used to issue obsolete function warnings and won't-run-in-Maclisp warnings, and other sorts of warnings. The default value is nil. See also the inhibit-style-warnings macro, which acts on one level only of an expression.

compiler-let ((*variable value*)...) *body*... *Macro*

Syntactically identical to let, compiler-let allows compiler switches to be bound locally at compile time, during the processing of the *body* forms.
Example:

```
        (compiler-let ((open-code-map-switch nil))
                  (map (function (lambda (x) ...)) foo))
```

will prevent the compiler from open-coding the map. When interpreted, compiler-let is equivalent to let. This is so that global switches which affect the behavior of macro expanders can be bound locally.

16.5 Controlling Compiler Warnings

By controlling the compile-time values of the variables run-in-maclisp-switch, obsolete-function-warning-switch, and inhibit-style-warning-switch (explained above), you can enable or disable some of the warning messages of the compiler. The following special form is also useful:

inhibit-style-warnings *form* *Macro*

Prevents the compiler from performing style-checking on the top level of *form*. Style-checking will still be done on the arguments of *form*. Both obsolete function warnings and won't-run-in-Maclisp warnings are done by means of the style-checking mechanism, so, for example,

```
        (setq bar (inhibit-style-warnings (value-cell-location foo)))
```

will not warn that value-cell-location will not work in Maclisp, but

```
        (inhibit-style-warnings (setq bar (value-cell-location foo)))
```

will warn, since inhibit-style-warnings applies only to the top level of the form inside it (in this case, to the setq).

Sometimes functions take argument that they deliberately do not use. Normally the compiler warns you if your program binds a variable that it never references. In order to disable this warning for variables that you know you are not going to use, there are two things you can do. The first thing is to name the variables ignore or ignored. The compiler will not complain if a variable by one of these names is not used. Furthermore, by special dispensation, it is all right to have more than one variable in a lambda-list that has one of these names. The other thing you can do is simply use the variable, for effect (ignoring its value), at the front of the function. Example:

```
(defun the-function (list fraz-name fraz-size)
  fraz-size      ; This argument is not used.
  ...)
```

This has the advantage that arglist (see page 150) will return a more meaningful argument list for the function, rather than returning something with ignores in it.

The following function is useful for requesting compiler warnings in certain esoteric cases. Normally, the compiler notices whenever any function *x* uses (calls) any other function *y*; it makes notes of all these uses, and then warns you at the end of the compilation if the function *y* got called but was neither defined nor declared (by *expr, see page 202). This usually does what you want, but sometimes there is no way the compiler can tell that a certain function is being used. Suppose that instead of *x*'s containing any forms that call *y*, *x* simply stores *y* away in a data structure somewhere, and someplace else in the program that data structure is accessed and funcall is done on it. There is no way that the compiler can see that this is going to happen, and so it can't notice the function usage, and so it can't create a warning message. In order to make such warnings happen, you can explicitly call the following function at compile-time.

compiler:function-referenced *what by*

what is a symbol that is being used as a function. *by* may be any function spec. compiler:function-referenced must be called at compile-time while a compilation is in progress. It tells the compiler that the function *what* is referenced by *by*. When the compilation is finished, if the function *what* has not been defined, the compiler will issue a warning to the effect that *by* referred to the function *what*, which was never defined.

compiler:make-obsolete *function reason* *Special Form*

This special form declares a function to be obsolete; code that calls it will get a compiler warning, under the control of obsolete-function-warning-switch. This is used by the compiler to mark as obsolete some Maclisp functions which exist in Zetalisp but should not be used in new programs. It can also be useful when maintaining a large system, as a reminder that a function has become obsolete and usage of it should be phased out. An example of an obsolete-function declaration is:

```
(compiler:make-obsolete create-mumblefrotz
        "use MUMBLIFY with the :FROTZ option instead")
```

16.6 Compiler Source-Level Optimizers

The compiler stores optimizers for source code on property lists so as to make it easy for the user to add them. An optimizer can be used to transform code into an equivalent but more efficient form (for example, (eq *obj* nil) is transformed into (null *obj*), which can be compiled better). An optimizer can also be used to tell the compiler how to compile a special form. For example, in the interpreter do is a special form, implemented by a function which takes quoted arguments and calls eval. In the compiler, do is expanded in a macro-like way by an optimizer, into equivalent Lisp code using prog, cond, and go, which the compiler understands.

The compiler finds the optimizers to apply to a form by looking for the compiler:optimizers property of the symbol which is the car of the form. The value of this property should be a list of optimizers, each of which must be a function of one argument. The compiler tries each optimizer in turn, passing the form to be optimized as the argument. An optimizer which returns the original form unchanged (eq to the argument) has "done nothing", and the next optimizer is tried. If the optimizer returns anything else, it has "done something", and the whole process starts over again. Only after all the optimizers have been tried and have done nothing is an ordinary macro definition processed. This is so that the macro definitions, which will be seen by the interpreter, can be overridden for the compiler by optimizers.

Optimizers should not be used to define new language features, because they only take effect in the compiler; the interpreter (that is, the evaluator) doesn't know about optimizers. So an optimizer should not change the effect of a form; it should produce another form that does the same thing, possibly faster or with less memory or something. That is why they are called optimizers. If you want to actually change the form to do something else, you should be using macros.

compiler:add-optimizer *function optimizer* *Special Form*

Puts *optimizer* on *function*'s optimizers list if it isn't there already. *optimizer* is the name of an optimization function, and *function* is the name of the function calls which are to be processed. Neither is evaluated.

(compiler:add-optimizer *function optimizer optimize-into-1 optimize-into-2...*) also remembers *optimize-into-1*, etc., as names of functions which may be called in place of *function* as a result of the optimization.

16.7 Files that Maclisp Must Compile

Certain programs are intended to be run both in Maclisp and in Zetalisp. Their source files need some special conventions. For example, all special declarations must be enclosed in declares, so that the Maclisp compiler will see them. The main issue is that many functions and special forms of Zetalisp do not exist in Maclisp. It is suggested that you turn on run-in-maclisp-switch in such files, which will warn you about a lot of problems that your program may have if you try to run it in Maclisp.

The macro-character combination "#Q" causes the object that follows it to be visible only when compiling for Zetalisp. The combination "#M" causes the following object to be visible only when compiling for Maclisp. These work both on subexpressions of the objects in the file, and at top level in the file. To conditionalize top-level objects, however, it is better to put the macros if-for-lispm and if-for-maclisp around them. (You can only put these around a single object.) The if-for-lispm macro turns off run-in-maclisp-switch within its object, preventing spurious warnings from the compiler. The #Q macro-character cannot do this, since it can be used to conditionalize any S-expression, not just a top-level form.

To allow a file to detect what environment it is being compiled in, the following macros are provided:

if-for-lispm *form* *Macro*

If (if-for-lispm *form*) is seen at the top level of the compiler, *form* is passed to the compiler top level if the output of the compiler is a QFASL file intended for Zetalisp. If the Zetalisp interpreter sees this it will evaluate *form* (the macro expands into *form*).

if-for-maclisp *form* *Macro*

If (if-for-maclisp *form*) is seen at the top level of the compiler, *form* is passed to the compiler top level if the output of the compiler is a FASL file intended for Maclisp (e.g. if the compiler is COMPLR). If the Zetalisp interpreter sees this it will ignore it (the macro expands into nil).

if-for-maclisp-else-lispm *maclisp-form lispm-form* *Macro*

If (if-for-maclisp-else-lispm *form1 form2*) is seen at the top level of the compiler, *form1* is passed to the compiler top level if the output of the compiler is a FASL file intended for Maclisp; otherwise *form2* is passed to the compiler top level.

if-in-lispm *form* *Macro*

In Zetalisp, (if-in-lispm *form*) causes *form* to be evaluated; in Maclisp, *form* is ignored.

if-in-maclisp *form* *Macro*

In Maclisp, (if-in-maclisp *form*) causes *form* to be evaluated; in Zetalisp, *form* is ignored.

When you have two definitions of one function, one conditionalized for one machine and one for the other, put them next to each other in the source file with the second "(defun" indented by one space, and the editor will put both function definitions on the screen when you ask to edit that function.

In order to make sure that those macros are defined when reading the file into the Maclisp compiler, you must make the file start with a prelude, which should look like:

```
(declare (cond ((not (status feature lispm))
                (load '|AI: LISPM2; CONDIT|))))
```

This will do nothing when you compile the program on the Lisp Machine. If you compile it with the Maclisp compiler, it will load in definitions of the above macros, so that they will be available to your program. The form (status feature lispm) is generally useful in other ways; it evaluates to t when evaluated on the Lisp machine and to nil when evaluated in Maclisp.

16.8 Putting Data in QFASL Files

It is possible to make a QFASL file containing data, rather than a compiled program. This can be useful to speed up loading of a data structure into the machine, as compared with reading in printed representations. Also, certain data structures such as arrays do not have a convenient printed representation as text, but can be saved in QFASL files. For example, the system stores fonts this way. Each font is in a QFASL file (on the LMFONT directory) which contains the data structures for that font. When the file is loaded, the symbol which is the name of the font gets set to the array which represents the font. Putting data into a QFASL file is often referred to as "*fasdumping* the data".

In compiled programs, the constants are saved in the QFASL file in this way. The compiler optimizes by making constants which are equal become eq when the file is loaded. This does not happen when you make a data file yourself; identity of objects is preserved. Note that when a QFASL file is loaded, objects that were eq when the file was written are still eq; this does not normally happen with text files.

The following types of objects can be represented in QFASL files: Symbols (but uninterned symbols will be interned when the file is loaded), numbers of all kinds, lists, strings, arrays of all kinds, instances, and FEFs.

When an instance is fasdumped (put into a QFASL file), it is sent a :fasd-form message, which must return a Lisp form which, when evaluated, will recreate the equivalent of that instance. This is because instances are often part of a large data structure, and simply fasdumping all of the instance variables and making a new instance with those same values is unlikely to work. Instances remain eq; the :fasd-form message is only sent the first time a particular instance is encountered during writing of a QFASL file. If the instance does not accept the :fasd-form message, it cannot be fasdumped.

compiler:fasd-symbol-value *filename symbol*

Writes a QFASL file named *filename* which contains the value of *symbol*. When the file is loaded, *symbol* will be setq'ed to the same value. *filename* is parsed with the same defaults that load and qc-file use. The file type defaults to "qfasl".

compiler:fasd-font *name*

Writes the font named *name* into a QFASL file with the appropriate name (on the LMFONT directory).

compiler:fasd-file-symbols-properties *filename symbols properties dump-values-p dump-functions-p new-symbol-function*

This is a way to dump a complex data structure into a QFASL file. The values, the function definitions, and some of the properties of certain symbols are put into the QFASL file in such a way that when the file is loaded the symbols will be setqed, fdefined, and putproped appropriately. The user can control what happens to symbols discovered in the data structures being fasdumped.

filename is the name of the file to be written. It is parsed with the same defaults that load and qc-file use. The file type defaults to "qfasl".

symbols is a list of symbols to be processed. *properties* is a list of properties which are to be fasdumped if they are found on the symbols. *dump-values-p* and *dump-functions-p* control whether the values and function definitions are also dumped.

new-symbol-function is called whenever a new symbol is found in the structure being dumped. It can do nothing, or it can add the symbol to the list to be processed by calling compiler:fasd-symbol-push. The value returned by *new-symbol-function* is ignored.

17. Macros

17.1 Introduction to Macros

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

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

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

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

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

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

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

macro *Special Form*

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

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

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

To define our first macro, we would say

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

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

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

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

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

Of course, this macro is of limited usefulness. The reason is that the form in the *cadr* of the increment form had better be a symbol. If you tried (increment (car x)), it would be translated into (setq (car x) (1+ (car x))), and setq would complain. (If you're interested in how to fix this problem, see setf (page 229); but this is irrelevant to how macros work.)

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

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

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

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

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

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

A macro to do this could be defined with

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

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

17.2 Aids for Defining Macros

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

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

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

17.2.1 Defmacro

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

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

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

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

defmacro *Macro*

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

(defmacro *name pattern . body*)

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

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

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

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

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

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

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

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

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

There is another version of defmacro which defines displacing macros (see section 17.6, page 226). defmacro has other, more complex features; see section 17.7, page 227.

17.2.2 Backquote

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

into

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

You could define the macro by

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

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

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

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

since backquote might choose to implement this as

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

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

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

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

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

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

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

This example is a very simple version of defstruct (see page 259). You should first understand the basic description of defstruct before proceeding with this example. The defstruct below does not accept any options, and only allows the simplest kind of items; that is, it only allows forms like

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

We would like this form to expand into

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

Here is the macro to perform the expansion:

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

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

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

Instead of using this backquote construction, we could have written

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

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

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

17.3 Substitutable Functions

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

defsubst *Special Form*

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

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

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

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

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

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

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

square's definition would be

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

(The internal formats of substs and named-substs are explained in section 10.5.1, page 143.)

A similar square could be defined as a macro, with

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

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

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

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

would present problems. When compiled, because of the substitution a call to reverse-cons would evaluate its arguments in the wrong order, and a call to in-order could evaluate its second argument twice. This will be fixed at some point in the future, but for now the writer of defsubst's must be cautious. Also all occurrences of the argument names in the body are replaced with the argument forms, wherever they appear. Thus an argument name should not be used in the body for anything else, such as a function name or a symbol in a constant.

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

17.4 Hints to Macro Writers

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

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

In general any macro that can be written as a substitutable function (see page 215) should be written as one, not as a macro, for several reasons: substitutable functions are easier to write and to read; they can be passed as functional arguments (for example, you can pass them to mapcar); and there are some subtleties that can occur in macro definitions that need not be worried about in substitutable functions. A macro can be a substitutable function only if it has exactly the semantics of a function, rather than a special form. The macros we will see in this section are not semantically like functions; they must be written as macros.

17.4.1 Name Conflicts

One of the most common errors in writing macros is best illustrated by example. Suppose we wanted to write dolist (see page 42) as a macro that expanded into a do (see page 38). The first step, as always, is to figure out what the expansion should look like. Let's pick a representative example form, and figure out what its expansion should be. Here is a typical dolist form.

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

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

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

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

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

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

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

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

Here's another example that causes trouble:

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

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

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

One way to avoid this problem is to choose a name that is very unlikely to be picked by the user, simply by choosing an unusual name. This will probably work, but it is inelegant since there is no guarantee that the user won't just happen to choose the same name. The way to really avoid the name conflict is to use an uninterned symbol as the variable in the generated code. The function gensym (see page 91) is useful for creating such symbols.

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

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

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

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

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

17.4.2 prog-context Conflicts

A related problem occurs when you write a macro that expands into a prog (or a do, or something that expands into prog or do) behind the user's back (unlike dolist, which is documented to be like do). Consider the error-restart special form (see page 446); suppose we wanted to implement it as a macro that expands into a prog. If it expanded into a plain-old prog, then the following (contrived) Lisp program would not behave correctly:

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

The problem is that the return would return from the error-restart instead of the prog. The way to avoid this problem is to use a named prog whose name is t. The name t is special in that it is invisible to the return function. If we write error-restart as a macro that expands into a prog named t, then the return will pass right through the error-restart form and return from the prog, as it ought to.

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

17.4.3 Macros Expanding into Many Forms

Sometimes a macro wants to do several different things when its expansion is evaluated. Another way to say this is that sometimes a macro wants to expand into several things, all of which should happen sequentially at run time (not macro-expand time). For example, suppose you wanted to implement defconst (see page 19) as a macro. defconst must do two things: declare the variable to be special, and set the variable to its initial value. (We will implement a simplified defconst that only does these two things, and doesn't have any options.) What should a defconst form expand into? Well, what we would like is for an appearance of

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

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

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

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

There is such a form. It looks like this:

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

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

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

Here is the macro definition:

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

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

names of all commands are put onto a list. A typical call to define-command would look like:
```
(define-command move-to-top
   "This command moves you to the top."
   (do ()
       ((at-the-top-p))
     (move-up-one)))
```
This could expand into:
```
(progn 'compile
       (defprop
         move-to-top
         "This command moves you to the top."
         help)
       (push 'move-to-top *command-name-list*)
       (defun move-to-top ()
         (do ()
             ((at-the-top-p))
           (move-up-one)))
       )
```
The define-command expands into three forms. The first one sets up the documentation string and the second one puts the command name onto the list of all command names. The third one is the defun that actually defines the function itself. Note that the defprop and push happen at load-time (when the file is loaded); the function, of course, also gets defined at load time. (See the description of eval-when (page 200) for more discussion of the differences between compile time, load time, and eval time.)

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

17.4.4 Macros that Surround Code

There is a particular kind of macro that is very useful for many applications. This is a macro that you place "around" some Lisp code, in order to make the evaluation of that code happen in some context. For a very simple example, we could define a macro called with-output-in-base, that executes the forms within its body with any output of numbers that is done defaulting to a specified base.
```
(defmacro with-output-in-base ((base-form) &body body)
   `(let ((base ,base-form))
      . ,body))
```
A typical use of this macro might look like:

```
      (with-output-in-base (*default-base*)
         (print x)
         (print y))
```
which would expand into
```
      (let ((base *default-base*))
        (print x)
        (print y))
```

This example is too trivial to be very useful; it is intended to demonstrate some stylistic issues. There are some special forms in Zetalisp that are similar to this macro; see with-open-file (page 365) and with-input-from-string (page 133), for example. The really interesting thing, of course, is that you can define your own such special forms for your own specialized applications. One very powerful application of this technique was used in a system that manipulates and solves the Rubik's cube puzzle. The system heavily uses a special form called with-front-and-top, whose meaning is "evaluate this code in a context in which this specified face of the cube is considered the front face, and this other specified face is considered the top face".

The first thing to keep in mind when you write this sort of macro is that you can make your macro much clearer to people who might read your program if you conform to a set of loose standards of syntactic style. By convention, the names of such special forms start with "with-". This seems to be a clear way of expressing the concept that we are setting up a context; the meaning of the special form is "do this stuff *with* the following things true". Another convention is that any "parameters" to the special form should appear in a list that is the first subform of the special form, and that the rest of the subforms should make up a body of forms that are evaluated sequentially with the last one returned. All of the examples cited above work this way. In our with-output-in-base example, there was one parameter (the base), which appears as the first (and only) element of a list that is the first subform of the special form. The extra level of parentheses in the printed representation serves to separate the "parameter" forms from the "body" forms so that it is textually apparent which is which; it also provides a convenient way to provide default parameters (a good example is the with-input-from-string special form (page 133), which takes two required and two optional "parameters"). Another convention/technique is to use the &body keyword in the defmacro to tell the editor how to correctly indent the special form (see page 227).

The other thing to keep in mind is that control can leave the special form either by the last form's returning, or by a non-local exit (that is, something doing a *throw). You should write the special form in such a way that everything will be cleaned up appropriately no matter which way control exits. In our with-output-in-base example, there is no problem, because non-local exits undo lambda-bindings. However, in even slightly more complicated cases, an unwind-protect form (see page 49) is needed: the macro must expand into an unwind-protect that surrounds the body, with "cleanup" forms that undo the context-setting-up that the macro did. For example, using-resource (see page 84) is implemented as a macro that does an allocate-resource and then performs the body inside of an unwind-protect that has a deallocate-resource in its "cleanup" forms. This way the allocated resource item will be deallocated whenever control leaves the using-resource special form.

17.4.5 Multiple and Out-of-order Evaluation

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

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

Simple cases will work all right:

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

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

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

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

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

With this definition, the expansion works as follows:

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

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

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

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

This defines test in such a way that the form is only evaluated once, and references to form inside the macro body refer to that value. once-only automatically introduces a lambda-binding of a generated symbol to hold the value of the form. Actually, it is more clever than that; it avoids introducing the lambda-binding for forms whose evaluation is trivial and may be repeated without harm nor cost, such as numbers, symbols, and quoted structure. This is just an optimization that helps produce more efficient code.

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

once-only *Macro*

A once-only form looks like

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

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

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

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

17.4.6 Nesting Macros

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

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

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

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

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

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

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

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

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

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

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

Writing the macros this way is not quite right. It works fine interpreted, but the problem is that it does not work when we try to compile Lisp code that uses these special forms. When code is being compiled, there isn't any interpreter to do the binding in our new let form; macro expansion is done at compile time, but generated code does not get run until the results of the compilation are loaded and run. The way to fix our definitions is to use compiler-let instead of let. compiler-let (see page 203) is a special form that exists specifically to do the sort of thing we are trying to do here. compiler-let is identical to let as far as the interpreter is concerned, so changing our **let** to a **compiler-let** won't affect the behavior in the interpreter; it will continue to work. When the compiler encounters a **compiler-let**, however, it actually performs the bindings that the **compiler-let** specifies, and proceeds to compile the body of the **compiler-let** with all of those bindings in effect. In other words, it acts as the interpreter would.

Here's the right way to write these macros:

```
(defvar *collect-variable*)

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

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

17.4.7 Functions Used During Expansion

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

A particular thing to note is that any functions used by macro-expander functions must be available at compile-time. You can make a function available at compile time by surrounding its defining form with an **(eval-when (compile load eval) ...)**; see page 200 for more details. Doing this means that at compile time the definition of the function will be interpreted, not compiled, and hence will run more slowly. Another approach is to separate macro definitions and the functions they call during expansion into a separate file, often called a "defs" (definitions) file. This file defines all the macros but does not use any of them. It can be separately compiled and loaded up before compiling the main part of the program, which uses the macros. The *system* facility (see chapter 24, page 406) helps keep these various files straight, compiling and loading things in the right order.

17.5 Aids for Debugging Macros

mexp

mexp goes into a loop in which it reads forms and sequentially expands them, printing out the result of each expansion (using the grinder (see page 360) to improve readability). It terminates when it reads an atom (anything that is not a cons). If you type in a form which is not a macro form, there will be no expansions and so it will not type anything out, but just prompt you for another form. This allows you to see what your macros are expanding into, without actually evaluating the result of the expansion.

17.6 Displacing Macros

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

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

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

displace *form expansion*

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

(si:displaced *original-form expansion*)

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

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

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

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

we would write

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

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

defmacro-displace *Macro*

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

Now we can write the displacing version of addone as

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

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

17.7 Advanced Features of Defmacro

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

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

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

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

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

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

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

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

We could define a send-commands macro as follows:

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

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

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

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

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

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

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

17.8 Functions to Expand Macros

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

macroexpand-1 *form*

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

macroexpand *form*

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

17.9 Generalized Variables

In Lisp, a variable is something that can remember one piece of data. The main operations on a variable are to recover that piece of data, and to change it. These might be called *access* and *update*. The concept of variables named by symbols, explained in section 3.1, page 14, can be generalized to any storage location that can remember one piece of data, no matter how that location is named.

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

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

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

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

setf *access-form value* *Macro*

setf takes a form which *accesses* something, and "inverts" it to produce a corresponding form to *update* the thing. A setf expands into an update form, which stores the result of evaluating the form *value* into the place referenced by the *access-form*.

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

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

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

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

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

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

Besides the *access* and *update* conceptual operations on variables, there is a third basic operation, which we might call *locate*. Given the name of a storage cell, the *locate* operation will return the address of that cell as a locative pointer (see chapter 13, page 170). This locative pointer is a kind of name for the variable which is a first-class Lisp data object. It can be passed as an argument to a function which operates on any kind of variable, regardless of how it is named. It can be used to *bind* the variable, using the bind subprimitive (see page 183).

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

locf *access-form* *Macro*

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

Examples:

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

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

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

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

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

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

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

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

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

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

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

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

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

push *item access-form* *Macro*

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

The form

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

replaces the commonly-used construct

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

and is intended to be more explicit and esthetic.

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

pop *access-form* *Macro*

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

Example:
```
    (setq x '(a b c))
    (pop x) => a
    x => (b c)
```
All the caveats that apply to incf apply to pop as well: forms within *ref* may be evaluated more than once.

18. The LOOP Iteration Macro

18.1 Introduction

loop is a Lisp macro which provides a programmable iteration facility. The same loop module operates compatibly in Zetalisp, Maclisp (PDP-10 and Multics), and NIL, and a moderately compatible package is under development for the MDL programming environment. loop was inspired by the "FOR" facility of CLISP in InterLisp; however, it is not compatible and differs in several details.

The general approach is that a form introduced by the word loop generates a single program loop, into which a large variety of features can be incorporated. The loop consists of some initialization (*prologue*) code, a body which may be executed several times, and some exit (*epilogue*) code. Variables may 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 philosophy difference from InterLisp's "FOR" facility.

Note that loop forms are intended to look like stylized English rather than Lisp code. There is a notably low density of parentheses, and many of the keywords are accepted in several synonymous forms to allow writing of more euphonious and grammatical English. Some find this notation verbose and distasteful, while others find it flexible and convenient. The former are invited to stick to do.

Here are some examples to illustrate the use of loop.

```
(defun print-elements-of-list (list-of-elements)
    (loop for element in list-of-elements
          do (print element)))
```

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) which satisfy the predicate interesting-p.

```
(defun find-maximum-element (an-array)
   (loop for i from 0 below (array-dimension-n 1 an-array)
         maximize (aref an-array i)))
```

find-maximum-element returns the maximum of the elements of its argument, a one-dimensional array. For Maclisp, aref could be a macro which 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 which satisfies the predicate frobp. If none is found, an error is generated.

18.2 Clauses

Internally, loop constructs a prog which includes variable bindings, pre-iteration (initialization) code, post-iteration (exit) code, the body of the iteration, and stepping of variables of iteration to their next values (which happens on every iteration after executing the body).

A *clause* consists of the keyword symbol and any Lisp forms and keywords which it deals with. For example,

```
(loop for x in l do (print x)),
```

contains two clauses, "for x in l" and "do (print x)". Certain of the parts of the clause will be described as being *expressions*, e.g. (print x) in the above. An expression can be a single Lisp form, or a series of forms implicitly collected with progn. An expression is terminated by the next following atom, which is taken to be a keyword. This syntax allows only the first form in an expression to be atomic, but makes misspelled keywords more easily detectable.

loop uses print-name equality to compare keywords so that loop forms may be written without package prefixes; in Lisp implementations that do not have packages, eq is used for comparison.

Bindings and iteration variable steppings may be performed either sequentially or in parallel, 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.

18.2.1 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 which 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 which produce "body" code: that is, all except those which produce prologue or epilogue code (initially and finally), bindings (with), the named clause, and the iteration termination clauses (while and until).

Clauses which drive the iteration may be arranged to perform their testing and stepping either in series or in parallel. They are by default grouped in series, which allows the stepping computation of one clause to use the just-computed values of the iteration variables of previous clauses. They may be made to step "in parallel", as is the case with the do special form, by "joining" the iteration clauses with the keyword and. The form this typically takes is something like

```
(loop ... for x = (f) and for y = init then (g x) ...)
```

which sets x to (f) on every iteration, and binds y to the value of *init* for the first iteration, and on every iteration thereafter sets it to (g x), where x still has the value from the *previous* iteration. Thus, if the calls to f and g are not order-dependent, this would be best written as

```
(loop ... for y = init then (g x) for x = (f) ...)
```

because, as a general rule, parallel stepping has more overhead than sequential stepping. Similarly, the example

```
(loop for sublist on some-list
      and for previous = 'undefined then sublist
      ...)
```

which is equivalent to the do construct

```
(do ((sublist some-list (cdr sublist))
     (previous 'undefined sublist))
    ((null sublist) ...)
  ...)
```

in terms of stepping, would be better written as

```
(loop for previous = 'undefined then sublist
      for sublist on some-list
      ...)
```

When iteration driving clauses are joined with **and**, if the token following the **and** is not a keyword which introduces an iteration driving clause, it is assumed to be the same as the keyword which 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 which are only evaluated once are evaluated in order at the beginning of the form, during the variable-binding phase, while those expressions which 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 zero or negative result, the body code will not be executed.

All remaining iteration driving clauses are subdispatches of the keyword **for**, which is synonomous with **as**. In all of them a *variable of iteration* is specified. Note that, in general, if an iteration driving clause implicitly supplies an endtest, the value of this iteration variable as the loop is exited (i.e., when the epilogue code is run) is undefined. (This is discussed in more detail in section 18.6.)

Here are all of the varieties of **for** clauses. Optional parts are enclosed in curly brackets. The *data-types* as used here are discussed fully in section 18.4.

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*, page 246. 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* (re-evaluated) at all but the first iteration. Since *expr1* is evaluated during the binding phase, it cannot reference other iteration variables set before it; for that, use the following:

for *var* {*data-type*} first *expr1* then *expr2*

This sets *var* to *expr1* on the first iteration, and to *expr2* (re-evaluated) on each succeeding iteration. The evaluation of both expressions is performed *inside* of the loop binding environment, before the loop body. This allows the first value of *var* to come from the first value of some other iteration variable, allowing such constructs as

```
(loop for term in poly
      for ans first (car term) then (gcd ans (car term))
      finally (return ans))
```

for *var* {*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; i.e. the code will not work if *expr3* is negative or zero. If no limit-specifying clause is given, then the direction of the stepping may be specified as 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 on Iteration Paths (page 249) for complete documentation.

18.2.2 Bindings

The with keyword may be used to establish initial bindings, that is, variables which 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; destructuring and data type keywords are discussed in sections 18.5 and 18.4.

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.

18.2.3 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 which terminates the iteration without running the epilogue code; this behavior is noted with those clauses. Most notable of these are those described in the section 18.2.7, Aggregated Boolean Tests. This clause may be used to cause the loop to return values in a non-standard way:

```
(loop for n in l
      sum n into the-sum
      count t into the-count
      finally (return (quotient the-sum the-count)))
```

18.2.4 Side Effects

do *expression*
doing *expression*

expression is evaluated each time through the loop, as shown in the **print-elements-of-list** example on page 233.

18.2.5 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 nreverses 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 nconced or appended 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 may there be multiple *accumulations* in a loop, but a single *accumulation* may come from multiple places *within the same* loop *form*. Obviously, the types of the collection must be compatible. collect, nconc, and append may all be mixed, as may sum and count, and maximize and minimize. For example,

```
      (loop for x in '(a b c) for y in '((1 2) (3 4) (5 6))
            collect x
            append y)
        => (a 1 2 b 3 4 c 5 6)
```

The following computes the average of the entries in the list *list-of-frobs*:

```
      (loop for x in list-of-frobs
            count t into count-var
            sum x into sum-var
          finally (return (quotient sum-var count-var)))
```

18.2.6 Endtests

The following clauses may be used to provide additional control over when the iteration gets terminated, possibly causing exit code (due to **finally**) to be performed and possibly returning a value (e.g., from **collect**).

while *expr*
: If *expr* evaluates to **nil**, the loop is exited, performing exit code (if any), and returning any accumulated value. The test is placed in the body of the loop where it is written. It may appear between sequential **for** clauses.

until *expr*
: Identical to **while** (**not** *expr*).

This may be needed, for example, to step through a strange data structure, as in

```
(loop until (top-of-concept-tree? concept)
      for concept = expr then (superior-concept concept)
      ...)
```

Note that the placement of the **while** 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.

18.2.7 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.

18.2.8 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 (memoize x)
           else do (process4 x))
```

Useful with the conditionalization clauses is the return clause, which causes an explicit return of its "argument" as the value of the iteration, bypassing any epilogue code. That is,

when *expr1* return *expr2*

is equivalent to

when *expr1* do (return *expr2*)

Conditionalization of one of the "aggregated boolean value" clauses simply causes the test which 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.

18.2.9 Miscellaneous Other Clauses

named *name*

This gives the prog which 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 one instead desires the loop to have some return value when it finishes normally, one may place a call to the **return** function in the epilogue (with the **finally** clause, page 239).

18.3 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),
```

one may now write an iteration as

```
(for i from 1 below n do ...)
```

This facility exists primarily for diehard users of a predecessor of **loop**. Its unconstrained use is not recommended, as it tends to decrease the transportability of the code and needlessly uses up a function name.

18.4 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 point 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 (i.e., anything else).

Note that explicit specification of a non-numeric type for an operation which 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, one can use the nodeclare clause, which is described on page 239.

18.5 Destructuring

Destructuring provides one with the ability to "simultaneously" assign or bind multiple variables to components of some data structure. Typically this is used with list structure. For example,

```
    (loop with (foo . bar) = '(a b c) ...)
```

has the effect of binding foo to a and bar to (b c).

loop's destructuring support is intended to parallel 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, one may still use list-structure patterns as loop iteration variables, and in with bindings. In NIL, loop also supports destructuring over vectors.

One may specify the data types of the components of a pattern by using a corresponding pattern of the data type keywords in place of a single data type keyword. This syntax remains unambiguous because wherever a data type keyword is possible, a loop keyword is the only other possibility. Thus, if one wants to do

```
    (loop for x in l
          as i fixnum = (car x)
          and j fixnum = (cadr x)
          and k fixnum = (cddr x)
          ...)
```

and no reference to x is needed, one 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, one can cause loop to use already provided destructuring support instead:

si:loop-use-system-destructuring? *Variable*

This variable *only* exists in loop implementations in Lisps which 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 which *do* provide destructuring support: destructuring binding will be performed using let, and destructuring assignment will be performed using desetq. Presumably if one's personalized environment supplies these macros, then one should set this variable to t; there is, however, little (if any) efficiency loss if this is not done.

18.6 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 which determines if it is safe to step to the next value of the iteration variable.

steps
: Variables which get "stepped". This is internally manipulated as a list of the form (*var1 val1 var2 val2* ...); all of those variables are stepped in parallel, meaning that all of the *val*s are evaluated before any of the *var*s are set.

post-step-endtest
: Sometimes you can't see if you are done until you step to the next value; that is, the endtest is a function of the stepped-to value.

pseudo-steps
: Other things which need to be stepped. This is typically used for internal variables which are more conveniently stepped here, or to set up iteration variables which are functions of some internal variable(s) which 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 which **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 which 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 which indicates whether one iteration has been performed. In general, sequential iterations have less overhead than parallel iterations, both from the inherent overhead of stepping multiple variables in parallel, and from the standpoint of potential code duplication.

One other point which must be noted about parallel stepping is that although the user iteration variables are guaranteed to be stepped in parallel, the placement of the endtest for any particular iteration may be either before or after the stepping. A notable case of this is

```
(loop for i from 1 to 3 and dummy = (print 'foo)
      collect i)
   => (1 2 3)
```

but prints **foo** *four* times. Certain other constructs, such as **for** *var* **on**, may or may not do this depending on the particular construction.

This problem also means that it may not be safe to examine an iteration variable in the epilogue of the loop form. As a general rule, if an iteration driving clause implicitly supplies an endtest, then one cannot know the state of the iteration variable when the loop terminates. Although one can guess on the basis of whether the iteration variable itself holds the data upon which the endtest is based, that guess *may* be wrong. Thus,

```
(loop for subl on expr
      ...
      finally (f subl))
```

is incorrect, but

```
(loop as frob = expr while (g frob)
      ...
      finally (f frob))
```

is safe because the endtest is explicitly dissociated from the stepping.

18.7 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 which 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 which 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)
```

To satisfy the anthropomorphic among you, **his**, **her**, or **their** may be substituted for the **its** keyword, as may **each**. Egocentricity is not condoned. Some example uses of iteration paths are shown in section 18.7.1.

Very often, iteration paths step internal variables which the user does not specify, such as an index into some data-structure. Although in most cases the user does not wish to be concerned with such low-level matters, it is occasionally useful to have a handle on such things. **loop** provides an additional syntax with which one may provide a variable name to be used as an "internal" variable by an iteration path, with the **using** "prepositional phrase". The **using** phrase is placed with the other phrases associated with the path, and contains any number of keyword/variable-name pairs:

```
(loop for x being the array-elements of a using (index i)
      ...)
```

which says that the variable **i** should be used to hold the index of the array being stepped through. The particular keywords which may be used are defined by the iteration path; the **index** keyword is recognized by all **loop** sequence paths (section 18.7.1.2). 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 which wish to use a special syntax for some feature they provide.

18.7.1 Pre-Defined Paths

loop comes with two pre-defined iteration path functions; one implements a **mapatoms**-like iteration path facility, and the other is used for defining iteration paths for stepping through sequences.

18.7.1.1 The Interned-Symbols Path

The **interned-symbols** iteration path is like a **mapatoms** for **loop**.
```
        (loop for sym being interned-symbols ...)
```
iterates over all of the symbols in the current package and its superiors (or, in Maclisp, the current obarray). This is the same set of symbols which **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 with some sort of hierarchical package structure such as Zetalisp, one may 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.

18.7.1.2 Sequence Iteration

One very common form of iteration is that over the elements of some object which is accessible by means of an integer index. loop defines an iteration path function for doing this in a general way, and provides a simple interface to allow users to define iteration paths for various kinds of "indexable" data.

define-loop-sequence-path *Macro*

```
(define-loop-sequence-path path-name-or-names
    fetch-fun size-fun
    sequence-type default-var-type)
```

path-name-or-names is either an atomic path name or list of path names. *fetch-fun* is a function of two arguments: the sequence, and the index of the item to be fetched. (Indexing is assumed to be zero-origined.) *size-fun* is a function of one argument, the sequence; it should return the number of elements in the sequence. *sequence-type* is the name of the data-type of the sequence, and *default-var-type* the name of the data-type of the elements of the sequence. These 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 of the same keywords which are accepted by the loop arithmetic stepper (for *var* from ...); they are by, to, downto, from, downfrom, below, and above, and are interpreted in the same manner. Thus,

```
     (loop for var being the array-elements of array
              from 1 by 2
           ...)
```

steps *var* over all of the odd elements of *array*, and

```
     (loop for var being the array-elements of array
              downto 0
           ...)
```

steps in "reverse" order.

```
     (define-loop-sequence-path (vector-elements vector-element)
         vref vector-length notype notype)
```

is how the vector-elements iteration path can be defined in NIL (which is). One can then do such things as

```
     (defun cons-a-lot (item &restv other-items)
         (and other-items
              (loop for x being the vector-elements of other-items
                    collect (cons item x))))
```

All such sequence iteration paths allow one to specify the variable to be used as the index variable, by use of the index keyword with the using prepositional phrase, as described (with an example) on page 249.

18.7.2 Defining Paths

This section and the next may not be of interest to those not interested in defining their own iteration paths.

A loop iteration clause (e.g. a for or as clause) produces, in addition to the code which defines the iteration (section 18.6), variables which must be bound, and pre-iteration (*prologue*) code. This breakdown allows a user-interface to loop which does not have to depend on or know about the internals of loop. To complete this separation, the iteration path mechanism parses the clause before giving it to the user function which will return those items. A function to generate code for a path may be declared to loop with the define-loop-path function:

define-loop-path *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 which 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 (e.g. an of phrase when the iteration is inclusive, and an in phrase for the default-loop-path path); the ordering will show the order of evaluation 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* (immediately below) may be used by the path function such that a single function may handle similar paths.

data This is the list of "data" declared for the 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 which 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 which should be included in the loop prologue.

the four items of the iteration specification
These are the four items described in section 18.6, page 247: *pre-step-endtest*, *steps*, *post-step-endtest*, and *pseudo-steps*.

another four items of iteration specification
If these four items are given, they apply to the first iteration, and the previous four apply to all succeeding iterations; otherwise, the previous four apply to *all* iterations.

Here are the routines which are used by loop to compare keywords for equality. In all cases, a *token* may be any Lisp object, but a *keyword* is expected to be an atomic symbol. In certain implementations these functions may be implemented as macros.

si:loop-tequal *token keyword*
This is the loop token comparison function. *token* is any Lisp object; *keyword* is the keyword it is to be compared against. It returns t if they represent the same token, comparing in a manner appropriate for the implementation.

si:loop-tmember *token keyword-list*
The member variant of si:loop-tequal.

si:loop-tassoc *token keyword-alist*
The assoc variant of si:loop-tequal.

If an iteration path function desires to make an internal variable accessible to the user, it should call the following function instead of gensym:

si:loop-named-variable *keyword*

This should only be called from within an iteration path function. If *keyword* has been specified in a using phrase for this path, the corresponding variable is returned; otherwise, gensym is called and that new symbol returned. Within a given path function, this routine should only be called once for any given keyword.

If the user specifies a using preposition containing any keywords for which the path function does not call si:loop-named-variable, loop will inform the user of his error.

18.7.2.1 An Example Path Definition

Here is an example function which 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 gensymmed), and the stepping of it (1+) is such that it may be performed safely without an endtest. Thus index-var may be stepped immediately after the setting of the user's variable, causing the iteration specification for the first iteration to be identical to the iteration specification for all remaining iterations. This is advantageous from the standpoint of the optimizations loop is able to perform, although it is frequently not possible due to the semantics of the iteration (e.g., for *var* first *expr1* then *expr2*) or to subtleties of the stepping. It is safe for this path to step the user's variable in the *pseudo-steps* (the fourth item of an iteration specification) rather than the "real" steps (the second), because the step value can have no dependencies on any other (user) iteration variables. Using the pseudo-steps generally results in some efficiency gains.

If one desired the index variable in the above definition to be user-accessible through the using phrase feature with the index keyword, the function would need to be changed in two ways. First, index-var should be bound to (si:loop-named-variable 'index) instead of (gensym). Secondly, the efficiency hack of stepping the index variable ahead of the iteration variable must not be done. This is effected by changing the last form to be

```
(list bindings prologue
      nil
      (list index-var '(1+ ,index-var))
      '(= ,index-var ,size-var)
      (list variable '(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.

19. Defstruct

19.1 Introduction to Structure Macros

defstruct provides a facility in Lisp for creating and using aggregate datatypes with named elements. These are like "structures" in PL/I, or "records" in PASCAL. In the last two chapters we saw how to use macros to extend the control structures of Lisp; here we see how they can be used to extend Lisp's data structures as well.

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

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

What we would like to see are names, easy to remember and to understand. If the symbol foo were bound to a representation of a space ship, then
```
(ship-x-position foo)
```
could return its x-position, and
```
(ship-y-position foo)
```
its y-position, and so forth. The defstruct facility does just this.

defstruct itself is a macro which defines a structure. For the space ship example above, we might define the structure by saying:
```
(defstruct (ship)
     ship-x-position
     ship-y-position
     ship-x-velocity
     ship-y-velocity
     ship-mass)
```

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

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

We also want to be able to change the contents of a structure. To do this, we use the setf macro (see page 229), as follows (for example):

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

Here x is bound to a ship, and after the evaluation of the setf form, the ship-x-position of that ship will be 100. Another way to change the contents of a structure is to use the alterant macro, which is described later, in section 19.4.3, page 269.

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

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

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

The accessor functions are not actually ordinary functions; instead they are substs (see section 10.5.1, page 143). This difference has two implications: it allows setf to understand the accessor functions, and it allows the compiler to substitute the body of an accessor function directly into any function that uses it, making compiled programs that use defstruct exactly equal in efficiency to programs that "do it by hand." Thus writing (ship-mass s) is exactly equivalent to writing (aref s 4), and writing (setf (ship-mass s) m) is exactly equivalent to writing (aset m s 4), when the program is compiled. It is also possible to tell defstruct to implement the accessor functions as macros; this is not normally done in Zetalisp, however.

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

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

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

(The describe-defstruct function is explained more fully on page 260.)

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

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

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

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

19.2 How to Use Defstruct

defstruct *Macro*

A call to **defstruct** looks like:

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

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

Each *option* may be either a symbol, which should be one of the recognized option names, listed in the next section, or a list, whose car should be one of the option names and the rest of which should be "arguments" to the option. Some options have arguments that default; others require that arguments be given explicitly.

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

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

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

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

In form (1), *slot-name* simply defines a slot with the given name. An accessor function will be defined with the name *slot-name* (but see the **:conc-name** option, page 262). Form (2) is similar, but allows a default initialization for the slot. Initialization is explained further on page 267. Form (3) lets you pack several slots into a single element of the physical underlying structure, using the byte field feature of **defstruct**, which is explained on page 269.

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

To systematize this necessary carefulness, as well as for clarity in the code, it is conventional to prefix the names of all of the accessor functions with some text unique to the structure. In the example above, all the names started with **ship-**. The **:conc-name** option can be used to provide such prefixes automatically (see page 262). Similarly, the conventional name for the constructor macro in the example above was **make-ship**, and the conventional name for the alterant macro (see section 19.4.3, page 269) was **alter-ship**.

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

describe-defstruct *instance* &optional *name*

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

If *instance* is a named structure, you don't have to provide *name*, since it is just the named structure symbol of *instance*. Normally the **describe** function (see page 500) will call **describe-defstruct** if it is asked to describe a named structure; however some named structures have their own idea of how to describe themselves. See page 271 for more information about named structures.

19.3 Options to Defstruct

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

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

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

:type The :type option specifies what kind of Lisp object will be used to implement the structure. It must be given one argument, which must be one of the symbols enumerated below, or a user-defined type. If the option itself is not provided, the type defaults to **:array**. You can define your own types; this is explained in section 19.9, page 275.

:array Use an array, storing components in the body of the array.

:named-array
: Like :array, but make the array a named structure (see page 271) using the name of the structure as the named structure symbol. Element 0 of the array will hold the named structure symbol and so will not be used to hold a component of the structure.

:array-leader
: Use an array, storing components in the leader of the array. (See the :make-array option, described below.)

:named-array-leader
: Like :array-leader, but make the array a named structure (see page 271) using the name of the structure as the named structure symbol. Element 1 of the leader will hold the named structure symbol and so will not be used to hold a component of the structure.

:list Use a list.

:named-list
: Like :list, but the first element of the list will hold the symbol that is the name of the structure and so will not be used as a component.

:fixnum-array
: Like :array, but the type of the array is **art-32b**.

:flonum-array
: Like :array, but the type of the array is **art-float**.

:tree The structure is implemented out of a binary tree of conses, with the leaves serving as the slots.

:fixnum
: This unusual type implements the structure as a single fixnum. The structure may only have one slot. This is only useful with the byte field feature (see page 269); it lets you store a bunch of small numbers within fields of a fixnum, giving the fields names.

:grouped-array
: This is described in section 19.6, page 271.

:constructor This option takes one argument, which specifies the name of the constructor macro. If the argument is not provided or if the option itself is not provided, the name of the constructor is made by concatenating the string "make-" to the name of the structure. If the argument is provided and is nil, no constructor is defined. Use of the constructor macro is explained in section 19.4.1, page 267.

:alterant This option takes one argument, which specifies the name of the alterant macro. If the argument is not provided or if the option itself is not provided, the name of the alterant is made by concatenating the string "alter-" to the name of the structure. If the argument is provided and is nil, no alterant is defined. Use of the alterant macro is explained in section 19.4.3, page 269.

:default-pointer

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

:make-array If the structure being defined is implemented as an array, this option may be used to control those aspects of the array that are not otherwise constrained by defstruct. For example, you might want to control the area in which the array is allocated. Also, if you are creating a structure of type :array-leader, you almost certainly want to specify the dimensions of the array to be created, and you may want to specify the type of the array. Of course, this option is only meaningful if the structure is, in fact, being implemented by an array.

The argument to the :make-array option should be a list of alternating keyword symbols to the make-array function (see page 111), and forms whose values are the arguments to those keywords. For example, (:make-array (:type 'art-16b)) would request that the type of the array be art-16b. Note that the keyword symbol is *not* evaluated.

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

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

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

:times This option is used for structures of type :grouped-array to control the number of repetitions of the structure that will be allocated by the constructor macro. (See section 19.6, page 271.) The constructor macro will also allow :times to be used as a keyword that will override the value given in the original defstruct form. If :times appears in neither the invocation of the constructor nor in the :make-array option to defstruct, then the constructor will only allocate one instance of the structure.

:size-symbol The :size-symbol option allows a user to specify a global variable whose value will be the "size" of the structure; this variable is declared with defconst. The exact meaning of the size varies, but in general this number is the one you would need to know if you were going to allocate one of these structures yourself. The symbol will have this value both at compile time and at run time. If this option is present without an argument, then the name of the structure is concatenated with "-size" to produce the symbol.

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

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

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

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

The accessors expand like this:

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

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

:displace Normally all of the macros defined by **defstruct** will be simple displacing macros. They will use the function **displace** to actually change the original macro form, so that it will not have to be expanded over and over (see page 226). The :displace option allows the user to supply some other function to use instead of **displace**.

The argument to the :displace option should be a two argument function that will be called whenever a macro expansion occurs. The two arguments will be the original form and the form resulting from macro expansion. The value returned by this function will be used for further evaluation. Note that the function **displace** is the function used if the :displace option isn't given. The function **progn** will cause the macro to be expanded every time.

Giving the :displace argument with no arguments, or with an argument of t, or with an argument of displace, is the same is not giving it at all. Giving an argument of nil or progn means to use regular macros instead of displacing macros.

Note that accessor functions are normally substs rather than macros (unless you give the :callable-accessors option with argument nil). If the accessors are substs, they are not affected by :displace. However, the constructor and alterant macros, and the :size-macro, are still affected.

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

:eval-when
: Normally the functions and macros defined by defstruct are defined at eval-time, compile-time, and load-time. This option allows the user to control this behavior. The argument to the :eval-when option is just like the list that is the first subform of an eval-when special form (see page 200). For example, (:eval-when (:eval :compile)) will cause the functions and macros to be defined only when the code is running interpreted or inside the compiler.

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

The :property option can be used to give a defstruct an arbitrary property. (:property *property-name value*) gives the defstruct a *property-name* property of *value*. Neither argument is evaluated. To access the property list, the user will have to look inside the defstruct-description structure himself (see page 273).

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

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

This provides a primitive way for you to define your own options to **defstruct**, particularly in connection with user-defined types (see section 19.9, page 275). Several of the options listed above are actually implemented using this mechanism.

19.4 Using the Constructor and Alterant Macros

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

19.4.1 Constructor Macros

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

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

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

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

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

There are two symbols which are specially recognized by the constructor. They are **:make-array**, which should only be used for **:array** and **:array-leader** type structures (or the named versions of those types), and **:times**, which should only be used for **:grouped-array** type structures. If one of these symbols appears instead of a slot name, then it is interpreted just as the **:make-array** option or the **:times** option (see page 264), and it overrides what was requested

in that option. For example:
```
    (make-ship ship-x-position 10.0
               ship-y-position 12.0
               :make-array (:leader-length 5 :area disaster-area))
```

19.4.2 By-position Constructor Macros

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

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

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

This defines make-foo to be a constructor of one or more arguments. The first argument is used to initialize the a slot. The second argument is used to initialize the b slot. If there isn't any second argument, then the default value given in the body of the defstruct (if given) is used instead. The third argument is used to initialize the c slot. If there isn't any third argument, then the symbol sea is used instead. Any arguments following the third argument are collected into a list and used to initialize the d slot. If there are three or fewer arguments, then nil is placed in the d slot. The e slot *is not initialized*; its initial value is undefined. Finally, the f slot is initialized to contain the symbol eff.

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

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

Note that even these "function style" constructors do not guarantee that their arguments will be evaluated in the order that you wrote them. Also note that you cannot specify the :make-array nor :times information in this form of constructor macro..

19.4.3 Alterant Macros

A call to the alterant macro, in general, has the form

```
(name-of-alterant-macro instance-form
     slot-name-1 form-1
     slot-name-2 form-2
     ...)
```

instance-form is evaluated, and should return an instance of the structure. Each *form* is evaluated, and the corresponding slot is changed to have the result as its new value. The slots are altered after all the *forms* are evaluated, so you can exchange the values of two slots, as follows:

```
(alter-ship enterprise
    ship-x-position (ship-y-position enterprise)
    ship-y-position (ship-x-position enterprise))
```

As with the constructor macro, the order of evaluation of the *forms* is undefined. Using the alterant macro can produce more efficient Lisp than using consecutive setfs when you are altering two byte fields of the same object, or when you are using the :but-first option.

19.5 Byte Fields

The byte field feature of defstruct allows you to specify that several slots of your structure are bytes (see section 7.7, page 102) in an integer stored in one element of the structure. For example, suppose we had the following structure:

```
(defstruct (phone-book-entry (:type :list))
  name
  address
  (area-code 617.)
  exchange
  line-number)
```

This will work correctly. However, it wastes space. Area codes and exchange numbers are always less than 1000., and so both can fit into 10. bit fields when expressed as binary numbers. Since Lisp Machine fixnums have (more than) 20. bits, both of these values can be packed into a single fixnum. To tell defstruct to do so, you can change the structure definition to the following:

```
(defstruct (phone-book-entry (:type :list))
  name
  address
  ((area-code #o1212 617.)
   (exchange #o0012))
  line-number)
```

The magic octal numbers #o1212 and #o0012 are byte specifiers to be used with the functions ldb and dpb. The accessors, constructor, and alterant will now operate as follows:

```
(area-code pbe) ==> (ldb #o1212 (caddr pbe))
(exchange pbe)  ==> (ldb #o0012 (caddr pbe))

(make-phone-book-entry
   name "Fred Derf"
   address "259 Octal St."
   exchange ex
   line-number 7788.)

==> (list "Fred Derf" "259 Octal St." (dpb ex 12 2322000) 17154)

(alter-phone-book-entry pbe
   area-code ac
   exchange ex)

==> ((lambda (g0530)
       (setf (nth 2 g0530)
             (dpb ac 1212 (dpb ex 12 (nth 2 g0530)))))
     pbe)
```

Note that the alterant macro is optimized to only read and write the second element of the list once, even though you are altering two different byte fields within it. This is more efficient than using two **setf**s. Additional optimization by the alterant macro occurs if the byte specifiers in the **defstruct** slot descriptions are constants. However, you don't need to worry about the details of how the alterant macro does its work.

If the byte specifier is **nil**, then the accessor will be defined to be the usual kind that accesses the entire Lisp object, thus returning all the byte field components as a fixnum. These slots may have default initialization forms.

The byte specifier need not be a constant; a variable or, indeed, any Lisp form, is legal as a byte specifier. It will be evaluated each time the slot is accessed. Of course, unless you are doing something very strange you will not want the byte specifier to change between accesses.

Constructor macros initialize words divided into byte fields as if they were deposited in in the following order:

1) Initializations for the entire word given in the defstruct form.

2) Initializations for the byte fields given in the defstruct form.

3) Initializations for the entire word given in the constructor macro form.

4) Initializations for the byte fields given in the constructor macro form.

Alterant macros work similarly: the modification for the entire Lisp object is done first, followed by modifications to specific byte fields. If any byte fields being initialized or altered overlap each other, the action of the constructor and alterant will be unpredictable.

19.6 Grouped Arrays

The grouped array feature allows you to store several instances of a structure side-by-side within an array. This feature is somewhat limited; it does not support the :include and :named options.

The accessor functions are defined to take an extra argument, which should be an integer, and is the index into the array of where this instance of the structure starts. This index should normally be a multiple of the size of the structure, for things to make sense. Note that the index is the *first* argument to the accessor function and the structure is the *second* argument, the opposite of what you might expect. This is because the structure is &optional if the :default-pointer option is used.

Note that the "size" of the structure (for purposes of the :size-symbol and :size-macro options) is the number of elements in *one* instance of the structure; the actual length of the array is the product of the size of the structure and the number of instances. The number of instances to be created by the constructor macro is given as the argument to the :times option to defstruct, or the :times keyword of the constructor macro.

19.7 Named Structures

The *named structure* feature provides a very simple form of user-defined data type. Any array may be made a named structure, although usually the :named option of defstruct is used to create named structures. The principal advantages to a named structure are that it has a more informative printed representation than a normal array and that the describe function knows how to give a detailed description of it. (You don't have to use describe-defstruct, because describe can figure out what the names of the slots of the structure are by looking at the named structure's name.) Because of these improved user-interface features it is recommended that "system" data structures be implemented with named structures.

Another kind of user-defined data type, more advanced but less efficient when just used as a record structure, is provided by the *flavor* feature (see chapter 20, page 279).

A named structure has an associated symbol, called its "named structure symbol", which represents what user-defined type it is an instance of; the typep function, applied to the named structure, will return this symbol. If the array has a leader, then the symbol is found in element 1 of the leader; otherwise it is found in element 0 of the array. (Note: if a numeric-type array is to be a named structure, it must have a leader, since a symbol cannot be stored in any element of a numeric array.)

If you call typep with two arguments, the first being an instance of a named structure and the second being its named structure symbol, typep will return t. t will also be returned if the second argument is the named structure symbol of a :named defstruct included (using the :include option, see page 262), directly or indirectly, by the defstruct for this structure. For example, if the structure astronaut includes the structure person, and person is a named structure, then giving typep an instance of an astronaut as the first argument, and the symbol person as the second argument, will return t. This reflects the fact that an astronaut is, in fact, a person, as well as being an astronaut.

You may associate with a named structure a function that will handle various operations that can be done on the named structure. Currently, you can control how the named structure is printed, and what describe will do with it.

To provide such a handler function, make the function be the named-structure-invoke property of the named structure symbol. The functions which know about named structures will apply this handler function to several arguments. The first is a "keyword" symbol to identify the calling function, and the second is the named structure itself. The rest of the arguments passed depend on the caller; any named structure function should have a "&rest" parameter to absorb any extra arguments that might be passed. Just what the function is expected to do depends on the keyword it is passed as its first argument. The following are the keywords defined at present:

:which-operations
: Should return a list of the names of the operations the function handles.

:print-self
: The arguments are :print-self, the named structure, the stream to output to, the current depth in list-structure, and t if slashification is enabled (prin1 versus princ). The printed representation of the named structure should be output to the stream. If the named structure symbol is not defined as a function, or :print-self is not in its :which-operations list, the printer will default to a reasonable printed representation, namely:
  #<named-structure-symbol octal-address>

:describe
: The arguments are :describe and the named structure. It should output a description of itself to standard-output. If the named structure symbol is not defined as a function, or :describe is not in its :which-operations list, the describe system will check whether the named structure was created by using the :named option of defstruct; if so, the names and values of the structure's fields will be enumerated.

Here is an example of a simple named-structure handler function:

```
(defun (person named-structure-invoke) (op self &rest args)
  (selectq op
    (:which-operations '(:print-self))
    (:print-self
      (format (first args)
              (if (third args) "#<person ~A>" "~A")
              (person-name self)))
    (otherwise (ferror nil "Illegal operation ~S" op))))
```

For this definition to have any effect, the person defstruct used as an example earlier must be modified to include the :named attribute.

This handler causes a person structure to include its name in its printed representation; it also causes princ of a person to print just the name, with no "#<" syntax. Even though the astronaut structure of our examples :includes the person structure, this named-structure handler will not be invoked when an astronaut is printed, and an astronaut will not include his name in his printed representation. This is because named structures are not as general as flavors (see chapter 20, page 279).

The following functions operate on named structures.

named-structure-p *x*

This semi-predicate returns **nil** if *x* is not a named structure; otherwise it returns *x*'s named structure symbol.

named-structure-symbol *x*

x should be a named structure. This returns *x*'s named structure symbol: if *x* has an array leader, element 1 of the leader is returned, otherwise element 0 of the array is returned.

make-array-into-named-structure *array*

array is made to be a named structure, and is returned.

named-structure-invoke *operation structure* &rest *args*

operation should be a keyword symbol, and *structure* should be a named structure. The handler function of the named structure symbol, found as the value of the **named-structure-invoke** property of the symbol, is called with appropriate arguments. (This function used to take its first two arguments in the opposite order, and that argument order will continue to work indefinitely, but it should not be used in new programs.)

See also the **:named-structure-symbol** keyword to **make-array**, page 111.

19.8 The si:defstruct-description Structure

This section discusses the internal structures used by **defstruct** that might be useful to programs that want to interface to **defstruct** nicely. For example, if you want to write a program that examines structures and displays them the way **describe** (see page 500) and the Inspector do, your program will work by examining these structures. The information in this section is also necessary for anyone who is thinking of defining his own structure types.

Whenever the user defines a new structure using **defstruct**, **defstruct** creates an instance of the **si:defstruct-description** structure. This structure can be found as the **si:defstruct-description** property of the name of the structure; it contains such useful information as the name of the structure, the number of slots in the structure, and so on.

The **si:defstruct-description** structure is defined as follows, in the **system-internals** package (also called the **si** package): (This is a simplified version of the real definition. There are other slots in the structure which we aren't telling you about.)

```
(defstruct (defstruct-description
               (:default-pointer description)
               (:conc-name defstruct-description-))
           name
           size
           property-alist
           slot-alist)
```

The **name** slot contains the symbol supplied by the user to be the name of his structure, such as **spaceship** or **phone-book-entry**.

The **size** slot contains the total number of locations in an instance of this kind of structure. This is *not* the same number as that obtained from the :size-symbol or :size-macro options to **defstruct**. A named structure, for example, usually uses up an extra location to store the name of the structure, so the :size-macro option will get a number one larger than that stored in the **defstruct** description.

The **property-alist** slot contains an alist with pairs of the form (*property-name* . *property*) containing properties placed there by the :property option to **defstruct** or by property names used as options to **defstruct** (see the :property option, page 266).

The **slot-alist** slot contains an alist of pairs of the form (*slot-name* . *slot-description*). A *slot-description* is an instance of the **defstruct-slot-description** structure. The **defstruct-slot-description** structure is defined something like this, also in the **si** package: (This is a simplified version of the real definition. There are other slots in the structure which we aren't telling you about.)

```
(defstruct (defstruct-slot-description
                (:default-pointer slot-description)
                (:conc-name defstruct-slot-description-))
           number
           ppss
           init-code
           ref-macro-name)
```

The **number** slot contains the number of the location of this slot in an instance of the structure. Locations are numbered starting with 0, and continuing up to one less than the size of the structure. The actual location of the slot is determined by the reference-consing function associated with the type of the structure; see page 276.

The **ppss** slot contains the byte specifier code for this slot if this slot is a byte field of its location. If this slot is the entire location, then the **ppss** slot contains **nil**.

The **init-code** slot contains the initialization code supplied for this slot by the user in his **defstruct** form. If there is no initialization code for this slot then the init-code slot contains the symbol **si:%%defstruct-empty%%**.

The **ref-macro-name** slot contains the symbol that is defined as a macro or a subst that expands into a reference to this slot (that is, the name of the accessor function).

19.9 Extensions to Defstruct

The macro defstruct-define-type can be used to teach defstruct about new types that it can use to implement structures.

defstruct-define-type *Macro*

This macro is used for teaching defstruct about new types; it is described in the rest of this chapter.

19.9.1 An Example

Let us start by examining a sample call to defstruct-define-type. This is how the :list type of structure might have been defined:

```
(defstruct-define-type :list
        (:cons (initialization-list description keyword-options)
               :list
               '(list . ,initialization-list))
        (:ref (slot-number description argument)
               '(nth ,slot-number ,argument)))
```

This is the simplest possible form of defstruct-define-type. It provides defstruct with two Lisp forms: one for creating forms to construct instances of the structure, and one for creating forms to become the bodies of accessors for slots of the structure.

The keyword :cons is followed by a list of three variables that will be bound while the constructor-creating form is evaluated. The first, initialization-list, will be bound to a list of the initialization forms for the slots of the structure. The second, description, will be bound to the defstruct-description structure for the structure (see page 273). The third variable and the :list keyword will be explained later.

The keyword :ref is followed by a list of three variables that will be bound while the accessor-creating form is evaluated. The first, slot-number, will bound to the number of the slot that the new accessor should reference. The second, description, will be bound to the defstruct-description structure for the structure. The third, argument, will be bound to the form that was provided as the argument to the accessor.

19.9.2 Syntax of defstruct-define-type

The syntax of defstruct-define-type is:

(defstruct-define-type *type*
    option-1
    option-2
    ...)

where each *option* is either the symbolic name of an option or a list of the form (*option-name rest*). Different options interpret *rest* in different ways. The symbol *type* is given an si:defstruct-type-description property of a structure that describes the type completely.

19.9.3 Options to defstruct-define-type

This section is a catalog of all the options currently known about by defstruct-define-type.

:cons The :cons option to defstruct-define-type is how you supply defstruct with the necessary code that it needs to cons up a form that will construct an instance of a structure of this type.

The :cons option has the syntax:

```
(:cons (inits description keywords) kind
       body)
```

body is some code that should construct and return a piece of code that will construct, initialize, and return an instance of a structure of this type.

The symbol *inits* will be bound to the information that the constructor conser should use to initialize the slots of the structure. The exact form of this argument is determined by the symbol *kind*. There are currently two kinds of initialization. There is the :list kind, where *inits* is bound to a list of initializations, in the correct order, with nils in uninitialized slots. And there is the :alist kind, where *inits* is bound to an alist with pairs of the form (*slot-number . init-code*).

The symbol *description* will be bound to the instance of the defstruct-description structure (see page 273) that defstruct maintains for this particular structure. This is so that the constructor conser can find out such things as the total size of the structure it is supposed to create.

The symbol *keywords* will be bound to an alist with pairs of the form (*keyword value*), where each *keyword* was a keyword supplied to the constructor macro that wasn't the name of a slot, and *value* was the Lisp object that followed the keyword. This is how you can make your own special keywords, like the existing :make-array and :times keywords. See the section on using the constructor macro, section 19.4.1, page 267. You specify the list of acceptable keywords with the :keywords option (see page 277).

It is an error not to supply the :cons option to defstruct-define-type.

:ref The :ref option to defstruct-define-type is how the user supplies defstruct with the necessary code that it needs to cons up a form that will reference an instance of a structure of this type.

The :ref option has the syntax:

```
(:ref (number description arg-1 ... arg-n)
      body)
```

body is some code that should construct and return a piece of code that will reference an instance of a structure of this type.

The symbol *number* will be bound to the location of the slot that is to be referenced. This is the same number that is found in the number slot of the defstruct-slot-description structure (see page 274).

The symbol *description* will be bound to the instance of the **defstruct-description** structure that **defstruct** maintains for this particular structure.

The symbols *arg-i* are bound to the forms supplied to the accessor as arguments. Normally there should be only one of these. The *last* argument is the one that will be defaulted by the :default-pointer option (see page 262). **defstruct** will check that the user has supplied exactly *n* arguments to the accessor function before calling the reference consing code.

It is an error not to supply the :ref option to **defstruct-define-type**.

:overhead — The :overhead option to **defstruct-define-type** is how the user declares to **defstruct** that the implementation of this particular type of structure "uses up" some number of locations in the object actually constructed. This option is used by various "named" types of structures that store the name of the structure in one location.

The syntax of :overhead is: (:overhead *n*) where *n* is a fixnum that says how many locations of overhead this type needs.

This number is only used by the :size-macro and :size-symbol options to **defstruct** (see page 264).

:named — The :named option to **defstruct-define-type** controls the use of the :named option to **defstruct**. With no argument, the :named option means that this type is an acceptable "named structure". With an argument, as in (:named *type-name*), the symbol *type-name* should be the name of some other structure type that **defstruct** should use if someone asks for the named version of this type. (For example, in the definition of the :list type the :named option is used like this: (:named :named-list).)

:keywords — The :keywords option to **defstruct-define-type** allows the user to define additional constructor keywords for this type of structure. (The :make-array constructor keyword for structures of type :array is an example.) The syntax is: (:keywords *keyword-1* ... *keyword-n*) where each *keyword* is a symbol that the constructor conser expects to find in the *keywords* alist (explained above).

:defstruct — The :defstruct option to **defstruct-define-type** allows the user to run some code and return some forms as part of the expansion of the **defstruct** macro.

The :defstruct option has the syntax:

```
        (:defstruct (description)
                    body)
```

body is a piece of code that will be run whenever **defstruct** is expanding a **defstruct** form that defines a structure of this type. The symbol *description* will be bound to the instance of the **defstruct-description** structure that **defstruct** maintains for this particular structure.

The value returned by the *body* should be a *list* of forms to be included with those that the **defstruct** expands into. Thus, if you only want to run some code at **defstruct-expand** time and you don't want to actually output any additional code, then you should be careful to return nil from the code in this option.

20. Objects, Message Passing, and Flavors

20.1 Introduction

The object oriented programming style used in the Smalltalk and Actor families of languages is available in Zetalisp, and used by the Lisp Machine software system. Its purpose is to perform *generic operations* on objects. Part of its implementation is simply a convention in procedure calling style; part is a powerful language feature, called Flavors, for defining abstract objects. This chapter attempts to explain what programming with objects and with message passing means, the various means of implementing these in Zetalisp, and when you should use them. It assumes no prior knowledge of any other languages.

20.2 Objects

When writing a program, it is often convenient to model what the program does in terms of *objects*: conceptual entities that can be likened to real-world things. Choosing what objects to provide in a program is very important to the proper organization of the program. In an object-oriented design, specifying what objects exist is the first task in designing the system. In a text editor, the objects might be "pieces of text", "pointers into text", and "display windows". In an electrical design system, the objects might be "resistors", "capacitors", "transistors", "wires", and "display windows". After specifying what objects there are, the next task of the design is to figure out what operations can be performed on each object. In the text editor example, operations on "pieces of text" might include inserting text and deleting text; operations on "pointers into text" might include moving forward and backward; and operations on "display windows" might include redisplaying the window and changing with which "piece of text" the window is associated.

In this model, we think of the program as being built around a set of objects, each of which has a set of operations that can be performed on it. More rigorously, the program defines several *types* of object (the editor above has three types), and it can create many *instances* of each type (that is, there can be many pieces of text, many pointers into text, and many windows). The program defines a set of types of object, and the operations that can be performed on any of the instances of each type.

This should not be wholly unfamiliar to the reader. Earlier in this manual, we saw a few examples of this kind of programming. A simple example is disembodied property lists, and the functions **get**, **putprop**, and **remprop**. The disembodied property list is a type of object; you can instantiate one with **(cons nil nil)** (that is, by evaluating this form you can create a new disembodied property list); there are three operations on the object, namely **get**, **putprop**, and **remprop**. Another example in the manual was the first example of the use of **defstruct**, which was called a **ship**. **defstruct** automatically defined some operations on this object: the operations to access its elements. We could define other functions that did useful things with **ships**, such as computing their speed, angle of travel, momentum, or velocity, stopping them, moving them elsewhere, and so on.

In both cases, we represent our conceptual object by one Lisp object. The Lisp object we use for the representation has *structure*, and refers to other Lisp objects. In the property list case, the Lisp object is a list with alternating indicators and values; in the ship case, the Lisp object is an array whose details are taken care of by defstruct. In both cases, we can say that the object keeps track of an *internal state*, which can be *examined* and *altered* by the operations available for that type of object. get examines the state of a property list, and putprop alters it; ship-x-position examines the state of a ship, and (setf (ship-mass) 5.0) alters it.

We have now seen the essence of object-oriented programming. A conceptual object is modelled by a single Lisp object, which bundles up some state information. For every type of object, there is a set of operations that can be performed to examine or alter the state of the object.

20.3 Modularity

An important benefit of the object-oriented style is that it lends itself to a particularly simple and lucid kind of modularity. If you have modular programming constructs and techniques available, it helps and encourages you to write programs that are easy to read and understand, and so are more reliable and maintainable. Object-oriented programming lets a programmer implement a useful facility that presents the caller with a set of external interfaces, without requiring the caller to understand how the internal details of the implementation work. In other words, a program that calls this facility can treat the facility as a black box; the program knows what the facility's external interfaces guarantee to do, and that is all it knows.

For example, a program that uses disembodied property lists never needs to know that the property list is being maintained as a list of alternating indicators and values; the program simply performs the operations, passing them inputs and getting back outputs. The program only depends on the external definition of these operations: it knows that if it putprops a property, and doesn't remprop it (or putprop over it), then it can do get and be sure of getting back the same thing it put in. The important thing about this hiding of the details of the implementation is that someone reading a program that uses disembodied property lists need not concern himself with how they are implemented; he need only understand what they undertake to do. This saves the programmer a lot of time, and lets him concentrate his energies on understanding the program he is working on. Another good thing about this hiding is that the representation of property lists could be changed, and the program would continue to work. For example, instead of a list of alternating elements, the property list could be implemented as an association list or a hash table. Nothing in the calling program would change at all.

The same is true of the ship example. The caller is presented with a collection of operations, such as ship-x-position, ship-y-position, ship-speed, and ship-direction; it simply calls these and looks at their answers, without caring how they did what they did. In our example above, ship-x-position and ship-y-position would be accessor functions, defined automatically by defstruct, while ship-speed and ship-direction would be functions defined by the implementor of the ship type. The code might look like this:

```
(defstruct (ship)
  ship-x-position
  ship-y-position
  ship-x-velocity
  ship-y-velocity
  ship-mass)

(defun ship-speed (ship)
  (sqrt (+ (^ (ship-x-velocity ship) 2)
           (^ (ship-y-velocity ship) 2))))

(defun ship-direction (ship)
  (atan (ship-y-velocity ship)
        (ship-x-velocity ship)))
```

The caller need not know that the first two functions were structure accessors and that the second two were written by hand and do arithmetic. Those facts would not be considered part of the black box characteristics of the implementation of the ship type. The ship type does not guarantee which functions will be implemented in which ways; such aspects are not part of the contract between ship and its callers. In fact, ship could have been written this way instead:

```
(defstruct (ship)
  ship-x-position
  ship-y-position
  ship-speed
  ship-direction
  ship-mass)

(defun ship-x-velocity (ship)
  (* (ship-speed ship) (cos (ship-direction ship))))

(defun ship-y-velocity (ship)
  (* (ship-speed ship) (sin (ship-direction ship))))
```

In this second implementation of the ship type, we have decided to store the velocity in polar coordinates instead of rectangular coordinates. This is purely an implementation decision; the caller has no idea which of the two ways the implementation works, because he just performs the operations on the object by calling the appropriate functions.

We have now created our own types of objects, whose implementations are hidden from the programs that use them. Such types are usually referred to as *abstract types*. The object-oriented style of programming can be used to create abstract types by hiding the implementation of the operations, and simply documenting what the operations are defined to do.

Some more terminology: the quantities being held by the elements of the ship structure are referred to as *instance variables*. Each instance of a type has the same operations defined on it; what distinguishes one instance from another (besides identity (eqness)) is the values that reside in its instance variables. The example above illustrates that a caller of operations does not know what the instance variables are; our two ways of writing the ship operations have different instance variables, but from the outside they have exactly the same operations.

One might ask: "But what if the caller evaluates (aref ship 2) and notices that he gets back the x-velocity rather than the speed? Then he can tell which of the two implementations were used." This is true; if the caller were to do that, he could tell. However, when a facility is implemented in the object-oriented style, only certain functions are documented and advertised: the functions which are considered to be operations on the type of object. The contract from ship to its callers only speaks about what happens if the caller calls these functions. The contract makes no guarantees at all about what would happen if the caller were to start poking around on his own using aref. A caller who does so *is in error*; he is depending on something that is not specified in the contract. No guarantees were ever made about the results of such action, and so anything may happen; indeed, ship may get reimplemented overnight, and the code that does the aref will have a different effect entirely and probably stop working. This example shows why the concept of a contract between a callee and a caller is important: the contract is what specifies the interface between the two modules.

Unlike some other languages that provide abstract types, Zetalisp makes no attempt to have the language automatically forbid constructs that circumvent the contract. This is intentional. One reason for this is that the Lisp Machine is an interactive system, and so it is important to be able to examine and alter internal state interactively (usually from a debugger). Furthermore, there is no strong distinction between the "system" programs and the "user" programs on the Lisp Machine; users are allowed to get into any part of the language system and change what they want to change.

In summary: by defining a set of operations, and making only a specific set of external entrypoints available to the caller, the programmer can create his own abstract types. These types can be useful facilities for other programs and programmers. Since the implementation of the type is hidden from the callers, modularity is maintained, and the implementation can be changed easily.

We have hidden the implementation of an abstract type by making its operations into functions which the user may call. The important thing is not that they are functions—in Lisp everything is done with functions. The important thing is that we have defined a new conceptual operation and given it a name, rather than requiring anyone who wants to do the operation to write it out step-by-step. Thus we say (ship-x-velocity s) rather than (aref s 2).

It is just as true of such abstract-operation functions as of ordinary functions that sometimes they are simple enough that we want the compiler to compile special code for them rather than really calling the function. (Compiling special code like this is often called *open-coding*.) The compiler is directed to do this through use of macros, defsubsts, or optimizers. defstruct arranges for this kind of special compilation for the functions that get the instance variables of a structure.

When we use this optimization, the implementation of the abstract type is only hidden in a certain sense. It does not appear in the Lisp code written by the user, but does appear in the compiled code. The reason is that there may be some compiled functions that use the macros (or whatever); even if you change the definition of the macro, the existing compiled code will continue to use the old definition. Thus, if the implementation of a module is changed programs that use it may need to be recompiled. This is something we sometimes accept for the sake of efficiency.

In the present implementation of flavors, which is discussed below, there is no such compiler incorporation of nonmodular knowledge into a program, except when the "outside-accessible instance variables" feature is used; see page 303, where this problem is explained further. If you don't use the "outside-accessible instance variables" feature, you don't have to worry about this.

20.4 Generic Operations

Suppose we think about the rest of the program that uses the ship abstraction. It may want to deal with other objects that are like ships in that they are movable objects with mass, but unlike ships in other ways. A more advanced model of a ship might include the concept of the ship's engine power, the number of passengers on board, and its name. An object representing a meteor probably would not have any of these, but might have another attribute such as how much iron is in it.

However, all kinds of movable objects have positions, velocities, and masses, and the system will contain some programs that deal with these quantities in a uniform way, regardless of what kind of object the attributes apply to. For example, a piece of the system that calculates every object's orbit in space need not worry about the other, more peripheral attributes of various types of objects; it works the same way for all objects. Unfortunately, a program that tries to calculate the orbit of a ship will need to know the ship's attributes, and will have to call ship-x-position and ship-y-velocity and so on. The problem is that these functions won't work for meteors. There would have to be a second program to calculate orbits for meteors that would be exactly the same, except that where the first one calls ship-x-position, the second one would call meteor-x-position, and so on. This would be very bad; a great deal of code would have to exist in multiple copies, all of it would have to be maintained in parallel, and it would take up space for no good reason.

What is needed is an operation that can be performed on objects of several different types. For each type, it should do the thing appropriate for that type. Such operations are called *generic* operations. The classic example of generic operations is the arithmetic functions in most programming languages, including Zetalisp. The + (or plus) function will accept either fixnums or flonums, and perform either fixnum addition or flonum addition, whichever is appropriate, based on the data types of the objects being manipulated. In our example, we need a generic x-position operation that can be performed on either ships, meteors, or any other kind of mobile object represented in the system. This way, we can write a single program to calculate orbits. When it wants to know the *x* position of the object it is dealing with, it simply invokes the generic x-position operation on the object, and whatever type of object it has, the correct operation is performed, and the *x* position is returned.

A terminology for the use of such generic operations has emerged from the Smalltalk and Actor languages: performing a generic operation is called *sending a message*. The objects in the program are thought of as little people who get sent messages and respond with answers. In the example above, the objects are sent x-position messages, to which they respond with their *x* position. This *message passing* is how generic operations are performed.

Sending a message is a way of invoking a function. Along with the *name* of the message, in general, some arguments are passed; when the object is done with the message, some values are returned. The sender of the message is simply calling a function with some arguments, and getting some values back. The interesting thing is that the caller did not specify the name of a procedure to call. Instead, it specified a message name and an object; that is, it said what operation to perform, and what object to perform it on. The function to invoke was found from this information.

When a message is sent to an object, a function therefore must be found to handle the message. The two data used to figure out which function to call are the *type* of the object, and the *name* of the message. The same set of functions are used for all instances of a given type, so the type is the only attribute of the object used to figure out which function to call. The rest of the message besides the name are data which are passed as arguments to the function, so the name is the only part of the message used to find the function. Such a function is called a *method*. For example, if we send an x-position message to an object of type ship, then the function we find is "the ship type's x-position method". A method is a function that handles a specific kind of message to a specific kind of object; this method handles messages named x-position to objects of type ship.

In our new terminology: the orbit-calculating program finds the *x* position of the object it is working on by sending that object a message named x-position (with no arguments). The returned value of the message is the *x* position of the object. If the object was of type ship, then the ship type's x-position method was invoked; if it was of type meteor, then the meteor type's x-position method was invoked. The orbit-calculating program just sends the message, and the right function is invoked based on the type of the object. We now have true generic functions, in the form of message passing: the same operation can mean different things depending on the type of the object.

20.5 Generic Operations in Lisp

How do we implement message passing in Lisp? By convention, objects that receive messages are always *functional* objects (that is, you can apply them to arguments), and a message is sent to an object by calling that object as a function, passing the name of the message as the first argument, and the arguments of the message as the rest of the arguments. Message names are represented by symbols; normally these symbols are in the keyword package (see chapter 23, page 392) since messages are a protocol for communication between different programs, which may reside in different packages. So if we have a variable my-ship whose value is an object of type ship, and we want to know its *x* position, we send it a message as follows:

```
(funcall my-ship ':x-position)
```

This form returns the *x* position as its returned value. To set the ship's *x* position to 3.0, we send it a message like this:

```
(funcall my-ship ':set-x-position 3.0)
```

It should be stressed that no new features are added to Lisp for message sending; we simply define a convention on the way objects take arguments. The convention says that an object accepts messages by always interpreting its first argument as a message name. The object must consider this message name, find the function which is the method for that message name, and invoke that function.

This raises the question of how message receiving works. The object must somehow find the right method for the message it is sent. Furthermore, the object now has to be callable as a function; objects can't just be **defstruct**s any more, since those aren't functions. But the structure defined by **defstruct** was doing something useful: it was holding the instance variables (the internal state) of the object. We need a function with internal state; that is, we need a coroutine.

Of the Zetalisp features presented so far, the most appropriate is the closure (see chapter 11, page 158). A message-receiving object could be implemented as a closure over a set of instance variables. The function inside the closure would have a big **selectq** form to dispatch on its first argument. (Actually, rather than using closures and a **selectq**, Zetalisp provides *entities* and **defselect**; see section 11.4, page 162.)

While using closures (or entities) does work, it has several serious problems. The main problem is that in order to add a new operation to a system, it is necessary to modify a lot of code; you have to find all the types that understand that operation, and add a new clause to the **selectq**. The problem with this is that you cannot textually separate the implementation of your new operation from the rest of the system; the methods must be interleaved with the other operations for the type. Adding a new operation should only require *adding* Lisp code; it should not require *modifying* Lisp code.

The conventional way of making generic operations is to have a procedure for each operation, which has a big **selectq** for all the types; this means you have to modify code to add a type. The way described above is to have a procedure for each type, which has a big **selectq** for all the operations; this means you have to modify code to add an operation. Neither of these has the desired property that extending the system should only require adding code, rather than modifying code.

Closures (and entities) are also somewhat clumsy and crude. A far more streamlined, convenient, and powerful system for creating message-receiving objects exists; it is called the *Flavor* mechanism. With flavors, you can add a new method simply by adding code, without modifying anything. Furthermore, many common and useful things to do are very easy to do with flavors. The rest of this chapter describes flavors.

20.6 Simple Use of Flavors

A *flavor*, in its simplest form, is a definition of an abstract type. New flavors are created with the **defflavor** special form, and methods of the flavor are created with the **defmethod** special form. New instances of a flavor are created with the **make-instance** function. This section explains simple uses of these forms.

For an example of a simple use of flavors, here is how the **ship** example above would be implemented.

```
(defflavor ship (x-position y-position
                 x-velocity y-velocity mass)
                ()
  :gettable-instance-variables)

(defmethod (ship :speed) ()
  (sqrt (+ (^ x-velocity 2)
           (^ y-velocity 2))))

(defmethod (ship :direction) ()
  (atan y-velocity x-velocity))
```

The code above creates a new flavor. The first subform of the **defflavor** is **ship**, which is the name of the new flavor. Next is the list of instance variables; they are the five that should be familiar by now. The next subform is something we will get to later. The rest of the subforms are the body of the **defflavor**, and each one specifies an option about this flavor. In our example, there is only one option, namely **:gettable-instance-variables**. This means that for each instance variable, a method should automatically be generated to return the value of that instance variable. The name of the message is a symbol with the same name as the instance variable, but interned on the keyword package. Thus, methods are created to handle the messages **:x-position**, **:y-position**, and so on.

Each of the two **defmethod** forms adds a method to the flavor. The first one adds a handler to the flavor **ship** for messages named **:speed**. The second subform is the lambda-list, and the rest is the body of the function that handles the **:speed** message. The body can refer to or set any instance variables of the flavor, the same as it can with local variables or special variables. When any instance of the **ship** flavor is invoked with a first argument of **:direction**, the body of the second **defmethod** will be evaluated in an environment in which the instance variables of **ship** refer to the instance variables of this instance (the one to which the message was sent). So when the arguments of **atan** are evaluated, the values of instance variables of the object to which the message was sent will be used as the arguments. **atan** will be invoked, and the result it returns will be returned by the instance itself.

Now we have seen how to create a new abstract type: a new flavor. Every instance of this flavor will have the five instance variables named in the **defflavor** form, and the seven methods we have seen (five that were automatically generated because of the **:gettable-instance-variables** option, and two that we wrote ourselves). The way to create an instance of our new flavor is with the **make-instance** function. Here is how it could be used:

```
    (setq my-ship (make-instance 'ship))
```

This will return an object whose printed representation is:

```
    #<SHIP 13731210>
```

(Of course, the value of the magic number will vary; it is not interesting anyway.) The argument to make-instance is, as you can see, the name of the flavor to be instantiated. Additional arguments, not used here, are *init options*, that is, commands to the flavor of which we are making an instance, selecting optional features. This will be discussed more in a moment.

Examination of the flavor we have defined shows that it is quite useless as it stands, since there is no way to set any of the parameters. We can fix this up easily, by putting the :settable-instance-variables option into the defflavor form. This option tells defflavor to generate methods for messages named :set-x-position, :set-y-position, and so on; each such method takes one argument, and sets the corresponding instance variable to the given value.

Another option we can add to the defflavor is :initable-instance-variables, to allow us to initialize the values of the instance variables when an instance is first created. :initable-instance-variables does not create any methods; instead, it makes *initialization keywords* named :x-position, :y-position, etc., that can be used as init-option arguments to make-instance to initialize the corresponding instance variables. The set of init options are sometimes called the *init-plist* because they are like a property list.

Here is the improved defflavor:

```
    (defflavor ship (x-position y-position
                     x-velocity y-velocity mass)
                    ()
      :gettable-instance-variables
      :settable-instance-variables
      :initable-instance-variables)
```

All we have to do is evaluate this new defflavor, and the existing flavor definition will be updated and now include the new methods and initialization options. In fact, the instance we generated a while ago will now be able to accept these new messages! We can set the mass of the ship we created by evaluating

```
    (funcall my-ship ':set-mass 3.0)
```

and the mass instance variable of my-ship will properly get set to 3.0. If you want to play around with flavors, it is useful to know that describe of an instance tells you the flavor of the instance and the values of its instance variables. If we were to evaluate (describe my-ship) at this point, the following would be printed:

```
#<SHIP 13731210>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:         unbound
        Y-POSITION:         unbound
        X-VELOCITY:         unbound
        Y-VELOCITY:         unbound
        MASS:               3.0
```

Now that the instance variables are "initable", we can create another ship and initialize some of the instance variables using the init-plist. Let's do that and describe the result:

```
(setq her-ship (make-instance 'ship ':x-position 0.0
                                    ':y-position 2.0
                                    ':mass 3.5))
                ==> #<SHIP 13756521>


(describe her-ship)
#<SHIP 13756521>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:         0.0
        Y-POSITION:         2.0
        X-VELOCITY:         unbound
        Y-VELOCITY:         unbound
        MASS:               3.5
```

A flavor can also establish default initial values for instance variables. ınese default values are used when a new instance is created if the values are not initialized any other way. The syntax for specifying a default initial value is to replace the name of the instance variable by a list, whose first element is the name and whose second is a form to evaluate to produce the default initial value. For example:

```
(defvar *default-x-velocity* 2.0)
(defvar *default-y-velocity* 3.0)

(defflavor ship ((x-position 0.0)
                 (y-position 0.0)
                 (x-velocity *default-x-velocity*)
                 (y-velocity *default-y-velocity*)
                 mass)
                ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)

(setq another-ship (make-instance 'ship ':x-position 3.4))

(describe another-ship)
#<SHIP 14563643>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:         3.4
        Y-POSITION:         0.0
        X-VELOCITY:         2.0
        Y-VELOCITY:         3.0
        MASS:               unbound
```

x-position was initialized explicitly, so the default was ignored. y-position was initialized from the default value, which was 0.0. The two velocity instance variables were initialized from their default values, which came from two global variables. mass was not explicitly initialized and did not have a default initialization, so it was left unbound.

There are many other options that can be used in defflavor, and the init options can be used more flexibly than just to initialize instance variables; full details are given later in this chapter. But even with the small set of features we have seen so far, it is easy to write object-oriented programs.

20.7 Mixing Flavors

Now we have a system for defining message-receiving objects so that we can have generic operations. If we want to create a new type called meteor that would accept the same generic operations as ship, we could simply write another defflavor and two more defmethods that looked just like those of ship, and then meteors and ships would both accept the same operations. ship would have some more instance variables for holding attributes specific to ships, and some more methods for operations that are not generic, but are only defined for ships; the same would be true of meteor.

However, this would be a a wasteful thing to do. The same code has to be repeated in several places, and several instance variables have to be repeated. The code now needs to be maintained in many places, which is always undesirable. The power of flavors (and the name "flavors") comes from the ability to mix several flavors and get a new flavor. Since the functionality of ship and meteor partially overlap, we can take the common functionality and move it into its own flavor, which might be called moving-object. We would define moving-object the same way as we defined ship in the previous section. Then, ship and meteor could be defined like this:

```
(defflavor ship (engine-power number-of-passengers name)
                (moving-object)
   :gettable-instance-variables)

(defflavor meteor (percent-iron) (moving-object)
   :initable-instance-variables)
```

These defflavor forms use the second subform, which we ignored previously. The second subform is a list of flavors to be combined to form the new flavor; such flavors are called *components*. Concentrating on ship for a moment (analogous things are true of meteor), we see that it has exactly one component flavor: moving-object. It also has a list of instance variables, which includes only the ship-specific instance variables and not the ones that it shares with meteor. By incorporating moving-object, the ship flavor acquires all of its instance variables, and so need not name them again. It also acquires all of moving-object's methods, too. So with the new definition, ship instances will still accept the :x-velocity and :speed messages, and they will do the same thing. However, the :engine-power message will also be understood (and will return the value of the engine-power instance variable).

What we have done here is to take an abstract type, moving-object, and build two more specialized and powerful abstract types on top of it. Any ship or meteor can do anything a moving object can do, and each also has its own specific abilities. This kind of building can continue; we could define a flavor called ship-with-passenger that was built on top of ship, and it would inherit all of moving-object's instance variables and methods as well as ship's instance variables and methods. Furthermore, the second subform of defflavor can be a list of several components, meaning that the new flavor should combine all the instance variables and methods of all the flavors in the list, as well as the ones *those* flavors are built on, and so on. All the components taken together form a big tree of flavors. A flavor is built from its components, its components' components, and so on. We sometimes use the term "components" to mean the immediate components (the ones listed in the defflavor), and sometimes to mean all the components (including the components of the immediate components and so on). (Actually, it is not strictly a tree, since some flavors might be components through more than one path. It is really a directed graph; it can even be cyclic.)

The order in which the components are combined to form a flavor is important. The tree of flavors is turned into an ordered list by performing a *top-down, depth-first* walk of the tree, including non-terminal nodes *before* the subtrees they head, and eliminating duplicates. For example, if flavor-1's immediate components are flavor-2 and flavor-3, and flavor-2's components are flavor-4 and flavor-5, and flavor-3's component was flavor-4, then the complete list of components of flavor-1 would be:

```
        flavor-1, flavor-2, flavor-4, flavor-5, flavor-3
```

The flavors earlier in this list are the more specific, less basic ones; in our example, ship-with-passengers would be first in the list, followed by ship, followed by moving-object. A flavor is always the first in the list of its own components. Notice that flavor-4 does not appear twice in this list. Only the first occurrence of a flavor appears; duplicates are removed. (The elimination of duplicates is done during the walk; if there is a cycle in the directed graph, it will not cause a non-terminating computation.)

The set of instance variables for the new flavor is the union of all the sets of instance variables in all the component flavors. If both flavor-2 and flavor-3 have instance variables named foo, then flavor-1 will have an instance variable named foo, and any methods that refer to foo will refer to this same instance variable. Thus different components of a flavor can communicate with one another using shared instance variables. (Typically, only one component ever sets the variable, and the others only look at it.) The default initial value for an instance variable comes from the first component flavor to specify one.

The way the methods of the components are combined is the heart of the flavor system. When a flavor is defined, a single function, called a *combined method*, is constructed for each message supported by the flavor. This function is constructed out of all the methods for that message from all the components of the flavor. There are many different ways that methods can be combined; these can be selected by the user when a flavor is defined. The user can also create new forms of combination.

There are several kinds of methods, but so far, the only kinds of methods we have seen are *primary* methods, The default way primary methods are combined is that all but the earliest one provided are ignored. In other words, the combined method is simply the primary method of the first flavor to provide a primary method. What this means is that if you are starting with a flavor foo and building a flavor bar on top of it, then you can override foo's method for a message by providing your own method. Your method will be called, and foo's will never be called.

Simple overriding is often useful; if you want to make a new flavor bar that is just like foo except that it reacts completely differently to a few messages, then this will work. However, often you don't want to completely override the base flavor's (foo's) method; sometimes you want to add some extra things to be done. This is where combination of methods is used.

The usual way methods are combined is that one flavor provides a primary method, and other flavors provide *daemon methods*. The idea is that the primary method is "in charge" of the main business of handling the message, but other flavors just want to keep informed that the message was sent, or just want to do the part of the operation associated with their own area of responsibility.

When methods are combined, a single primary method is found; it comes from the first component flavor that has one. Any primary methods belonging to later component flavors are ignored. This is just what we saw above; bar could override foo's primary method by providing its own primary method.

However, you can define other kinds of methods. In particular, you can define *daemon* methods. They come in two kinds, *before* and *after*. There is a special syntax in defmethod for defining such methods. Here is an example of the syntax. To give the ship flavor an after-daemon method for the :speed message, the following syntax would be used:

```
(defmethod (ship :after :speed) ()
   body)
```

Now, when a message is sent, it is handled by a new function called the *combined* method. The combined method first calls all of the before daemons, then the primary method, then all the after daemons. Each method is passed the same arguments that the combined method was given. The returned values from the combined method are the values returned by the primary method; any values returned from the daemons are ignored. Before-daemons are called in the order that flavors are combined, while after-daemons are called in the reverse order. In other words, if you build **bar** on top of **foo**, then **bar**'s before-daemons will run before any of those in **foo**, and **bar**'s after-daemons will run after any of those in **foo**.

The reason for this order is to keep the modularity order correct. If we create **flavor-1** built on **flavor-2**; then it should not matter what **flavor-2** is built out of. Our new before-daemons go before all methods of **flavor-2**, and our new after-daemons go after all methods of **flavor-2**. Note that if you have no daemons, this reduces to the form of combination described above. The most recently added component flavor is the highest level of abstraction; you build a higher-level object on top of a lower-level object by adding new components to the front. The syntax for defining daemon methods can be found in the description of **defmethod** below.

To make this a bit more clear, let's consider a simple example that is easy to play with: the **:print-self** method. The Lisp printer (i.e. the **print** function; see section 21.2.1, page 319) prints instances of flavors by sending them **:print-self** messages. The first argument to the **:print-self** message is a stream (we can ignore the others for now), and the receiver of the message is supposed to print its printed representation on the stream. In the **ship** example above, the reason that instances of the **ship** flavor printed the way they did is because the **ship** flavor was actually built on top of a very basic flavor called **vanilla-flavor**; this component is provided automatically by **defflavor**. It was **vanilla-flavor**'s **:print-self** method that was doing the printing. Now, if we give **ship** its own primary method for the **:print-self** message, then that method will take over the job of printing completely; **vanilla-flavor**'s method will not be called at all. However, if we give **ship** a before-daemon method for the **:print-self** message, then it will get invoked before the **vanilla-flavor** message, and so whatever it prints will appear before what **vanilla-flavor** prints. So we can use before-daemons to add prefixes to a printed representation; similarly, after-daemons can add suffixes.

There are other ways to combine methods besides daemons, but this way is the most common. The more advanced ways of combining methods are explained in a later section; see section 20.12, page 306. The **vanilla-flavor** and what it does for you are also explained later; see section 20.11, page 305.

20.8 Flavor Functions

defflavor *Macro*

A flavor is defined by a form

(defflavor *flavor-name* (*var1 var2*...) (*flav1 flav2*...)
    opt1 opt2...)

flavor-name is a symbol which serves to name this flavor. It will get an **si:flavor** property of the internal data-structure containing the details of the flavor.

(typep *obj*), where *obj* is an instance of the flavor named *flavor-name*, will return the symbol *flavor-name*. (typep *obj flavor-name*) is t if *obj* is an instance of a flavor, one of whose components (possibly itself) is *flavor-name*.

var1, *var2*, etc. are the names of the instance-variables containing the local state for this flavor. A list of the name of an instance-variable and a default initialization form is also acceptable; the initialization form will be evaluated when an instance of the flavor is created if no other initial value for the variable is obtained. If no initialization is specified, the variable will remain unbound.

flav1, *flav2*, etc. are the names of the component flavors out of which this flavor is built. The features of those flavors are inherited as described previously.

opt1, *opt2*, etc. are options; each option may be either a keyword symbol or a list of a keyword symbol and arguments. The options to **defflavor** are described on section 20.9, page 300.

all-flavor-names *Variable*

This is a list of the names of all the flavors that have ever been **defflavor**'ed.

defmethod *Macro*

A method, that is, a function to handle a particular message sent to an instance of a particular flavor, is defined by a form such as

```
(defmethod (flavor-name method-type message) lambda-list
  form1 form2...)
```

flavor-name is a symbol which is the name of the flavor which is to receive the method. *method-type* is a keyword symbol for the type of method; it is omitted when you are defining a primary method, which is the usual case. *message* is a keyword symbol which names the message to be handled.

The meaning of the *method-type* depends on what kind of method-combination is declared for this message. For instance, for daemons **:before** and **:after** are allowed. See section 20.12, page 306 for a complete description of method types and the way methods are combined.

lambda-list describes the arguments and "aux variables" of the function; the first argument to the method, which is the message keyword, is automatically handled, and so it is not included in the *lambda-list*. Note that methods may not have **&quote** arguments; that is they must be functions, not special forms. *form1*, *form2*, etc. are the function body; the value of the last form is returned.

The variant form

```
(defmethod (flavor-name message) function)
```

where *function* is a symbol, says that *flavor-name*'s method for *message* is *function*, a symbol which names a function. That function must take appropriate arguments; the first argument is the message keyword.

If you redefine a method that is already defined, the old definition is replaced by the new one. Given a flavor, a message name, and a method type, there can only be one function, so if you define a :before daemon method for the foo flavor to handle the :bar message, then you replace the previous before-daemon; however, you do not affect the primary method or methods of any other type, message name or flavor.

The function spec for a method (see section 10.2, page 137) looks like:
    (:method *flavor-name message*) or
    (:method *flavor-name method-type message*)
This is useful to know if you want to trace (page 457) or advise (page 460) a method, or if you want to poke around at the method function itself, e.g. disassemble (page 500) it.

defmethod actually defines a symbol, called the *flavor-method-symbol*, as a function, and the flavor system goes through that symbol to call the method. The flavor-method-symbol is formed by concatenating (with hyphens) the flavor name, the method type, the message name, and "method" (for example, ship-x-position-method, ship-after-y-velocity-method, ship-combined-mass-method, etc.). The property list of this symbol is used to allow undefun (page 150) and uncompile (page 197) to work. This is likely to be changed in the future.

make-instance *flavor-name init-option1 value1 init-option2 value2...*

Creates and returns an instance of the specified flavor. Arguments after the first are alternating init-option keywords and arguments to those keywords. These options are used to initialize instance variables and to select arbitrary options, as described above. If the flavor supports the :init message, it is sent to the newly-created object with one argument, the init-plist. This is a disembodied property-list containing the init-options specified and those defaulted from the flavor's :default-init-plist. make-instance is an easy-to-call interface to instantiate-flavor; for full details refer to that function.

instantiate-flavor *flavor-name init-plist* &optional *send-init-message-p return-unhandled-keywords area*

This is an extended version of make-instance, giving you more features. Note that it takes the init-plist as an argument, rather than taking a &rest argument of init-options and values.

The *init-plist* argument must be a disembodied property list; locf of a &rest argument will do. Beware! This property list can be modified; the properties from the default-init-plist are putprop'ed on if not already present, and some :init methods do explicit putprops onto the init-plist.

In the event that :init methods do remprop of properties already on the init-plist (as opposed to simply doing get and putprop), then the init-plist will get rplacd'ed. This means that the actual list of options will be modified. It also means that locf of a &rest argument will not work; the caller of instantiate-flavor must copy its rest argument (e.g. with append); this is because rplacd is not allowed on &rest arguments.

First, if the flavor's method-table and other internal information have not been computed or are not up to date, they are computed. This may take a substantial amount of time and invoke the compiler, but will only happen once for a particular flavor no matter how many instances you make, unless you change something.

Next, the instance variables are initialized. There are several ways this initialization can happen. If an instance variable is declared initable, and a keyword with the same spelling as its name appears in *init-plist*, it is set to the value specified after that keyword. If an instance variable does not get initialized this way, and an initialization form was specified for it in a **defflavor**, that form is evaluated and the variable is set to the result. The initialization form may not depend on any instance variables nor on **self**; it will not be evaluated in the "inside" environment in which methods are called. If an instance variable does not get initialized either of these ways it will be left unbound; presumably an **:init** method should initialize it (see below). Note that a simple empty disembodied property list is **(nil)**, which is what you should give if you want an empty init-plist. If you use **nil**, the property list of **nil** will be used, which is probably not what you want.

If any keyword appears in the *init-plist* but is not used to initialize an instance variable and is not declared in an **:init-keywords** option (see page 300) it is presumed to be a misspelling. So any keywords that you handle in an **:init** handler should also be mentioned in the **:init-keywords** option of the definition of the flavor.

If the *return-unhandled-keywords* argument is not supplied, such keywords are complained about by signalling an error. But if *return-unhandled-keywords* is supplied non-**nil**, a list of such keywords is returned as the second value of **instantiate-flavor**.

Note that default values in the *init-plist* can come from the **:default-init-plist** option to **defflavor**. See page 300.

If the *send-init-message-p* argument is supplied and non-**nil**, an **:init** message is sent to the newly-created instance, with one argument, the *init-plist*. **get** can be used to extract options from this property-list. Each flavor that needs initialization can contribute an **:init** method, by defining a daemon.

If the *area* argument is specified, it is the number of an area in which to cons the instance; otherwise it is consed in the default area.

defwrapper *Macro*

This is hairy and if you don't understand it you should skip it.

Sometimes the way the flavor system combines the methods of different flavors (the daemon system) is not powerful enough. In that case **defwrapper** can be used to define a macro which expands into code which is wrapped around the invocation of the methods. This is best explained by an example; suppose you needed a lock locked during the processing of the **:foo** message to the **bar** flavor, which takes two arguments, and you have a **lock-frobboz** special-form which knows how to lock the lock (presumably it generates an **unwind-protect**). **lock-frobboz** needs to see the first argument to the message; perhaps that tells it what sort of operation is going to be performed (read or write).

```
(defwrapper (bar :foo) ((arg1 arg2) . body)
  `(lock-frobboz (self arg1)
     . ,body))
```

The use of the **body** macro-argument prevents the **defwrapper**'ed macro from knowing the exact implementation and allows several **defwrappers** from different flavors to be combined properly.

Note well that the argument variables, **arg1** and **arg2**, are not referenced with commas before them. These may look like **defmacro** "argument" variables, but they are not. Those variables are not bound at the time the **defwrapper**-defined macro is expanded and the back-quoting is done; rather the result of that macro-expansion and back-quoting is code which, when a message is sent, will bind those variables to the arguments in the message as local variables of the combined method.

Consider another example. Suppose you thought you wanted a **:before** daemon, but found that if the argument was **nil** you needed to return from processing the message immediately, without executing the primary method. You could write a wrapper such as

```
(defwrapper (bar :foo) ((arg1) . body)
  `(cond ((null arg1))          ;Do nothing if arg1 is nil
         (t before-code
            . ,body)))
```

Suppose you need a variable for communication among the daemons for a particular message; perhaps the **:after** daemons need to know what the primary method did, and it is something that cannot be easily deduced from just the arguments. You might use an instance variable for this, or you might create a special variable which is bound during the processing of the message and used free by the methods.

```
(defvar *communication*)
(defwrapper (bar :foo) (ignore . body)
  `(let ((*communication* nil))
     . ,body))
```

Similarly you might want a wrapper which puts a ***catch** around the processing of a message so that any one of the methods could throw out in the event of an unexpected condition.

If you change a wrapper, the change may not take effect automatically. You must use **recompile-flavor** with a third argument of **nil** to force the effect to propagate into the compiled code which the system generates to implement the flavor. The reason for this is that the flavor system cannot reliably tell the difference between reloading a file containing a wrapper and really redefining the wrapper to be different, and propagating a change to a wrapper is expensive. [This may be fixed in the future.]

Like daemon methods, wrappers work in outside-in order; when you add a **defwrapper** to a flavor built on other flavors, the new wrapper is placed outside any wrappers of the component flavors. However, *all* wrappers happen before *any* daemons happen. When the combined method is built, the calls to the before-daemon methods, primary methods, and after-daemon methods are all placed together, and then the wrappers are wrapped around them. Thus, if a component flavor defines a wrapper, methods added by new flavors will execute within that wrapper's context.

undefmethod (*flavor* [*type*] *message*) *Macro*

`(undefmethod (flavor :before :message))`
removes the method created by
`(defmethod (flavor :before :message) (`*args*`) ...)`

To remove a wrapper, use undefmethod with :wrapper as the method type.

self *Variable*

When a message is sent to an object, the variable self is automatically bound to that object, for the benefit of methods which want to manipulate the object itself (as opposed to its instance variables).

funcall-self *message arguments...*

When self is an instance or an entity, (funcall-self *args...*) has the same effect as (funcall self *args...*) except that it is a little faster since it doesn't have to re-establish the context in which the instance variables evaluate correctly. If self is not an instance (nor an "entity", see section 11.4, page 162), funcall-self and funcall self do the same thing.

When self is an instance, funcall-self will only work correctly if it is used in a method or a function, wrapped in a declare-flavor-instance-variables, that was called (not necessarily directly) from a method. Otherwise the instance-variables will not be already set up.

lexpr-funcall-self *message arguments... list-of-arguments*

This function is a cross between lexpr-funcall and funcall-self. When self is an instance or an entity, (lexpr-funcall-self *args...*) has the same effect as (lexpr-funcall self *args...*) except that it is a little faster since it doesn't have to re-establish the context in which the instance variables evaluate correctly. If self is not an instance (nor an "entity", see section 11.4, page 162), lexpr-funcall-self and lexpr-funcall do the same thing.

declare-flavor-instance-variables (*flavor*) *body...* *Macro*

Sometimes you will write a function which is not itself a method, but which is to be called by methods and wants to be able to access the instance variables of the object self. The form

`(declare-flavor-instance-variables (`*flavor-name*`)`
    function-definition`)`

surrounds the *function-definition* with a declaration of the instance variables for the specified flavor, which will make them accessible by name. Currently this works by declaring them as special variables, but this implementation may be changed in the future. Note that it is only legal to call a function defined this way while executing inside a method for an object of the specified flavor, or of some flavor built upon it.

recompile-flavor *flavor-name* &optional *single-message* (*use-old-combined-methods* t) (*do-dependents* t)

Updates the internal data of the flavor and any flavors that depend on it. If *single-message* is supplied non-nil, only the methods for that message are changed. The system does this when you define a new method that did not previously exist. If *use-old-*

combined-methods is t, then the existing combined method functions will be used if possible. New ones will only be generated if the set of methods to be called has changed. This is the default. If *use-old-combined-methods* is nil, automatically-generated functions to call multiple methods or to contain code generated by wrappers will be regenerated unconditionally. If you change a wrapper, you must do recompile-flavor with third argument nil in order to make the new wrapper take effect. If *do-dependents* is nil, only the specific flavor you specified will be recompiled. Normally it and all flavors that depend on it will be recompiled.

recompile-flavor only affects flavors that have already been compiled. Typically this means it affects flavors that have been instantiated, but does not bother with mixins (see page 304).

compile-flavor-methods *flavor...* *Macro*

The form (compile-flavor-methods *flavor-name-1 flavor-name-2...*), placed in a file to be compiled, will cause the compiler to include the automatically-generated combined methods for the named flavors in the resulting qfasl file, provided all of the necessary flavor definitions have been made. Furthermore, when the qfasl file is loaded, internal data structures (such as the list of all methods of a flavor) will get generated.

This means that the combined methods get compiled at compile time, and the data structures get generated at load time, rather than both things happening at run time. This is a very good thing to use, since the need to invoke the compiler at run-time makes programs that use flavors slow the first time they are run. (The compiler will still be called if incompatible changes have been made, such as addition or deletion of methods that must be called by a combined method.)

You should only use compile-flavor-methods for flavors that are going to be instantiated. For a flavor that will never be instantiated (that is, a flavor that only serves to be a component of other flavors that actually do get instantiated), it is a complete waste of time.

The compile-flavor-methods forms should be compiled after all of the information needed to create the combined methods is available. You should put these forms after all of the definitions of all relevant flavors, wrappers, and methods of all components of the flavors mentioned.

When a compile-flavor-methods form is seen by the interpreter, the combined methods are compiled and the internal data structures are generated.

get-handler-for *object message*

Given an object and a message, will return that object's method for that message, or nil if it has none. When *object* is an instance of a flavor, this function can be useful to find which of that flavor's components supplies the method. If you get back a combined method, you can use the List Combined Methods editor command (page 312) to find out what it does.

This function can be used with other things than flavors, and has an optional argument which is not relevant here and not documented.

flavor-allows-init-keyword-p *flavor-name keyword*
Returns non-nil if the flavor named *flavor-name* allows *keyword* in the init options when it is instantiated, or nil if it does not. The non-nil value is the name of the component flavor which contributes the support of that keyword.

symeval-in-instance *instance symbol* &optional *no-error-p*
This function is used to find the value of an instance variable inside a particular instance. *Instance* is the instance to be examined, and *symbol* is the instance variable whose value should be returned. If there is no such instance variable, an error is signalled, unless *no-error-p* is non-nil in which case nil is returned.

set-in-instance *instance symbol value*
This function is used to alter the value of an instance variable inside a particular instance. *Instance* is the instance to be altered, *symbol* is the instance variable whose value should be set, and *value* is the new value. If there is no such instance variable, an error is signalled.

locate-in-instance *instance symbol*
Returns a locative pointer to the cell inside *instance* which holds the value of the instance variable named *symbol*.

describe-flavor *flavor-name*
This function prints out descriptive information about a flavor; it is self-explanatory. An important thing it tells you that can be hard to figure out yourself is the combined list of component flavors; this list is what is printed after the phrase "and directly or indirectly depends on".

si:*flavor-compilations* *Variable*
This variable contains a history of when the flavor mechanism invoked the compiler. It is a list; elements toward the front of the list represent more recent compilations. Elements are typically of the form
(:method *flavor-name type message-name*)
and *type* is typically :combined.

You may setq this variable to nil at any time; for instance before loading some files that you suspect may have missing or obsolete compile-flavor-methods in them.

20.9 Defflavor Options

There are quite a few options to **defflavor**. They are all described here, although some are for very specialized purposes and not of interest to most users. Each option can be written in two forms; either the keyword by itself, or a list of the keyword and "arguments" to that keyword.

Several of these options declare things about instance variables. These options can be given with arguments which are instance variables, or without any arguments in which case they refer to all of the instance variables listed at the top of the **defflavor**. This is *not* necessarily all the instance variables of the component flavors; just the ones mentioned in this flavor's **defflavor**. When arguments are given, they must be instance variables that were listed at the top of the **defflavor**; otherwise they are assumed to be misspelled and an error is signalled. It is legal to declare things about instance variables inherited from a component flavor, but to do so you must list these instance variables explicitly in the instance variable list at the top of the **defflavor**.

:gettable-instance-variables
: Enables automatic generation of methods for getting the values of instance variables. The message name is the name of the variable, in the keyword package (i.e. put a colon in front of it.)

:settable-instance-variables
: Enables automatic generation of methods for setting the values of instance variables. The message name is ":set-" followed by the name of the variable. All settable instance variables are also automatically made gettable and initable.

:initable-instance-variables
: The instance variables listed as arguments, or all instance variables listed in this **defflavor** if the keyword is given alone, are made *initable*. This means that they can be initialized through use of a keyword (a colon followed by the name of the variable) as an init-option argument to make-instance.

:init-keywords
: The arguments are declared to be keywords in the initialization property-list which are processed by this flavor's :init methods. The system uses this for error-checking: before the system sends the :init message, it makes sure that all the keywords in the init-plist are either initable-instance-variables, or elements of this list. If the caller misspells a keyword or otherwise uses a keyword that no component flavor handles, this feature will signal an error. When you write a :init handler that accepts some keywords, they should be listed in the :init-keywords option of the flavor.

:default-init-plist
: The arguments are alternating keywords and value forms, like a property-list. When the flavor is instantiated, these properties and values are put into the init-plist unless already present. This allows one component flavor to default an option to another component flavor. The value forms are only evaluated when and if they are used. For example,

```
(:default-init-plist :frob-array
                     (make-array 100))
```

would provide a default "frob array" for any instance for which the user did not provide one explicitly.

:required-instance-variables
: Declares that any flavor incorporating this one which is instantiated into an object must

contain the specified instance variables. An error occurs if there is an attempt to instantiate a flavor that incorporates this one if it does not have these in its set of instance variables. Note that this option is not one of those which checks the spelling of its arguments in the way described at the start of this section (if it did, it would be useless).

Required instance variables may be freely accessed by methods just like normal instance variables. The difference between listing instance variables here and listing them at the front of the **defflavor** is that the latter declares that this flavor "owns" those variables and will take care of initializing them, while the former declares that this flavor depends on those variables but that some other flavor must be provided to manage them and whatever features they imply.

:required-methods

The arguments are names of messages which any flavor incorporating this one must handle. An error occurs if there is an attempt to instantiate such a flavor and it is lacking a method for one of these messages. Typically this option appears in the **defflavor** for a base flavor (see page 304). Usually this is used when a base flavor does a **funcall-self** (page 297) to send itself a message that is not handled by the base flavor itself; the idea is that the base flavor will not be instantiated alone, but only with other components (mixins) that do handle the message. This keyword allows the error of having no handler for the message be detected when the flavor is defined (which usually means at compile time) rather than at run time.

:required-flavors

The arguments are names of flavors which any flavor incorporating this one must include as components, directly or indirectly. The difference between declaring flavors as required and listing them directly as components at the top of the **defflavor** is that declaring flavors to be required does not make any commitments about where those flavors will appear in the ordered list of components; that is left up to whoever does specify them as components. The main thing that declaring a flavor as required accomplishes is to allow instance variables declared by that flavor to be accessed. It also provides error checking: an attempt to instantiate a flavor which does not include the required flavors as components will signal an error. Compare this with **:required-methods** and **:required-instance-variables**.

For an example of the use of required flavors, consider the **ship** example given earlier, and suppose we want to define a **relativity-mixin** which increases the mass dependent on the speed. We might write,

```
(defflavor relativity-mixin () (moving-object))
(defmethod (relativity-mixin :mass) ()
  (// mass (sqrt (- 1 (^ (// (funcall-self ':speed)
                             *speed-of-light*)
                         2)))))
```

but this would lose because any flavor that had **relativity-mixin** as a component would get **moving-object** right after it in its component list. As a base flavor, **moving-object** should be last in the list of components so that other components mixed in can replace its methods and so that daemon methods combine in the right order. **relativity-mixin** has no business changing the order in which flavors are combined, which should be under the control of its caller, for example:

```
(defflavor starship ()
           (relativity-mixin long-distance-mixin ship))
```
which puts moving-object last (inheriting it from ship).

So instead of the definition above we write,
```
(defflavor relativity-mixin () ()
        (:required-flavors moving-object))
```
which allows relativity-mixin's methods to access moving-object instance variables such as mass (the rest mass), but does not specify any place for moving-object in the list of components.

It is very common to specify the *base flavor* of a mixin with the :required-flavors option in this way.

:included-flavors
The arguments are names of flavors to be included in this flavor. The difference between declaring flavors here and declaring them at the top of the defflavor is that when component flavors are combined, if an included flavor is not specified as a normal component, it is inserted into the list of components immediately after the last component to include it. Thus included flavors act like defaults. The important thing is that if an included flavor *is* specified as a component, its position in the list of components is completely controlled by that specification, independently of where the flavor that includes it appears in the list.

:included-flavors and :required-flavors are used in similar ways; it would have been reasonable to use :included-flavors in the relativity-mixin example above. The difference is that when a flavor is required but not given as a normal component, an error is signalled, but when a flavor is included but not given as a normal component, it is automatically inserted into the list of components at a "reasonable" place.

:no-vanilla-flavor
Normally when a flavor is instantiated, the special flavor si:vanilla-flavor is included automatically at the end of its list of components. The vanilla flavor provides some default methods for the standard messages which all objects are supposed to understand. These include :print-self, :describe, :which-operations, and several other messages. See section 20.11, page 305.

If any component of a flavor specifies the :no-vanilla-flavor option, then si:vanilla-flavor will not be included in that flavor. This option should not be used casually.

:default-handler
The argument is the name of a function which is to be called when a message is received for which there is no method. It will be called with whatever arguments the instance was called with, including the message name; whatever values it returns will be returned. If this option is not specified on any component flavor, it defaults to a function which will signal an error.

:ordered-instance-variables
This option is mostly for esoteric internal system uses. The arguments are names of instance variables which must appear first (and in this order) in all instances of this flavor, or any flavor depending on this flavor. This is used for instance variables which are

specially known about by microcode, and in connection with the **:outside-accessible-instance-variables** option. If the keyword is given alone, the arguments default to the list of instance variables given at the top of this **defflavor**.

:outside-accessible-instance-variables

The arguments are instance variables which are to be accessible from "outside" of this object, that is from functions other than methods. A macro (actually a **defsubst**) is defined which takes an object of this flavor as an argument and returns the value of the instance variable; **setf** may be used to set the value of the instance variable. The name of the macro is the name of the flavor concatenated with a hyphen and the name of the instance variable. These macros are similar to the accessor macros created by **defstruct** (see chapter 19, page 257.)

This feature works in two different ways, depending on whether the instance variable has been declared to have a fixed slot in all instances, via the **:ordered-instance-variables** option.

If the variable is not ordered, the position of its value cell in the instance will have to be computed at run time. This takes noticeable time, although less than actually sending a message would take. An error will be signalled if the argument to the accessor macro is not an instance or is an instance which does not have an instance variable with the appropriate name. However, there is no error check that the flavor of the instance is the flavor the accessor macro was defined for, or a flavor built upon that flavor. This error check would be too expensive.

If the variable is ordered, the compiler will compile a call to the accessor macro into a subprimitive which simply accesses that variable's assigned slot by number. This subprimitive is only 3 or 4 times slower than **car**. The only error-checking performed is to make sure that the argument is really an instance and is really big enough to contain that slot. There is no check that the accessed slot really belongs to an instance variable of the appropriate name. Any functions that use these accessor macros will have to be recompiled if the number or order of instance variables in the flavor is changed. The system will not know automatically to do this recompilation. If you aren't very careful, you may forget to recompile something, and have a very hard-to-find bug. Because of this problem, and because using these macros is less elegant than sending messages, the use of this option is discouraged. In any case the use of these accessor macros should be confined to the module which owns the flavor, and the "general public" should send messages.

:accessor-prefix

Normally the accessor macro created by the **:outside-accessible-instance-variables** option to access the flavor *f*'s instance variable *v* is named *f-v*. Specifying **(:accessor-prefix get$)** would cause it to be named **get$***v* instead.

:select-method-order

This is purely an efficiency hack due to the fact that currently the method-table is searched linearly when a message is sent. The arguments are names of messages which are frequently used or for which speed is important. Their methods are moved to the front of the method table so that they are accessed more quickly.

:method-combination

Declares the way that methods from different flavors will be combined. Each "argument" to this option is a list (*type order message1 message2...*). *Message1*, *message2*, etc. are names of messages whose methods are to be combined in the declared fashion. *type* is a keyword which is a defined type of combination; see section 20.12, page 306. *Order* is a keyword whose interpretation is up to *type*; typically it is either :base-flavor-first or :base-flavor-last.

Any component of a flavor may specify the type of method combination to be used for a particular message. If no component specifies a type of method combination, then the default type is used, namely :daemon. If more than one component of a flavor specifies it, then they must agree on the specification, or else an error is signalled.

:documentation

The list of arguments to this option is remembered on the flavor's property list as the :documentation property. The (loose) standard for what can be in this list is as follows; this may be extended in the future. A string is documentation on what the flavor is for; this may consist of a brief overview in the first line, then several paragraphs of detailed documentation. A symbol is one of the following keywords:

:mixin	A flavor that you may want to mix with others to provide a useful feature.
:essential-mixin	A flavor that must be mixed in to all flavors of its class, or inappropriate behavior will ensue.
:lowlevel-mixin	A mixin used only to build other mixins.
:combination	A combination of flavors for a specific purpose.
:special-purpose	A flavor used for some internal or kludgey purpose by a particular program, which is not intended for general use.

This documentation can be viewed with the describe-flavor function (see page 299) or the editor's Meta-X Describe Flavor command (see page 311).

20.10 Flavor Families

The following organization conventions are recommended for all programs that use flavors.

A *base flavor* is a flavor that defines a whole family of related flavors, all of which will have that base flavor as one of their components. Typically the base flavor includes things relevant to the whole family, such as instance variables, :required-methods and :required-instance-variables declarations, default methods for certain messages, :method-combination declarations, and documentation on the general protocols and conventions of the family. Some base flavors are complete and can be instantiated, but most are not instantiatable and merely serve as a base upon which to build other flavors. The base flavor for the *foo* family is often named basic-*foo*.

A *mixin flavor* is a flavor that defines one particular feature of an object. A mixin cannot be instantiated, because it is not a complete description. Each module or feature of a program is defined as a separate mixin; a usable flavor can be constructed by choosing the mixins for the desired characteristics and combining them, along with the appropriate base flavor. By organizing your flavors this way, you keep separate features in separate flavors, and you can pick and choose among them. Sometimes the order of combining mixins does not matter, but often it does, because the order of flavor combination controls the order in which daemons are invoked and wrappers are wrapped. Such order dependencies would be documented as part of the conventions of the appropriate family of flavors. A mixin flavor that provides the *mumble* feature is often named *mumble*-**mixin**.

If you are writing a program that uses someone else's facility to do something, using that facility's flavors and methods, your program might still define its own flavors, in a simple way. The facility might provide a base flavor and a set of mixins, and the caller can combine these in various combinations depending on exactly what it wants, since the facility probably would not provide all possible useful combinations. Even if your private flavor has exactly the same components as a pre-existing flavor, it can still be useful since you can use its **:default-init-plist** (see page 300) to select options of its component flavors and you can define one or two methods to customize it "just a little".

20.11 Vanilla Flavor

The messages described in this section are a standard protocol which all message-receiving objects are assumed to understand. The standard methods that implement this protocol are automatically supplied by the flavor system unless the user specifically tells it not to do so. These methods are associated with the flavor **si:vanilla-flavor**:

si:vanilla-flavor *Flavor*

Unless you specify otherwise (with the **:no-vanilla-flavor** option to **defflavor**), every flavor includes the "vanilla" flavor, which has no instance variables but provides some basic useful methods.

:print-self *stream prindepth slashify-p*

The object should output its printed-representation to a stream. The printer sends this message when it encounters an instance or an entity. The arguments are the stream, the current depth in list-structure (for comparison with **prinlevel**), and whether slashification is enabled (**prin1** vs **princ**; see page 319). Vanilla-flavor ignores the last two arguments, and prints something like *#<flavor-name octal-address>*. The *flavor-name* tells you what type of object it is, and the *octal-address* allows you to tell different objects apart (provided the garbage collector doesn't move them behind your back).

:describe

The object should describe itself, printing a description onto the **standard-output** stream. The **describe** function sends this message when it encounters an instance or an entity. Vanilla-flavor outputs the object, the name of its flavor, and the names and values of its instance-variables, in a reasonable format.

:which-operations

The object should return a list of the messages it can handle. Vanilla-flavor generates the list once per flavor and remembers it, minimizing consing and compute-time. If a new method is added, the list is regenerated the next time someone asks for it.

:operation-handled-p *operation*

operation is a message name. The object should return t if it has a handler for the specified message, or nil if it does not.

:get-handler-for *operation*

operation is a message name. The object should return the method it uses to handle *operation*. If it has no handler for that message, it should return nil. This is like the get-handler-for function (see page 298), but, of course, you can only use it on objects known to accept messages.

:send-if-handles *operation* &rest *arguments*

operation is a message name and *arguments* is a list of arguments for that message. The object should send itself that message with those arguments, if it handles the message. If it doesn't handle the message it should just return nil.

:eval-inside-yourself *form*

The argument is a form which is evaluated in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. It works to setq one of these special variables; the instance variable will be modified. This is mainly intended to be used for debugging. An especially useful value of *form* is (break t); this gets you a Lisp top level loop inside the environment of the methods of the flavor, allowing you to examine and alter instance variables, and run functions that use the instance variables.

:funcall-inside-yourself *function* &rest *args*

function is applied to *args* in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. It works to setq one of these special variables; the instance variable will be modified. This is mainly intended to be used for debugging.

20.12 Method Combination

As was mentioned earlier, there are many ways to combine methods. The way we have seen is called the :daemon type of combination. To use one of the others, you use the :method-combination option to defflavor (see page 304) to say that all the methods for a certain message to this flavor, or a flavor built on it, should be combined in a certain way.

The following types of method combination are supplied by the system. It is possible to define your own types of method combination; for information on this, see the code. Note that for most types of method combination other than :daemon you must define the order in which the methods are combined, either :base-flavor-first or :base-flavor-last. In this context, base-flavor means the last element of the flavor's fully-expanded list of components.

Which method type keywords are allowed depends on the type of method combination selected. Many of them allow only untyped methods. There are also certain method types used for internal purposes.

:daemon	This is the default type of method combination. All the :before methods are called, then the primary (untyped) method for the outermost flavor that has one is called, then all the :after methods are called. The value returned is the value of the primary method.
:progn	All the methods are called, inside a progn special form. No typed methods are allowed. This means that all of the methods are called, and the result of the combined method is whatever the last of the methods returns.
:or	All the methods are called, inside an or special form. No typed methods are allowed. This means that each of the methods is called in turn. If a method returns a non-nil value, that value is returned and none of the rest of the methods are called; otherwise, the next method is called. In other words, each method is given a chance to handle the message; if it doesn't want to handle the message, it should return nil, and the next method will get a chance to try.
:and	All the methods are called, inside an and special form. No typed methods are allowed. The basic idea is much like :or; see above.
:list	Calls all the methods and returns a list of their returned values. No typed methods are allowed.
:inverse-list	Calls each method with one argument; these arguments are successive elements of the list which is the sole argument to the message. No typed methods are allowed. Returns no particular value. If the result of a :list-combined message is sent back with an :inverse-list-combined message, with the same ordering and with corresponding method definitions, each component flavor receives the value which came from that flavor.

Here is a table of all the method types used in the standard system (a user can add more, by defining new forms of method-combination).

(no type)	If no type is given to defmethod, a primary method is created. This is the most common type of method.
:before :after	These are used for the before-daemon and after-daemon methods used by :daemon method-combination.
:default	If there are no untyped methods among any of the flavors being combined, then the :default methods (if any) are treated as if they were untyped. If there are any untyped methods, the :default methods are ignored. Typically a base-flavor (see page 304) will define some default methods for certain of the messages understood by its family. When using the default kind of method-combination these default methods will not be called if a flavor provides its own method. But with certain strange forms of method-combination (:or for example) the base-flavor uses a :default method to achieve its desired effect.

:wrapper	Used internally by **defwrapper**.
:combined	Used internally for automatically-generated *combined* methods.

The most common form of combination is **:daemon**. One thing may not be clear: when do you use a **:before** daemon and when do you use an **:after** daemon? In some cases the primary method performs a clearly-defined action and the choice is obvious: **:before :launch-rocket** puts in the fuel, and **:after :launch-rocket** turns on the radar tracking.

In other cases the choice can be less obvious. Consider the **:init** message, which is sent to a newly-created object. To decide what kind of daemon to use, we observe the order in which daemon methods are called. First the **:before** daemon of the highest level of abstraction is called, then **:before** daemons of successively lower levels of abstraction are called, and finally the **:before** daemon (if any) of the base flavor is called. Then the primary method is called. After that, the **:after** daemon for the lowest level of abstraction is called, followed by the **:after** daemons at successively higher levels of abstraction.

Now, if there is no interaction among all these methods, if their actions are completely orthogonal, then it doesn't matter whether you use a **:before** daemon or an **:after** daemon. It makes a difference if there is some interaction. The interaction we are talking about is usually done through instance variables; in general, instance variables are how the methods of different component flavors communicate with each other. In the case of the **:init** message, the *init-plist* can be used as well. The important thing to remember is that no method knows beforehand which other flavors have been mixed in to form this flavor; a method cannot make any assumptions about how this flavor has been combined, and in what order the various components are mixed.

This means that when a **:before** daemon has run, it must assume that none of the methods for this message have run yet. But the **:after** daemon knows that the **:before** daemon for each of the other flavors has run. So if one flavor wants to convey information to the other, the first one should "transmit" the information in a **:before** daemon, and the second one should "receive" it in an **:after** daemon. So while the **:before** daemons are run, information is "transmitted"; that is, instance variables get set up. Then, when the **:after** daemons are run, they can look at the instance variables and act on their values.

In the case of the **:init** method, the **:before** daemons typically set up instance variables of the object based on the init-plist, while the **:after** daemons actually do things, relying on the fact that all of the instance variables have been initialized by the time they are called.

Of course, since flavors are not hierarchically organized, the notion of levels of abstraction is not strictly applicable. However, it remains a useful way of thinking about systems.

20.13 Implementation of Flavors

An object which is an instance of a flavor is implemented using the data type **dtp-instance**. The representation is a structure whose first word, tagged with **dtp-instance-header**, points to a structure (known to the microcode as an "instance descriptor") containing the internal data for the flavor. The remaining words of the structure are value cells containing the values of the instance variables. The instance descriptor is a **defstruct** which appears on the **si:flavor** property of the flavor name. It contains, among other things, the name of the flavor, the size of an instance, the table of methods for handling messages, and information for accessing the instance variables.

defflavor creates such a data structure for each flavor, and links them together according to the dependency relationships between flavors.

A message is sent to an instance simply by calling it as a function, with the first argument being the message keyword. The microcode binds **self** to the object, binds the instance variables (as **special** closure variables) to the value cells in the instance, and calls a **dtp-select-method** associated with the flavor. This **dtp-select-method** associates the message keyword to the actual function to be called. If there is only one method, this is that method, otherwise it is an automatically-generated function, called the combined method (see page 291), which calls the appropriate methods in the right order. If there are wrappers, they are incorporated into this combined method.

The function-specifier syntax (**:method** *flavor-name optional-method-type message-name*) is understood by **fdefine** and related functions. It is preferable to refer to methods this way rather than by explicit use of the flavor-method-symbol (see page 294).

20.13.1 Order of Definition

There is a certain amount of freedom to the order in which you do **defflavor**'s, **defmethod**'s, and **defwrapper**'s. This freedom is designed to make it easy to load programs containing complex flavor structures without having to do things in a certain order. It is considered important that not all the methods for a flavor need be defined in the same file. Thus the partitioning of a program into files can be along modular lines.

The rules for the order of definition are as follows.

Before a method can be defined (with **defmethod** or **defwrapper**) its flavor must have been defined (with **defflavor**). This makes sense because the system has to have a place to remember the method, and because it has to know the instance-variables of the flavor if the method is to be compiled.

When a flavor is defined (with **defflavor**) it is not necessary that all of its component flavors be defined already. This is to allow **defflavor**'s to be spread between files according to the modularity of a program, and to provide for mutually-dependent flavors. Methods can be defined for a flavor some of whose component flavors are not yet defined; however, in certain cases compiling those methods will produce a spurious warning that an instance variable was declared special (because the system did not realize it was an instance variable). In the current implementation these warnings may be ignored, although that may not always be true in the future.

The methods automatically generated by the :gettable-instance-variables and :settable-instance-variables defflavor options (see page 300) are generated at the time the defflavor is done.

The first time a flavor is instantiated, the system looks through all of the component flavors and gathers various information. At this point an error will be signalled if not all of the components have been defflavor'ed. This is also the time at which certain other errors are detected, for instance lack of a required instance-variable (see the :required-instance-variables defflavor option, page 300). The combined methods (see page 291) are generated at this time also, unless they already exist. They will already exist if compile-flavor-methods was used, but if those methods are obsolete because of changes made to component flavors since the compilation, new combined methods will be made.

After a flavor has been instantiated, it is possible to make changes to it. These changes will affect all existing instances if possible. This is described more fully immediately below.

20.13.2 Changing a Flavor

You can change anything about a flavor at any time. You can change the flavor's general attributes by doing another defflavor with the same name. You can add or modify methods by doing defmethod's. If you do a defmethod with the same flavor-name, message-name, and (optional) method-type as an existing method, that method is replaced with the new definition. You can remove a method with undefmethod (see page 297).

These changes will always propagate to all flavors that depend upon the changed flavor. Normally the system will propagate the changes to all existing instances of the changed flavor and all flavors that depend on it. However, this is not possible when the flavor has been changed so drastically that the old instances would not work properly with the new flavor. This happens if you change the number of instance variables, which changes the size of an instance. It also happens if you change the order of the instance variables (and hence the storage layout of an instance), or if you change the component flavors (which can change several subtle aspects of an instance). The system does not keep a list of all the instances of each flavor, so it cannot find the instances and modify them to conform to the new flavor definition. Instead it gives you a warning message, on the error-output stream, to the effect that the flavor was changed incompatibly and the old instances will not get the new version. The system leaves the old flavor data-structure intact (the old instances will continue to point at it) and makes a new one to contain the new version of the flavor. If a less drastic change is made, the system modifies the original flavor data-structure, thus affecting the old instances that point at it. However, if you redefine methods in such a way that they only work for the new version of the flavor, then trying to use those methods with the old instances won't work.

One exception to this is that changes to defwrapper's are never automatically propagated. This is because doing so is expensive and the system cannot tell whether you really changed it or just redefined it to be the same as it was. (Note that the initial definition of a wrapper *is* propagated, but redefinitions of it are not.) See the documentation of defwrapper for more details.

20.13.3 Restrictions

There is presently an implementation restriction that when using daemons, the primary method may return at most three values if there are any :after daemons. This is because the combined method needs a place to remember the values while it calls the daemons. This will be fixed some day.

In this implementation, all message names must be in the keyword package, in order for the flavor-method-symbols (see page 294) to be unique, and for various tools in the editor to work correctly.

20.14 Entities

An *entity* is a Lisp object; the entity is one of the primitive datatypes provided by the Lisp Machine system (the data-type function (see page 173) will return dtp-entity if it is given an entity). Entities are just like closures: they have all the same attributes and functionality. The only difference between the two primitive types is their data type: entities are clearly distinguished from closures because they have a different data type. The reason there is an important difference between them is that various parts of the (not so primitive) Lisp system treat them differently. The Lisp functions that deal with entities are discussed in section 11.4, page 162.

A closure is simply a kind of function, but an entity is assumed to be a message-receiving object. Thus, when the Lisp printer (see section 21.2.1, page 319) is given a closure, it prints a simple textual representation, but when it is handed an entity, it sends the entity a :print-self message, which the entity is expected to handle. The describe function (see page 500) also sends entities messages when it is handed them. So when you want to make a message-receiving object out of a closure, as described on page 285, you should use an entity instead.

Usually there is no point in using entities instead of flavors. Entities were introduced into Zetalisp before flavors were, and perhaps they would not have been had flavors already existed. Flavors have had considerably more attention paid to efficiency and to good tools for using them.

Entities are created with the entity function (see page 162). The function part of an entity should usually be a function created by defselect (see page 147).

20.15 Useful Editor Commands

Since we presently lack an editor manual, this section briefly documents some editor commands that are useful in conjunction with flavors.

meta-.

The meta-. (Edit Definition) command can find the definition of a flavor in the same way that it can find the definition of a function.

Edit Definition can find the definition of a method if you give

```
(:method flavor type message)
```

as the function name. The keyword :method may be omitted. Completion will occur on the flavor name and message name as usual with Edit Definition.

meta-X Describe Flavor
: Asks for a flavor name in the mini-buffer and describes its characteristics. When typing the flavor name you have completion over the names of all defined flavors (thus this command can be used to aid in guessing the name of a flavor). The display produced is mouse sensitive where there are names of flavors and of methods; as usual the right-hand mouse button gives you a menu of operations and the left-hand mouse button does the most common operation, typically positioning the editor to the source code for the thing you are pointing at.

meta-X List Methods
meta-X Edit Methods
: Asks you for a message in the mini-buffer and lists all the flavors which have a method for that message. You may type in the message name, point to it with the mouse, or let it default to the message which is being sent by the Lisp form the cursor is inside of. List Methods produces a mouse-sensitive display allowing you to edit selected methods or just see which flavors have methods, while Edit Methods skips the display and proceeds directly to editing the methods. As usual with this type of command, the editor command control-. is redefined to advance the editor cursor to the next method in the list, reading in its source file if necessary. Typing control-. while the display is on the screen edits the first method.

meta-X List Combined Methods
meta-X Edit Combined Methods
: Asks you for a message and a flavor in two mini-buffers and lists all the methods which would be called if that message were sent to an instance of that flavor. You may point to the message and flavor with the mouse, and there is completion for the flavor name. As in List/Edit Methods, the display is mouse sensitive and the Edit version of the command skips the display and proceeds directly to the editing phase.

: List Combined Methods can be very useful for telling what a flavor will do in response to a message. It shows you the primary method, the daemons, and the wrappers and lets you see the code for all of them; type control-. to get to successive ones.

20.16 Property List Messages

It is often useful to associate a property list with an abstract object, for the same reasons that it is useful to have a property list associated with a symbol. This section describes a mixin flavor that can be used as a component of any new flavor in order to provide that new flavor with a property list. For more details and examples, see the general discussion of property lists (section 5.8, page 71). [Currently, the functions get, putprop, etc., do not accept flavor instances as arguments and send the corresponding message; this will be fixed.]

si:property-list-mixin *Flavor*
: This mixin flavor provides the basic operations on property lists.

:get *indicator* (to si:property-list-mixin)
The :get message looks up the object's *indicator* property. If it finds such a property, it returns the value; otherwise it returns nil.

:getl *indicator-list* (to si:property-list-mixin)
The :getl message is like the :get message, except that the argument is a list of indicators. The :getl message searches down the property list for any of the indicators in *indicator-list*, until it finds a property whose indicator is one of those elements. It returns the portion of the property list begining with the first such property that it found. If it doesn't find any, it returns nil.

:putprop *property indicator* (to si:property-list-mixin)
This gives the object an *indicator*-property of *property*.

:remprop *indicator* (to si:property-list-mixin)
This removes the object's *indicator* property, by splicing it out of the property list. It returns that portion of the list inside the object of which the former *indicator*-property was the car.

:push-property *value indicator* (to si:property-list-mixin)
The *indicator*-property of the object should be a list (note that nil is a list and an absent property is nil). This message sets the *indicator*-property of the object to a list whose car is *value* and whose cdr is the former *indicator*-property of the list. This is analogous to doing

```
(push value (get object indicator))
```

See the push special form (page 231).

:property-list (to si:property-list-mixin)
This returns the list of alternating indicators and values that implements the property list.

:set-property-list *list* (to si:property-list-mixin)
This sets the list of alternating indicators and values that implements the property list to *list*.

:property-list *list* (Init Option for si:property-list-mixin)
This initializes the list of alternating indicators and values that implements the property list to *list*.

21. The I/O System

Zetalisp provides a powerful and flexible system for performing input and output to peripheral devices. To allow device independent I/O (that is, to allow programs to be written in a general way so that the program's input and output may be connected with any device), the Zetalisp I/O system provides the concept of an "I/O stream". What streams are, the way they work, and the functions to create and manipulate streams, are described in this chapter. This chapter also describes the Lisp "I/O" operations read and print, and the printed representation they use for Lisp objects.

21.1 The Character Set

Zetalisp represents characters as fixnums. The Lisp Machine's mapping between these numbers and the characters is listed here. The mapping is similar to ASCII, but somewhat modified to allow the use of the so-called SAIL extended graphics, while avoiding certain ambiguities present in ITS. For a long time ITS treated the Backspace, Control-H, and Lambda keys on the keyboard identically as character code 10 octal; this problem is avoided from the start in the Lisp Machine's mapping.

It is worth pointing out that although the Zetalisp character set is different from the pdp-10 character set, when files are transferred between Lisp Machines and pdp-10's the characters are automatically converted. Details of the mapping are explained below.

Fundamental characters are eight bits wide. Those less than 200 octal (with the 200 bit off) and only those are printing graphics; when output to a device they are assumed to print a character and move the "cursor" one character position to the right. (All software provides for variable-width fonts, so the term "character position" shouldn't be taken too literally.)

Characters in the range of 200 to 236 inclusive are used for special characters. Character 200 is a "null character", which does not correspond to any key on the keyboard. The null character is not used for anything much; fasload uses it internally. Characters 201 through 236 correspond to the special function keys on the keyboard such as Return and Call. The remaining characters are reserved for future expansion.

It should never be necessary for a user or a source program to know these numerical values. Indeed, they are likely to be changed in the future. There are symbolic names for all characters; see below.

Most of the special characters do not normally appear in files (although it is not forbidden for files to contain them). These characters exist mainly to be used as "commands" from the keyboard.

A few special characters, however, are "format effectors" which are just as legitimate as printing characters in text files. The names and meanings of these characters are:

Return	The "carriage return" character which separates lines of text. Note that the pdp-10 convention that lines are ended by a pair of characters, "carriage return" and "line feed", is not used.

Page The "page separator" character which separates pages of text.

Tab The "tabulation" character which spaces to the right until the next "tab stop". Tab stops are normally every 8 character positions.

The space character is considered to be a printing character whose printed image happens to be blank, rather than a format effector.

In some contexts, a fixnum can hold both a character code and a font number for that character. The following byte specifiers are defined:

%%ch-char *Variable*
The value of %%ch-char is a byte specifier for the field of a fixnum character which holds the character code.

%%ch-font *Variable*
The value of %%ch-font is a byte specifier for the field of a fixnum character which holds the font number.

Characters read in from the keyboard include a character code and control bits. A character cannot contain both a font number and control bits, since these data are both stored in the same bits. The following byte specifiers are provided:

%%kbd-char *Variable*
The value of %%kbd-char is a byte specifier for the field of a keyboard character which holds the normal eight-bit character code.

%%kbd-control *Variable*
The value of %%kbd-control is a byte specifier for the field of a keyboard character which is 1 if either Control key was held down.

%%kbd-meta *Variable*
The value of %%kbd-meta is a byte specifier for the field of a keyboard character which is 1 if either Meta key was held down.

%%kbd-super *Variable*
The value of %%kbd-super is a byte specifier for the field of a keyboard character which is 1 if either Super key was held down.

%%kbd-hyper *Variable*
The value of %%kbd-hyper is a byte specifier for the field of a keyboard character which is 1 if either Hyper key was held down.

This bit is also set if Control and/or Meta is typed in combination with Shift and a letter. Shift is much easier than Hyper to reach with the left hand.

%%kbd-control-meta *Variable*
The value of %%kbd-control-meta is a byte specifier for the four-bit field of a keyboard character which contains the above control bits. The least-significant bit is Control. The most significant bit is Hyper.

The following fields are used by some programs that encode signals from the mouse in a the format of a character. Refer to the window system documentation for an explanation of how these characters are generated.

%%kbd-mouse *Variable*
The value of %%kbd-mouse is a byte specifier for the bit in a keyboard character which indicates that the character is not really a character, but a signal from the mouse.

%%kbd-mouse-button *Variable*
The value of %%kbd-mouse-button is a byte specifier for the field in a mouse signal which says which button was clicked. The value is 0, 1, or 2 for the left, middle, or right button, respectively.

%%kbd-mouse-n-clicks *Variable*
The value of %%kbd-mouse-n-clicks is a byte specifier for the field in a mouse signal which says how many times the button was clicked. The value is one less than the number of times the button was clicked.

When any of the control bits (Control, Meta, Super, or Hyper) is set in conjunction with a letter, the letter will always be upper-case. The character codes which consist of a lower-case letter and non-zero control bits are "holes" in the character set which are never used for anything. Note that when Shift is typed in conjuction with Control and/or Meta and a letter, it means Hyper rather than Shift.

Since the control bits are not part of the fundamental 8-bit character codes, there is no way to express keyboard input in terms of simple character codes. However, there is a convention which the relevant programs accept for encoding keyboard input into a string of characters: if a character has its Control bit on, prefix it with an Alpha. If a character has its Meta bit on, prefix it with a Beta. If a character has both its Control and Meta bits on, prefix it with an Epsilon. If a character has its Super bit on, prefix it with a Pi. If a character has its Hyper bit on, prefix it with a Lambda. To get an Alpha, Beta, Epsilon, Pi, Lambda, or Equivalence into the string, quote it by prefixing it with an Equivalence.

When characters are written to a file server computer that normally uses the ASCII character set to store text, Lisp Machine characters are mapped into an encoding that is reasonably close to an ASCII transliteration of the text. When a file is written, the characters are converted into this encoding, and the inverse transformation is done when a file is read back. No information is lost. Note that the length of a file, in characters, will not be the same measured in original Lisp Machine characters as it will measured in the encoded ASCII characters. In the currently implemented ASCII file servers, the following encoding is used. All printing characters and any characters not mentioned explicitly here are represented as themselves. Codes 010 (lambda), 011 (gamma), 012 (delta), 014 (plus-minus), 015 (circle-plus), 177 (integral), 200 through 207 inclusive, 213 (delete/vt), and 216 and anything higher, are preceded by a 177; that is, 177 is used as a "quoting character" for these codes. Codes 210 (overstrike), 211 (tab), 212 (line), and

214 (page), are converted to their ASCII cognates, namely 010 (backspace), 011 (horizontal tab), 012 (line feed), and 014 (form feed) respectively. Code 215 (return) is converted into 015 (carriage return) followed by 012 (line feed). Code 377 is ignored completely, and so cannot be stored in files.

000	center-dot (·)	040	space	100	@	140	`
001	down arrow (↓)	041	!	101	A	141	a
002	alpha (α)	042	"	102	B	142	b
003	beta (β)	043	#	103	C	143	c
004	and-sign (∧)	044	$	104	D	144	d
005	not-sign (¬)	045	%	105	E	145	e
006	epsilon (ε)	046	&	106	F	146	f
007	pi (π)	047	'	107	G	147	g
010	lambda (λ)	050	(	110	H	150	h
011	gamma (γ)	051	)	111	I	151	i
012	delta (δ)	052	*	112	J	152	j
013	uparrow (↑)	053	+	113	K	153	k
014	plus-minus (±)	054	,	114	L	154	l
015	circle-plus (⊕)	055	-	115	M	155	m
016	infinity (∞)	056	.	116	N	156	n
017	partial delta (∂)	057	/	117	O	157	o
020	left horseshoe (⊂)	060	0	120	P	160	p
021	right horseshoe (⊃)	061	1	121	Q	161	q
022	up horseshoe (∩)	062	2	122	R	162	r
023	down horseshoe (∪)	063	3	123	S	163	s
024	universal quantifier (∀)	064	4	124	T	164	t
025	existential quantifier (∃)	065	5	125	U	165	u
026	circle-X (⊗)	066	6	126	V	166	v
027	double-arrow (↔)	067	7	127	W	167	w
030	left arrow (←)	070	8	130	X	170	x
031	right arrow (→)	071	9	131	Y	171	y
032	not-equals (≠)	072	:	132	Z	172	z
033	diamond (altmode) (◊)	073	;	133	[	173	{
034	less-or-equal (≤)	074	<	134	\	174	|
035	greater-or-equal (≥)	075	=	135	]	175	}
036	equivalence (≡)	076	>	136	^	176	~
037	or (∨)	077	?	137	_	177	∫
200	null character	210	overstrike	220	stop-output	230	roman-iv
201	break	211	tab	221	abort	231	hand-up
202	clear	212	line	222	resume	232	hand-down
203	call	213	delete/vt	223	status	233	hand-left
204	terminal escape	214	page	224	end	234	hand-right
205	macro/backnext	215	return	225	roman-i	235	system
206	help	216	quote	226	roman-ii	236	network
207	rubout	217	hold-output	227	roman-iii		

237-377 reserved for the future

The Lisp Machine Character Set

21.2 Printed Representation

People cannot deal directly with Lisp objects, because the objects live inside the machine. In order to let us get at and talk about Lisp objects, Lisp provides a representation of objects in the form of printed text; this is called the *printed representation*. This is what you have been seeing in the examples throughout this manual. Functions such as **print**, **prin1**, and **princ** take a Lisp object, and send the characters of its printed representation to a stream. These functions (and the internal functions they call) are known as the *printer*. The **read** function takes characters from a stream, interprets them as a printed representation of a Lisp object, builds a corresponding object and returns it; it and its subfunctions are known as the *reader*. (Streams are explained in section 21.5.1, page 338.)

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.".

21.2.1 What the Printer Produces

The printed representation of an object depends on its type. In this section, we will consider each type of object and explain how it is printed.

Printing is done either with or without *slashification*. The non-slashified version is nicer looking in general, but if you give it to **read** it won't do the right thing. The slashified version is carefully set up so that **read** will be able to read it in. The primary effects of slashification are that special characters used with other than their normal meanings (e.g. a parenthesis appearing in the name of a symbol) are preceeded by slashes or cause the name of the symbol to be enclosed in vertical bars, and that symbols which are not from the current package get printed out with their package prefixes (a package prefix looks like a 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 which 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".

For a symbol: if slashification is off, the p.r. is simply the successive characters of the print-name of the symbol. If slashification is on, two changes must be made. First, the symbol might require a package prefix in order that **read** work correctly, assuming that the package into which **read** will read the symbol is the one in which it is being printed. See the section on packages (chapter 23, page 392) for an explanation of the package name prefix. Secondly, if the p.r. would not read in as a symbol at all (that is, if the print-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 (see section 21.2.6, page 328), so it is always accurate provided that **readtable** has the same value when the output is read back in as when it was printed.

For a string: if slashification is off, the p.r. is simply the successive characters of the string. If slashification is on, the string is printed between double quotes, and any characters inside the string which need to be 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 (see below), and whether slashification is enabled. The object should print a suitable p.r. on the stream. See chapter 20, page 279 for documentation on instances. Most such objects print like "any other data type" below, except with additional information such as a name. Some objects print only their name when slashification is not in effect (when **princ**'ed).

For an array which is a named structure: if the array has a named structure symbol with a **named-structure-invoke** property which 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 (see below), and whether slashification is enabled. A suitable printed representation should be sent to the stream. This allows a user to define his own p.r. for his named structures; more information can be found in the named structure section (see page 271). 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 which is more concise and suppresses detail. Of course, neither the ellipsis nor the "**" can be interpreted by read, since the relevant information is lost.

prinlevel *Variable*

prinlevel can be set to the maximum number of nested lists that can be printed before the printer will give up and just print a "**". If it is nil, which it is initially, any number of nested lists can be printed. Otherwise, the value of prinlevel must be a fixnum.

prinlength *Variable*

prinlength can be set to the maximum number of elements of a list that will be printed before the printer will give up and print a "...". If it is nil, which it is initially, any length list may be printed. Otherwise, the value of prinlength must be a fixnum.

For any other data type: the p.r. starts with a number sign and a less-than sign ("<"), the "dtp-" symbol for this datatype, a space, and the 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. Just what read accepts is the topic of the next section.

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 which is a named structure (see page 271), or an instance of a flavor (see chapter 20, page 279). However, occasionally it is desirable to get control over all printing of objects, in order to change, in some way, how they are printed. If you need to do this, the best way to proceed is to customize the behavior of si:print-object (see page 365), which is the main internal function of the printer. All of the printing functions, such as print and princ, as well as format, go through this function. The way to customize it is by using the "advice" facility (see section 26.4, page 460).

21.2.2 What The Reader Accepts

The purpose of the reader is to accept characters, interpret them as the p.r. of a Lisp object, and return a corresponding Lisp object. The reader cannot accept everything that the printer produces; for example, the p.r.'s of arrays (other than strings), compiled code objects, closures, stack groups etc. cannot be read in. However, it has many features which are not seen in the printer at all, such as more flexibility, comments, and convenient abbreviations for frequently-used unwieldy constructs.

This section shows what kind of p.r.'s the reader understands, and explains the readtable, reader macros, and various features provided by read.

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 preceeded by a plus sign or a minus sign, and optionally followed by a trailing decimal point. A simple fixnum will be interpreted by read as a fixnum. If the trailing decimal point is present, the digits will be interpreted in decimal radix; otherwise, they will be considered as a number whose radix is the value of the variable ibase.

ibase *Variable*

The value of ibase is a number which is the radix in which fixnums are read. The initial value of ibase is 8.

read will also understand a simple fixnum, followed by an underscore ("_") or a circumflex ("^"), followed by another simple fixnum. The two simple fixnums will be interpreted in the usual way, then the character in between indicates an operation to be performed on the two fixnums. The underscore indicates a binary "left shift"; that is, the fixnum to its left is doubled the number of times indicated by the fixnum to its right. The circumflex multiplies the fixnum to its left by ibase the number of times indicated by the fixnum to its right. (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. 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.) 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
```

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; lower-case letters are translated to upper-case. When the reader sees the p.r. of a symbol, it *interns* it on a *package* (see chapter 23, page 392 for an explanation of interning and the package system). Symbols may start with digits; you could even have one named "-345T"; **read** will accept this as a symbol without complaint. If you want to put strange characters (such as lower-case letters, parentheses, or reader macro characters) inside the name of a symbol, put a slash before 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 svmbols:
```
foo
bar/(baz/)
34w23
|Frob Sale|
```

The reader will also recognize strings, which should be surrounded by double-quotes. If you want to put a double-quote or a slash inside a string, preceed it by a slash.
Examples of strings:
```
"This is a typical string."
"That is known as a /"cons cell/" in Lisp."
```

When read sees an open parenthesis, it knows that the p.r. of a cons is coming, and calls itself recursively to get the elements of the cons or the list that follows. Any of the following are valid:
```
(foo . bar)
(foo bar baz)
(foo . (bar . (baz . nil)))
(foo bar . quux)
```
The first is a cons, whose car and cdr are both symbols. The second is a list, and the third is exactly the same as the second (although print would never produce it). The fourth is a "dotted list"; the cdr of the last cons cell (the second one) is not nil, but quux.

Whenever the reader sees any of the above, it creates new cons cells; it never returns existing list structure. This contrasts with the case for symbols, as very often read returns symbols that it found 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.

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.

21.2.3 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 section 17.2.2, page 211 for details.

Comma (,) is part of the syntax of backquote and is invalid if used other than inside the body of a backquote. See section 17.2.2, page 211 for details.

Sharp sign (#) introduces a number of other syntax extensions. See the following section. 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 (see page 329) can be used to define your own macro characters.

21.2.4 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.

#/ #/*x* reads in as the number which is the character code for the character *x*. For example, #/a is equivalent to 141 but clearer in its intent. This is the recommended way to include character constants in your code. Note that the slash causes this construct to be parsed correctly by the editors, Emacs and Zwei.

As in strings, upper and lower-case letters are distinguished after #/. Any character works after #/, even those that are normally special to read, such as parentheses. Even non-printing characters may be used, although for them #\ is preferred.

The character can 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; it is superseded by the \# syntax (see below). 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 lower-case alphabetic character, the character code for the upper-case version of the character is produced.

#\ #*name* reads in as the number which is the character code for the non-printing character symbolized by *name*. A large number of character names are recognized; these are documented below (section 21.2.5, page 328). 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 lower-case 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.

#\ 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 lower-case letters are not distinguished. Also, control may be spelled ctrl as it is on the keyboards. 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 lower-case character or a special character such as a parenthesis, you must preceed it by 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 special Greek characters as with #/, but this is less clear than spelling them out, and should be avoided in new programs.

#^ #^*x* is exactly like *#α/x* if the input is being read by Zetalisp; it generates Control-*x*. In Maclisp *x* is converted to upper case and then exclusive-or'ed with 100 (octal). Thus #^*x* always generates the character returned by tyi if the user holds down the control key and types *x*. (In Maclisp *#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 which 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.

#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.
[Unfortunately #X does not completely work, currently, since it does not cause the letters A through F to be recognized as numbers. This does not seem to have bothered anyone yet.]

#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.

For example, #3R102 is another way of writing 11. and #11R32 is another way of writing 35. Bases larger than ten do not work completely, since there are only ten digit characters.

#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 (white space). 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 page 508.

#- #-*feature form* is equivalent to #+(not *feature*) *form*.

#< This is not legal reader syntax. It is used in the p.r. of objects which cannot be read back in. Attempting to read a #< will cause an error.

The function set-syntax-#-macro-char (see page 330) can be used to define your own sharp sign abbreviations.

21.2.5 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 which also have special names because they may be hard to type on a pdp-10.

altmode	circle-plus	delta	gamma
integral	lambda	plus-minus	uparrow

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-l-2 = mouse-1-2
mouse-m-1 = mouse-2-1	mouse-m-2 = mouse-2-2
mouse-r-1 = mouse-3-1	mouse-r-2 = mouse-3-2

21.2.6 The Readtable

There is a data structure called the *readtable* which is used to control the reader. It contains information about the syntax of each character. Initially it is set up to give the standard Lisp meanings to all the characters, but the user can change 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 the initial standard 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 not ever change the contents of this readtable; only examine it, by using it as the *from-readtable* argument to copy-readtable or set-syntax-from-char.

The user can program the reader by changing the readtable in any of three ways. The syntax of a character can be set to one of several predefined possibilities. A character can be made into a *macro character*, whose interpretation is controlled by a user-supplied function which is called when the character is read. The user can create a completely new readtable, using the readtable compiler (LMIO;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*

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*

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*

Changes *readtable* so that *from-char* will be translated to *to-char* upon read-in, when *readtable* is the current readtable. This is normally used only for translating lower case letters to upper case. Character 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*

Causes *char* to be a macro character which when read calls *function*. *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 rubout-handler works, *function* can be called several times during the reading of a single expression in which the macro character only appears once.

char is given the same syntax that single-quote, backquote, and comma have in the initial readtable (it is called :macro syntax).

set-syntax-#-macro-char *char function* &optional *readtable*
Causes *function* to be called when *#char* is read. *readtable* defaults to the current readtable. The function's arguments and return values are the same as for normal macro characters, documented above. When *function* is called, the special variable **si:xr-sharp-argument** contains nil or a number which is the number or special bits between the *#* and *char*.

set-syntax-from-description *char description* &optional *readtable*
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 which 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. Don't use this, use set-syntax-macro-char.
si:circlecross	The octal escape for special characters. In the initial readtable this is "⊗".

These symbols will probably be moved to the standard keyword package at some point. *readtable* defaults to the current readtable.

setsyntax *character arg2 arg3*
This exists only for Maclisp compatibility. The above functions are preferred in new programs. The syntax of *character* is altered in the current readtable, according to *arg2* and *arg3*. *character* can be a fixnum, a symbol, or a string, i.e. anything acceptable to the character function. *arg2* is usually a keyword; it can be in any package since this is a Maclisp compatibility function. The following values are allowed for *arg2*:

:macro	The character becomes a macro character. *arg3* is the name of a function to be invoked when this character is read. The function takes no arguments, may tyi or read from standard-input (i.e. may call tyi or read without specifying a stream), and returns an object which is taken as the result of the read.

:splicing	Like :macro but the object returned by the macro function is a list which is nconced 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*

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 which 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*.

21.3 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 pseudo-streams nil and t are also accepted, mainly for Maclisp compatibility. nil means the value of standard-input (i.e. the default) and t means the value of terminal-io (i.e. 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. For detailed documentation of streams, refer to section 21.5.1, page 338.

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 will be 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 which read an "object" rather than a single character will 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 will complain. If a file ends in a symbol or a number immediately followed by end-of-file, read will read the symbol or number successfully and when called again will see the end-of-file and obey *eof-option*. If a file contains ignorable text at the end, such as blank lines and comments, read will not consider it to end in the middle of an object and will obey *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 will 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 will echo their input if used on an interactive stream (one which supports the :rubout-handler operation; see below.) The functions that input more than one character at a time (read, readline) allow the input to be edited using rubout. tyipeek echoes all of the characters that were skipped over if tyi would have echoed them; the character not removed from the stream is not echoed either.

read &optional *stream eof-option*

read reads in the printed representation of a Lisp object from *stream*, builds a corresponding Lisp object, and returns the object. The details have been explained above. (This function can take its arguments in the other order, for Maclisp compatibility only; see the note above.)

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 will throw away the delimiting character if it is "whitespace", but will preserve 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 will be thrown away, even if they are whitespace. This may be useful for certain reader macros or special syntaxes.

tyi &optional *stream eof-option*

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, etc. shifts echo as prefix alpha, beta, etc.

The :tyi stream operation is preferred over the tyi function for some purposes. Note that it does not echo. See section 21.5.2, page 338.

(This function can take its arguments in the other order, for Maclisp compatibility only; see the note above.)

read-for-top-level &optional *stream eof-option*

This is a slightly different version of **read**. It differs from **read** only in that it ignores close parentheses seen at top level, and it returns the symbol **si:eof** if the stream reaches end-of-file if you have not supplied an *eof-option* (instead of signalling an error as **read** would). This version of **read** is used in the system's "read-eval-print" loops.

(This function can take its arguments in the other order, for uniformity with **read** only; see the note above.)

readline &optional *stream eof-option options*

readline reads in a line of text, terminated by a return. It returns the line as a character string, *without* the return character. This function is usually used to get a line of input from the user. If rubout processing is happening, then *options* is passed as the list of options to the rubout handler. One option that is particularly useful is the **:do-not-echo** option (see page 364), which you can use to make the **return** character that terminates the line not be echoed. (This function can take its arguments in the other order, for Maclisp compatibility only; see the note above.)

readch &optional *stream eof-option*

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*

This function is provided mainly for Maclisp compatibility; the **:listen** stream operation is usually clearer (see page 339).

What **tyipeek** does depends on the *peek-type*, which defaults to **nil**. With a *peek-type* of **nil**, **tyipeek** returns the next character to be read from *stream*, without actually removing it from the input stream. The next time input is done from *stream* the character will still be there; in general, **(= (tyipeek) (tyi))** is **t**.

If *peek-type* is a fixnum less than 1000 octal, then **tyipeek** reads characters from *stream* until it gets one equal to *peek-type*. That character is not removed from the input stream.

If *peek-type* is **t**, then **tyipeek** skips over input characters until the start of the printed representation of a Lisp object is reached. As above, the last character (the one that starts an object) is not removed from the input stream.

The form of **tyipeek** supported by Maclisp in which *peek-type* is a fixnum not less than 1000 octal is not supported, since the readtable formats of the Maclisp reader and the Zetalisp reader are quite different.

Characters passed over by **tyipeek** are echoed if *stream* is interactive.

The following functions are related functions which 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* (*idx 0*)

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. *idx* is the index in the string of the first 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 will be either the length of the string or 1 more than the length of the string.

Example:
```
        (read-from-string "(a b c)") => (a b c) and 7
```

readlist *char-list*

This function is provided mainly for Maclisp compatibility. *char-list* is a list of characters. The characters may be represented by anything that the function character accepts: 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 also the with-input-from-string special form (page 133).

21.4 Output Functions

These functions all take an optional argument called *stream*, which is where to send the output. If unsupplied *stream* defaults to the value of standard-output. If *stream* is nil, the value of standard-output (i.e. the default) is used. If it is t, the value of terminal-io is used (i.e. the interactive terminal). 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. For detailed documentation of streams, refer to section 21.5.1, page 338.

prin1 *x* &optional *stream*

prin1 outputs the printed representation of *x* to *stream*, with slashification (see page 319). *x* is returned.

prin1-then-space *x* &optional *stream*

prin1-then-space is like prin1 except that output is followed by a space.

print *x* &optional *stream*

print is just like prin1 except that output is preceeded by a carriage return and followed by a space. *x* is returned.

princ *x* &optional *stream*

princ is just like prin1 except that the output is not slashified. *x* is returned.

tyo *char* &optional *stream*

tyo outputs the character *char* to *stream*.

terpri &optional *stream*

terpri outputs a carriage return character to stream.

The format function (see page 346) is very useful for producing nicely formatted text. It can do anything any of the above functions can do, and it makes it easy to produce good looking messages and such. format can generate a string or output to a stream.

The grindef function (see page 360) is useful for formatting Lisp programs.

See also the with-output-to-string special form (page 134).

stream-copy-until-eof *from-stream* *to-stream* &optional *leader-size*

stream-copy-until-eof inputs characters from *from-stream* and outputs them to *to-stream*, until it reaches the end-of-file on the *from-stream*. For example, if x is bound to a stream for a file opened for input, then (stream-copy-until-eof x terminal-io) will print the file on the console.

If *from-stream* supports the :line-in operation and *to-stream* supports the :line-out operation, then stream-copy-until-eof will use those operations instead of :tyi and :tyo, for greater efficiency. *leader-size* will be passed as the argument to the :line-in operation.

beep &optional *beep-type* (*stream* terminal-io)

This function is intended to attract the user's attention by causing an audible beep, or flashing the screen, or something similar. If the stream supports the :beep operation, then this function sends it a :beep message, passing *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. (The :beep message is described on page 341.)

cursorpos &rest *args*

This function exists primarily for Maclisp compatibility. Usually it is preferable to send the appropriate messages (see the window system documentation).

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 which are capable of these operations, for instance 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 doesn't.

(cursorpos *op*) performs a special operation coded by *op*, and returns t if it succeeds and nil if it doesn't. *op* is tested by string comparison, it is not a keyword symbol and may be in any package.

F	Moves one space to the right.
B	Moves one space to the left.
D	Moves one line down.
U	Moves one line up.
T	Homes up (moves to the top left corner). Note that t as the last argument to cursorpos is interpreted as a stream, so a stream *must* be specified if the T operation is used.
Z	Home down (moves to the bottom left corner).
A	Advances to a fresh line. See the :fresh-line stream operation.
C	Clears the window.
E	Clear from the cursor to the end of the window.
L	Clear from the cursor to the end of the line.
K	Clear the character position at the cursor.
X	B then K.

exploden *x*

exploden returns a list of characters (as fixnums) which are the characters that would be typed out by (princ *x*) (i.e. the unslashified printed representation of *x*).
Example:

```
(exploden '(+ /12 3)) => (50 53 40 61 62 40 63 51)
```

explodec *x*

explodec returns a list of characters represented by symbols which are the characters that would be typed out by (princ *x*) (i.e. the unslashified printed representation of *x*).
Example:

```
(explodec '(+ /12 3)) => ( /( + /  /1 /2 /  /3 /) )
```

(Note that there are slashified spaces in the above list.)

explode *x*

explode returns a list of characters represented by symbols which are the characters that would be typed out by (prin1 *x*) (i.e. the slashified printed representation of *x*).
Example:

```
(explode '(+ /12 3)) => ( /( + /  // /1 /2 /  /3 /) )
```

(Note that there are slashified spaces in the above list.)

flatsize *x*

flatsize returns the number of characters in the slashified printed representation of *x*.

flatc *x*

flatc returns the number of characters in the unslashified printed representation of *x*.

21.5 I/O Streams

21.5.1 What Streams Are

Many programs accept input characters and produce output characters. The method for performing input and output to one device is very different from the method for some other device. We would like our programs to be able to use any device available, but without each program having to know about each device.

In order to solve this problem, we introduce the concept of a *stream*. A stream is a source and/or sink of characters. A set of *operations* is available with every stream; operations include things like "output a character" and "input a character". The way to perform an operation to a stream is the same for all streams, although what happens inside the stream is very different depending on what kind of a stream it is. So all a program has to know is how to deal with streams.

A stream is a message-receiving object. This means that it is something that you can apply to arguments. The first argument is a keyword symbol which is the name of the operation you wish to perform. The rest of the arguments depend on what operation you are doing. Message-passing is explained in the flavor chapter (chapter 20, page 279).

Some streams can only do input, some can only do output, and some can do both. Some operations are only supported by some streams. Also, there are some operations which the stream may not support by itself, but will work anyway, albeit slowly, because the "stream default handler" can handle them. If you have a stream, there is an operation called :which-operations that will return a list of the names of all of the operations that are supported "natively" by the stream. *All* streams support :which-operations, and so it may not be in the list itself.

21.5.2 General Purpose Stream Operations

Here are some simple operations. Listed are the name of the operation, what arguments it takes, and what it does.

:tyo *char*

The stream will output the character *char*. For example, if s is bound to a stream, then the form

```
(funcall s ':tyo #/B)
```

will output a "B" to the stream.

:tyi &optional *eof*

The stream will input one character and return it. For example, if the next character to be read in by the stream is a "C", then the form

```
(funcall s ':tyi)
```

will return the value of #/C (that is, 103 octal). Note that the :tyi operation will not "echo" the character in any fashion; it just does the input. The tyi function (see page 332) will do echoing when reading from the terminal.

The optional *eof* argument to the :tyi message tells the stream what to do if it gets to the end of the file. If the argument is not provided or is **nil**, the stream will return **nil** at the end of file. Otherwise it will signal an error, and print out the argument as the error message. Note that this is *not* the same as the eof-option argument to **read**, **tyi**, and related functions.

:untyi *char*

The stream will remember the character *char*, and the next time a character is input, it will return the saved character. In other words, :untyi means "stuff this character back into the input source". For example,

```
(funcall s ':untyi 120)
(funcall s ':tyi) ==> 120
```

This operation is used by **read**, and any stream which supports :tyi must support :untyi as well. Note that you are only allowed to :untyi one character before doing a :tyi, and you aren't allowed to :untyi a different character than the last character you read from the stream. Some streams implement :untyi by saving the character, while others implement it by backing up the pointer to a buffer.

:which-operations

Returns a list of the operations supported "natively" by the stream.
Example:

```
(funcall s ':which-operations)
   ==> (:tyi :tyo :untyi :line-out :listen)
```

Any stream must either support :tyo, or support both :tyi and :untyi. There are several other, more advanced input and output operations which will work on any stream that can do input or output (respectively). Some streams support these operations themselves; you can tell by looking at the list returned by the :which-operations operation. Others will be handled by the "stream default handler" even if the stream does not know about the operation itself. However, in order for the default handler to do one of the more advanced output operations, the stream must support :tyo, and for the input operations the stream must support :tyi (and :untyi).

Here is the list of such operations:

:listen

On an interactive device, the :listen operation returns non-**nil** if there are any input characters immediately available, or **nil** if there is no immediately available input. On a non-interactive device, the operation always returns non-nil except at end-of-file, by virtue of the default handler. The main purpose of :listen is to test whether the user has hit a key, perhaps trying to stop a program in progress.

:fresh-line

This tells the stream that it should position itself at the beginning of a new line; if the stream is already at the beginning of a fresh line it will do nothing, otherwise it will output a carriage return. For streams which don't support this, the default handler will always output a carriage return.

:string-out *string* &optional *start end*

The characters of the string are successively output to the stream. This operation is provided for two reasons; first, it saves the writing of a loop which is used very often, and second, many streams can perform this operation much more efficiently than the equivalent sequence of :tyo operations. If the stream doesn't support :string-out itself, the default handler will turn it into a bunch of :tyos.

If *start* and *end* are not supplied, the whole string is output. Otherwise a substring is output; *start* is the index of the first character to be output (defaulting to 0), and *end* is one greater than the index of the last character to be output (defaulting to the length of the string). Callers need not pass these arguments, but all streams that handle :string-out must check for them and interpret them appropriately.

:line-out *string* &optional *start end*

The characters of the string, followed by a carriage return character, are output to the stream. *start* and *end* optionally specify a substring, as with :string-out. If the stream doesn't support :line-out itself, the default handler will turn it into a bunch of :tyos.

:line-in &optional *leader*

The stream should input one line from the input source, and return it as a string with the carriage return character stripped off. Contrary to what you might assume from its name, this operation is not much like the readline function.

Many streams have a string which is used as a buffer for lines. If this string itself were returned, there would be problems caused if the caller of the stream attempted to save the string away somewhere, because the contents of the string would change when the next line was read in. In order to solve this problem, the string must be copied. On the other hand, some streams don't reuse the string, and it would be wasteful to copy it on every :line-in operation. This problem is solved by using the *leader* argument to :line-in. If *leader* is nil (the default), the stream will not bother to copy the string, and the caller should not rely on the contents of that string after the next operation on the stream. If *leader* is t, the stream will make a copy. If *leader* is a fixnum then the stream will make a copy with an array leader *leader* elements long. (This is used by the editor, which represents lines of buffers as strings with additional information in their array-leaders, to eliminate an extra copy operation.)

If the stream reaches the end-of-file while reading in characters, it will return the characters it has read in as a string, and return a second value of t. The caller of the stream should therefore arrange to receive the second value, and check it to see whether the string returned was a whole line or just the trailing characters after the last carriage return in the input source.

:clear-input

The stream clears any buffered input. If the stream does not handle this, the default handler will ignore it.

:clear-output
The stream clears any buffered output. If the stream does not handle this, the default handler will ignore it.

:finish
This is for output streams to buffered asynchronous devices, such as the Chaosnet. **:finish** waits until the currently pending I/O operation has been completed. It does not do anything itself; it is just used to await completion of an operation. If there is buffered output for which I/O has not yet been started, it remains buffered. Do **:force-output** before **:finish** if you do not want this effect. If the stream does not handle this, the default handler will ignore it.

:force-output
This is for output streams to buffered asynchronous devices, such as the Chaosnet. **:force-output** causes any buffered output to be sent to the device. It does not wait for it to complete; use **:finish** for that. If a stream supports **:force-output**, then **:tyo**, **:string-out**, and **:line-out** may have no visible effect until a **:force-output** is done. If the stream does not handle this, the default handler will ignore it.

:close &optional *mode*
The stream is "closed", and no further operations should be performed on it. However, it is all right to **:close** a closed stream. If the stream does not handle **:close**, the default handler will ignore it.

The *mode* argument is normally not supplied. If it is **:abort**, we are abnormally exiting from the use of this stream. If the stream is outputting to a file, and has not been closed already, the stream's newly-created file will be deleted; it will be as if it was never opened in the first place. Any previously existing file with the same name will remain, undisturbed.

21.5.3 Special Purpose Stream Operations

There are several other defined operations which the default handler cannot deal with; if the stream does not support the operation itself, then sending that message will cause an error. This section documents the most commonly-used, least device-dependent stream operations. Windows, files, and Chaosnet connections have their own special stream operations which are documented separately.

:rubout-handler *options function* &rest *args*
This is supported by interactive streams such as windows on the TV terminal, and is described in its own section below (see section 21.7, page 361).

:beep &optional *type*
This is supported by interactive streams. It attracts the attention of the user by making an audible beep and/or flashing the screen. *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**.

:tyi-no-hang &optional *eof*

Just like :tyi except that if it would be necessary to wait in order to get the character, returns nil instead. This lets the caller efficiently check for input being available and get the input if there is any. :tyi-no-hang is different from :listen because it reads a character and because it is not simulated by the default-handler for streams which don't support it.

:untyo-mark

This is used by the grinder (see page 360) if the output stream supports it. It takes no arguments. The stream should return some object which indicates where output has gotten up to in the stream.

:untyo *mark*

This is used by the grinder (see page 360) in conjunction with :untyo-mark. It takes one argument, which is something returned by the :untyo-mark operation of the stream. The stream should back up output to the point at which the object was returned.

:read-cursorpos &optional (*units* ':pixel)

This operation is supported by windows. It returns two values: the current *x* and *y* coordinates of the cursor. It takes one optional argument, which is a symbol indicating in what units *x* and *y* should be; the symbols :pixel and :character are understood. :pixel means that the coordinates are measured in display pixels (bits), while :character means that the coordinates are measured in characters horizontally and lines vertically.

This operation, and :set-cursorpos, are used by the format "~T" request (see page 349), which is why "~T" doesn't work on all streams. Any stream that supports this operation must support :set-cursorpos as well.

:set-cursorpos *x y* &optional (*units* ':pixel)

This operation is supported by the same streams that support :read-cursorpos. It sets the position of the cursor. *x* and *y* are like the values of :read-cursorpos and *units* is the same as the *units* argument to :read-cursorpos.

:clear-screen

Erases the screen area on which this stream displays. Non-window streams don't support this operation.

There are many other special-purpose stream operations for graphics. They are not documented here, but in the window-system documentation. No claim that the above operations are the most useful subset should be implied.

21.5.4 Standard Streams

There are several variables whose values are streams used by many functions in the Lisp system. These variables and their uses are listed here. By convention, variables which are expected to hold a stream capable of input have names ending with -input, and similarly for output. Those expected to hold a bidirectional stream have names ending with -io.

standard-input *Variable*
In the normal Lisp top-level loop, input is read from standard-input (that is, whatever stream is the value of standard-input). Many input functions, including tyi and read, take a stream argument which defaults to standard-input.

standard-output *Variable*
In the normal Lisp top-level loop, output is sent to standard-output (that is, whatever stream is the value of standard-output). Many output functions, including tyo and print, take a stream argument which defaults to standard-output.

error-output *Variable*
The value of error-output is a stream to which error messages should be sent. Normally this is the same as standard-output, but standard-output might be bound to a file and error-output left going to the terminal. [This seems not be used by things which ought to use it.]

query-io *Variable*
The value of query-io is a stream which should be used when asking questions of the user. The question should be output to this stream, and the answer read from it. The reason for this is that when the normal input to a program may be coming from a file, questions such as "Do you really want to delete all of the files in your directory??" should be sent directly to the user, and the answer should come from the user, not from the data file. query-io is used by fquery and related functions; see page 488.

terminal-io *Variable*
The value of terminal-io is the stream which connects to the user's console. In an "interactive" program, it will be the window from which the program is being run; I/O on this stream will read from the keyboard and display on the TV. However, in a "background" program which does not normally talk to the user, terminal-io defaults to a stream which does not ever expect to be used. If it is used, perhaps by an error printout, it turns into a "background" window and requests the user's attention.

trace-output *Variable*
The value of trace-output is the stream on which the trace function prints its output.

eh:error-handler-io *Variable*
If non-nil, this is the stream which the error handler should use. This is used during debugging to divert the error handler to a stream which is known to work. The default value of nil causes the error handler to use error-output [or should, anyway].

standard-input, standard-output, error-output, trace-output, and query-io are initially bound to synonym streams which pass all operations on to the stream which is the value of terminal-io. Thus any operations performed on those streams will go to the TV terminal.

No user program should ever change the value of terminal-io. A program which wants (for example) to divert output to a file should do so by binding the value of standard-output; that way error messages sent to error-output can still get to the user by going through terminal-io, which is usually what is desired.

make-syn-stream *symbol*

make-syn-stream creates and returns a "synonym stream" (syn for short). Any operations sent to this stream will be redirected to the stream which is the value of *symbol*. A synonym stream is actually a symbol named *symbol*-syn-stream whose function definition is *symbol*, with a property that declares it to be a legitimate stream. The generated symbol is interned in the same package as *symbol*.

make-broadcast-stream &rest *streams*

Returns a stream which only works in the output direction. Any output sent to this stream will be sent to all of the streams given. The :which-operations is the intersection of the :which-operations of all of the streams. The value(s) returned by a stream operation are the values returned by the last stream in *streams*.

21.5.5 Making Your Own Stream

Here is a sample output stream, which accepts characters and conses them onto a list.

```
(defvar the-list nil)
(defun list-output-stream (op &optional arg1 &rest rest)
    (selectq op
        (:tyo
         (setq the-list (cons arg1 the-list)))
        (:which-operations '(:tyo))
        (otherwise
          (stream-default-handler (function list-output-stream)
                                  op arg1 rest))))
```

The lambda-list for a stream must always have one required parameter (op), one optional parameter (arg1), and a rest parameter (rest). This allows an arbitrary number of arguments to be passed to the default handler. This is an output stream, and so it supports the :tyo operation. Note that all streams must support :which-operations. If the operation is not one that the stream understands (e.g. :string-out), it calls the stream-default-handler. The calling of the default handler is *required*, since the willingness to accept :tyo indicates to the caller that :string-out will work.

Here is a typical input stream, which generates successive characters of a list.

```
(defvar the-list)       ;Put your input list here
(defvar untyied-char nil)
(defun list-input-stream (op &optional arg1 &rest rest)
  (selectq op
    (:tyi
     (cond ((not (null untyied-char))
            (prog1 untyied-char (setq untyied-char nil)))
           ((null the-list)
            (and arg1 (error arg1)))
           (t (prog1 (car the-list)
                     (setq the-list (cdr the-list))))))
    (:untyi
     (setq untyied-char arg1))
    (:which-operations '(:tyi :untyi))
    (otherwise
      (stream-default-handler (function list-input-stream)
                              op arg1 rest))))
```

The important things to note are that :untyi must be supported, and that the stream must check for having reached the end of the information, and do the right thing with the argument to the :tyi operation.

The above stream uses a free variable (the-list) to hold the list of characters, and another one (untyied-char) to hold the :untyied character (if any). You might want to have several instances of this type of stream, without their interfering with one another. This is a typical example of the usefulness of closures in defining streams. The following function will take a list, and return a stream which generates successive characters of that list.

```
(defun make-a-list-input-stream (list)
   (let-closed ((list list) (untyied-char nil))
      (function list-input-stream)))
```

The above streams are very simple and primitive. When designing a more complex stream, it is useful to have some tools to aid in the task. The defselect function (page 147) aids in defining message-receiving functions. The flavor system (chapter 20, page 279) provides powerful and elaborate facilities for programming message-receiving objects.

stream-default-handler *stream op arg1 rest*

stream-default-handler tries to handle the *op* operation on *stream*, given arguments of *arg1* and the elements of *rest*. The exact action taken for each of the defined operations is explained with the documentation on that operation, above.

21.6 Formatted Output

There are two ways of doing general formatted output. One is the function **format**. The other is the **output** subsystem. **format** uses a control string written in a special format specifier language to control the output format. **output** provides Lisp functions to do output in particular formats.

For simple tasks in which only the most basic format specifiers are needed, **format** is easy to use and has the advantage of brevity. For more complicated tasks, the format specifier language becomes obscure and hard to read. Then **output** becomes advantageous because it works with ordinary Lisp control constructs.

For formatting Lisp code (as opposed to text and tables), there is the grinder (see page 360).

21.6.1 The Format Function

format *destination control-string* &rest *args*

format is used to produce formatted output. format outputs the characters of *control-string*, except that a tilde ("~") introduces a directive. The character after the tilde, possibly preceded by prefix parameters and modifiers, specifies what kind of formatting is desired. Most directives use one or more elements of *args* to create their output; the typical directive puts the next element of *args* into the output, formatted in some special way.

The output is sent to *destination*. If *destination* is nil, a string is created which contains the output; this string is returned as the value of the call to format. In all other cases format returns no interesting value (generally it returns nil). If *destination* is a stream, the output is sent to it. If *destination* is t, the output is sent to standard-output. If *destination* is a string with an array-leader, such as would be acceptable to string-nconc (see page 129), the output is added to the end of that string.

A directive consists of a tilde, optional prefix parameters separated by commas, optional colon (":") and atsign ("@") modifiers, and a single character indicating what kind of directive this is. The alphabetic case of the character is ignored. The prefix parameters are generally decimal numbers. Examples of control strings:

```
"~S"        ; This is an S directive with no parameters.
"~3,4:@s"   ; This is an S directive with two parameters, 3 and 4,
            ;   and both the colon and atsign flags.
"~,4S"      ; The first prefix parameter is omitted and takes
            ;   on its default value, while the second is 4.
```

format includes some extremely complicated and specialized features. It is not necessary to understand all or even most of its features to use format efficiently. The beginner should skip over anything in the following documentation that is not immediately useful or clear. The more sophisticated features are there for the convenience of programs with complicated formatting requirements.

Sometimes a prefix parameter is used to specify a character, for instance the padding character in a right- or left-justifying operation. In this case a single quote (" ' ") followed by the desired character may be used as a prefix parameter, so that you don't have to know the decimal numeric values of characters in the character set. For example, you can use

    "~5,'0d"   instead of "~5,48d"

to print a decimal number in five columns with leading zeros.

In place of a prefix parameter to a directive, you can put the letter V, which takes an argument from *args* as a parameter to the directive. Normally this should be a number but it doesn't really have to be. This feature allows variable column-widths and the like. Also, you can use the character # in place of a parameter; it represents the number of arguments remaining to be processed.

Here are some relatively simple examples to give you the general flavor of how **format** is used.

```
(format nil "foo") => "foo"
(setq x 5)
(format nil "The answer is ~D." x) => "The answer is 5."
(format nil "The answer is ~3D." x) => "The answer is   5."
(setq y "elephant")
(format nil "Look at the ~A!" y) => "Look at the elephant!"
(format nil "The character ~:@C is strange." 1003)
      => "The character Meta-β (Top-X) is strange."
(setq n 3)
(format nil "~D item~:P found." n) => "3 items found."
(format nil "~R dog~:[s are~; is~] here." n (= n 1))
      => "three dogs are here."
(format nil "~R dog~:*~[~1; is~:;s are~] here." n)
      => "three dogs are here."
(format nil "Here ~[~1;is~:;are~] ~:*~R pupp~:@P." n)
      => "Here are three puppies."
```

The directives will now be described. *arg* will be used to refer to the next argument from *args*.

~A *arg*, any Lisp object, is printed without slashification (as by **princ**). ~:A prints **()** if *arg* is **nil**; this is useful when printing something that is always supposed to be a list. ~*n*A inserts spaces on the right, if necessary, to make the column width at least *n*. The @ modifier causes the spaces to be inserted on the left rather than the right. ~*mincol,colinc,minpad,padchar*A is the full form of ~A, which allows elaborate control of the padding. The string is padded on the right with at least *minpad* copies of *padchar*; padding characters are then inserted *colinc* characters at a time until the total width is at least *mincol*. The defaults are 0 for *mincol* and *minpad*, 1 for *colinc*, and space for *padchar*.

~S This is just like ~A, but *arg* is printed *with* slashification (as by **prin1** rather than **princ**).

~D *arg*, a number, is printed as a decimal integer. Unlike **print**, ~D will never put a decimal point after the number. ~*n*D uses a column width of *n*; spaces are inserted on the left if the number requires less than *n* columns for its digits and sign. If the number doesn't fit in *n* columns, additional columns are used as needed. ~*n*,*m*D uses *m* as the pad character instead of space. If *arg* is not a number, it is printed in ~A format and decimal base. The @ modifier causes the number's sign to be printed always; the default is only to print it if the number is negative. The : modifier causes commas to be printed between groups of three digits; the third prefix parameter may be used to change the character used as the comma. Thus the most general form of ~D is ~*mincol*,*padchar*,*commachar*D.

~O This is just like ~D but prints in octal instead of decimal.

~F *arg* is printed in floating point. ~*n*F rounds *arg* to a precision of *n* digits. The minimum value of *n* is 2, since a decimal point is always printed. If the magnitude of *arg* is too large or too small, it is printed in exponential notation. If *arg* is not a number, it is printed in ~A format. Note that the prefix parameter *n* is not *mincol*; it is the number of digits of precision desired. Examples:

```
(format nil "~2F" 5)  =>  "5.0"
(format nil "~4F" 5)  =>  "5.0"
(format nil "~4F" 1.5) => "1.5"
(format nil "~4F" 3.14159265)  =>  "3.142"
(format nil "~3F" 1e10)  =>  "1.0e10"
```

~E *arg* is printed in exponential notation. This is identical to ~F, including the use of a prefix parameter to specify the number of digits, except that the number is always printed with a trailing exponent, even if it is within a reasonable range.

~C (character *arg*) is put in the output. *arg* is treated as a keyboard character (see page 315), thus it may contain extra control-bits. These are printed first by representing them with Greek letters: alpha (α) for Control, beta (β) for Meta, epsilon (ε) for Control and Meta, lambda (λ) for Hyper, pi (π) for Super. If the character itself is alpha, beta, epsilon, lambda, pi, or equivalence-sign ($\equiv$), then it is preceded by an equivalence-sign to quote it.

With the colon flag (~:C), the names of the control bits are spelled out (e.g. "Control-Meta-F"), and also non-printing characters are represented by their names (e.g. "Return") rather than being output as themselves.

With both colon and atsign (~:@C), the colon-only format is printed, and then if the character requires the Top, Front, or Greek shift key(s) to type it, this fact is mentioned (e.g. "∀ (Top-U)"). This is the format used for telling the user about a key he is expected to type, for instance in prompt messages.

For all three of these formats, if the character is not a keyboard character but a mouse "character", it is printed as "Mouse-", the name of the button, "-", and the number of clicks.

With just an atsign (~@C), the character is printed in such a way that the Lisp reader can understand it, using "#/" or "#\".

~% Outputs a carriage return. ~*n*% outputs *n* carriage returns. No argument is used. Simply putting a carriage return in the control string would work, but ~% is usually used because it makes the control string look nicer in the Lisp source program.

~& The :fresh-line operation is performed on the output stream. Unless the stream knows that it is already at the front of a line, this outputs a carriage return. ~*n*& does a :fresh-line operation and then outputs *n-1* carriage returns.

~| Outputs a page separator character (#\page). ~*n*| does this *n* times. With a : modifier, if the output stream supports the :clear-screen operation this directive clears the screen, otherwise it outputs page separator character(s) as if no : modifier were present. | is vertical bar, not capital I.

~X Outputs a space. ~*n*X outputs *n* spaces.

~~ Outputs a tilde. ~*n*~ outputs *n* tildes.

~<CR> Tilde immediately followed by a carriage return ignores the carriage return and any whitespace at the beginning of the next line. With a :, the whitespace is left in place. With an @, the carriage return is left in place. This directive is typically used when a format control string is too long to fit nicely into one line of the program.

~* *arg* is ignored. ~*n** ignores the next *n* arguments. ~:* "ignores backwards"; that is, it backs up in the list of arguments so that the argument last processed will be processed again. ~*n*:* backs up *n* arguments. When within a ~{ construct (see below), the ignoring (in either direction) is relative to the list of arguments being processed by the iteration.

~P If *arg* is not 1, a lower-case "s" is printed. ("P" is for "plural".) ~:P does the same thing, after doing a ~:*; that is, it prints a lower-case s if the *last* argument was not 1. ~@P prints "y" if the argument is 1, or "ies" if it is not. ~:@P does the same thing, but backs up first.

~T Spaces over to a given column. ~*n*,*m*T will output sufficient spaces to move the cursor to column *n*. If the cursor is already past column *n*, it will output spaces to move it to column $n+mk$, for the smallest integer value *k* possible. *n* and *m* default to 1. Without the colon flag, *n* and *m* are in units of characters; with it, they are in units of pixels. Note: this operation *only* works properly on streams that support the :read-cursorpos and :set-cursorpos stream operations (see page 342). On other streams, any ~T operation will simply output two spaces. When **format** is creating a string, ~T will work, assuming that the first character in the string is at the left margin.

~R ~R prints *arg* as a cardinal English number, e.g. four. ~:R prints *arg* as an ordinal number, e.g. fourth. ~@R prints *arg* as a Roman numeral, e.g. IV. ~:@R prints *arg* as an old Roman numeral, e.g. IIII.

~*n*R prints *arg* in radix *n*. The flags and any remaining parameters are used as for the ~D directive. Indeed, ~D is the same as ~10R. The full form here is therefore ~*radix*,*mincol*,*padchar*,*commachar*R.

~*n* G "Goes to" the *n*th argument. ~0G goes back to the first argument in *args*. Directives after a ~*n*G will take sequential arguments after the one gone to. When within a ~{ construct, the "goto" is relative to the list of arguments being processed by the iteration. This is an "absolute goto"; for a "relative goto", see ~*.

~[*str0*~;*str1*~;...~;*strn*~]

This is a set of alternative control strings. The alternatives (called *clauses*) are separated by ~; and the construct is terminated by ~]. For example,

```
"~[Siamese ~;Manx ~;Persian ~;Tortoise-Shell ~
   ~;Tiger ~;Yu-Shiang ~]kitty"
```

The *arg*th alternative is selected; 0 selects the first. If a prefix parameter is given (i.e. ~*n*[), then the parameter is used instead of an argument (this is useful only if the parameter is "#"). If *arg* is out of range no alternative is selected. After the selected alternative has been processed, the control string continues after the ~].

~[*str0*~;*str1*~;...~;*strn*~:;*default*~] has a default case. If the *last* ~; used to separate clauses is instead ~:;, then the last clause is an "else" clause, which is performed if no other clause is selected. For example,

```
"~[Siamese ~;Manx ~;Persian ~;Tiger ~
   ~;Yu-Shiang ~:;Bad ~] kitty"
```

~[~*tag00,tag01,...*;*str0*~*tag10,tag11,...*;*str1*...~] allows the clauses to have explicit tags. The parameters to each ~; are numeric tags for the clause which follows it. That clause is processed which has a tag matching the argument. If ~*a1,a2,b1,b2,...*:; (note the colon) is used, then the following clause is tagged not by single values but by ranges of values *a1* through *a2* (inclusive), *b1* through *b2*, etc. ~:; with no parameters may be used at the end to denote a default clause. For example,

```
"~[~'+,'-,'*,'//;operator ~'A,'Z,'a,'z:;letter ~
   ~'0,'9:;digit ~:;other ~]"
```

~:[*false*~;*true*~] selects the *false* control string if *arg* is nil, and selects the *true* control string otherwise.

~@[*true*~] tests the argument. If it is not nil, then the argument is not used up, but is the next one to be processed, and the one clause is processed. If it is nil, then the argument is used up, and the clause is not processed. For example,

```
(setq prinlevel nil prinlength 5)
(format nil "~@[ PRINLEVEL=~D~]~@[ PRINLENGTH=~D~]"
            prinlevel prinlength)
   =>  " PRINLENGTH=5"
```

The combination of ~[and # is useful, for example, for dealing with English conventions for printing lists:

```
(setq foo "Items:~#[ none~; ~S~; ~S and ~
           ~S~:;~@{~#[~1; and~] ~S~^,~}~].")
(format nil foo)
        =>  "Items: none."
(format nil foo 'foo)
        =>  "Items: FOO."
(format nil foo 'foo 'bar)
        =>  "Items: FOO and BAR."
(format nil foo 'foo 'bar 'baz)
        =>  "Items: FOO, BAR, and BAZ."
(format nil foo 'foo 'bar 'baz 'quux)
        =>  "Items: FOO, BAR, BAZ, and QUUX."
```

~; Separates clauses in ~[and ~< constructions. It is undefined elsewhere.

~] Terminates a ~[. It is undefined elsewhere.

~{*str*~} This is an iteration construct. The argument should be a list, which is used as a set of arguments as if for a recursive call to **format**. The string *str* is used repeatedly as the control string. Each iteration can absorb as many elements of the list as it likes; if *str* uses up two arguments by itself, then two elements of the list will get used up each time around the loop. If before any iteration step the list is empty, then the iteration is terminated. Also, if a prefix parameter *n* is given, then there will be at most *n* repetitions of processing of *str*. Here are some simple examples:

```
(format nil "Here it is:~{ ~S~}." '(a b c))
    => "Here it is: A B C."
(format nil "Pairs of things:~{ <~S,~S>~}." '(a 1 b 2 c 3))
    => "Pairs of things: <A,1> <B,2> <C,3>."
```

~:{*str*~} is similar, but the argument should be a list of sublists. At each repetition step one sublist is used as the set of arguments for processing *str*; on the next repetition a new sublist is used, whether or not all of the last sublist had been processed. Example:

```
(format nil "Pairs of things:~:{ <~S,~S>~}."
            '((a 1) (b 2) (c 3)))
    => "Pairs of things: <A,1> <B,2> <C,3>."
```

~@{*str*~} is similar to ~{*str*~}, but instead of using one argument which is a list, all the remaining arguments are used as the list of arguments for the iteration. Example:

```
(format nil "Pairs of things:~@{ <~S,~S>~}."
            'a 1 'b 2 'c 3)
    => "Pairs of things: <A,1> <B,2> <C,3>."
```

~:@{*str*~} combines the features of ~:{*str*~} and ~@{*str*~}. All the remaining arguments are used, and each one must be a list. On each iteration the next argument is used as a list of arguments to *str*. Example:

```
(format nil "Pairs of things:~:@{ <~S,~S>~}."
            '(a 1) '(b 2) '(c 3))
     => "Pairs of things: <A,1> <B,2> <C,3>."
```

Terminating the repetition construct with ~:} instead of ~} forces *str* to be processed at least once even if the initial list of arguments is null (however, it will not override an explicit prefix parameter of zero).

If *str* is empty, then an argument is used as *str*. It must be a string, and precedes any arguments processed by the iteration. As an example, the following are equivalent:

```
(lexpr-funcall #'format stream string args)
(format stream "~1{~:}" string args)
```

This will use string as a formatting string. The ~1{ says it will be processed at most once, and the ~:} says it will be processed at least once. Therefore it is processed exactly once, using args as the arguments.

As another example, the format function itself uses format-error (a routine internal to the format package) to signal error messages, which in turn uses ferror, which uses format recursively. Now format-error takes a string and arguments, just like format, but also prints some additional information: if the control string in ctl-string actually is a string (it might be a list—see below), then it prints the string and a little arrow showing where in the processing of the control string the error occurred. The variable ctl-index points one character after the place of the error.

```
(defun format-error (string &rest args)
  (if (stringp ctl-string)
      (ferror nil "~1{~:}~%~VT↓~%~3X/"~A/"~%"
                string args (+ ctl-index 3) ctl-string)
      (ferror nil "~1{~:}" string args)))
```

This first processes the given string and arguments using ~1{~:}, then tabs a variable amount for printing the down-arrow, then prints the control string between double-quotes. The effect is something like this:

```
(format t "The item is a ~[Foo~;Bar~;Loser~]." 'quux)
>>ERROR: The argument to the FORMAT "~[" command
         must be a number
                    ↓
   "The item is a ~[Foo~;Bar~;Loser~]."
...
```

~} Terminates a ~{. It is undefined elsewhere.

~< *~mincol,colinc,minpad,padchar<text~>* justifies *text* within a field at least *mincol* wide. *text* may be divided up into segments with ~;—the spacing is evenly divided between the text segments. With no modifiers, the leftmost text segment is left justified in the field, and the rightmost text segment right justified; if there is only one, as a special case, it is right justified. The : modifier causes spacing to be introduced before the first text segment; the @ modifier causes spacing to be added after the last. *Minpad*, default 0, is the minimum number of *padchar* (default space) padding characters to be output between each segment. If the total width needed to satisfy these constraints is greater than *mincol*, then *mincol* is adjusted upwards in *colinc* increments. *colinc* defaults to 1. *mincol* defaults to 0. For example,

```
(format nil "~10<foo~;bar~>")              =>  "foo    bar"
(format nil "~10:<foo~;bar~>")             =>  "  foo  bar"
(format nil "~10:@<foo~;bar~>")            =>  "  foo bar "
(format nil "~10<foobar~>")                =>  "    foobar"
(format nil "~10:<foobar~>")               =>  "    foobar"
(format nil "~10@<foobar~>")               =>  "foobar    "
(format nil "~10:@<foobar~>")              =>  "  foobar  "
(format nil "$~10,,,'*<~3f~>" 2.59023) =>  "$******2.59"
```

Note that *text* may include format directives. The last example illustrates how the ~< directive can be combined with the ~f directive to provide more advanced control over the formatting of numbers.

Here are some examples of the use of ~^ within a ~< construct. ~^ is explained in detail below, however the general idea is that it eliminates the segment in which it appears and all following segments if there are no more arguments.

```
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo)
        =>  "            FOO"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar)
        =>  "FOO         BAR"
(format nil "~15<~S~;~^~S~;~^~S~>" 'foo 'bar 'baz)
        =>  "FOO   BAR   BAZ"
```

The idea is that if a segment contains a ~^, and **format** runs out of arguments, it just stops there instead of getting an error, and it as well as the rest of the segments are ignored.

If the first clause of a ~< is terminated with ~:; instead of ~;, then it is used in a special way. All of the clauses are processed (subject to ~^, of course), but the first one is omitted in performing the spacing and padding. When the padded result has been determined, then if it will fit on the current line of output, it is output, and the text for the first clause is discarded. If, however, the padded text will not fit on the current line, then the text segment for the first clause is output before the padded text. The first clause ought to contain a carriage return (~%). The first clause is always processed, and so any arguments it refers to will be used; the decision is whether to use the resulting segment of text, not whether to process the first clause. If the ~:; has a prefix parameter *n*, then the padded text must fit on the current line with *n* character positions to spare to avoid outputting the first clause's text. For example, the control string

```
"~%;; ~{~<~%;; ~1:; ~S~>~^,~}.~%"
```

can be used to print a list of items separated by commas, without breaking items over line boundaries, and beginning each line with ";; ". The prefix parameter 1 in ~1:; accounts for the width of the comma which will follow the justified item if it is not the last element in the list, or the period if it is. If ~:; has a second prefix parameter, then it is used as the width of the line, thus overriding the natural line width of the output stream. To make the preceding example use a line width of 50, one would write

```
"~%;; ~{~<~%;; ~1,50:; ~S~>~^,~}.~%"
```

If the second argument is not specified, then **format** sees whether the stream handles the :size-in-characters message. If it does, then **format** sends that message and uses the first returned value as the line length in characters. If it doesn't, **format** uses **95.** as the line length.

Rather than using this complicated syntax, one can often call the function **format:print-list** (see page 355).

~> Terminates a ~<. It is undefined elsewhere.

~^ This is an escape construct. If there are no more arguments remaining to be processed, then the immediately enclosing ~{ or ~< construct is terminated. If there is no such enclosing construct, then the entire formatting operation is terminated. In the ~< case, the formatting *is* performed, but no more segments are processed before doing the justification. The ~^ should appear only at the *beginning* of a ~< clause, because it aborts the entire clause. ~^ may appear anywhere in a ~{ construct.

If a prefix parameter is given, then termination occurs if the parameter is zero. (Hence ~^ is the same as ~#^.) If two parameters are given, termination occurs if they are equal. If three are given, termination occurs if the second is between the other two in ascending order. Of course, this is useless if all the prefix parameters are constants; at least one of them should be a # or a V parameter.

If ~^ is used within a ~:{ construct, then it merely terminates the current iteration step (because in the standard case it tests for remaining arguments of the current step only); the next iteration step commences immediately. To terminate the entire iteration process, use ~:^.

~Q An escape to arbitrary user-supplied code. *arg* is called as a function; its arguments are the prefix parameters to ~Q, if any. *args* can be passed to the function by using the V prefix parameter. The function may output to **standard-output** and may look at the variables **format:colon-flag** and **format:atsign-flag**, which are **t** or **nil** to reflect the : and @ modifiers on the ~Q. For example,

```
    (format t "~VQ" foo bar)
```
is a fancy way to say
```
    (funcall bar foo)
```
and discard the value. Note the reversal of order; the V is processed before the Q.

You can define your own directives. How to do this is not documented here; read the code. Names of user-defined directives longer than one character may be used if they are enclosed in backslashes (e.g. ~4,3\GRAPH\).

(Note: **format** also allows *control-string* to be a list. If the list is a list of one element, which is a string, the string is simply printed. This is for the use of the **format:outfmt** function below. The old feature wherein a more complex interpretation of this list was possible is now considered obsolete; use **format:output** if you like using lists.)

format:print-list *destination element-format list* &optional *separator start-line tilde-brace-options*

This function provides a simpler interface for the specific purpose of printing comma-separated lists with no list element split across two lines; see the description of the ~:; directive (page 353) to see the more complex way to do this within **format**. *destination* tells where to send the output; it can be **t**, **nil**, a **string-nconc**-able string, or a stream, as with **format**. *element-format* is a **format** control-string which tells how to print each element of *list*; it is used as the body of a "~{...~}" construct. *separator*, which defaults to ", " (comma, space) is a string which goes after each element except the last. **format** control commands are not recommended in *separator*. *start-line*, which defaults to three spaces, is a **format** control-string which is used as a prefix at the beginning of each line of output, except the first. **format** control commands are allowed in *separator*, but they should not swallow arguments from *list*. *tilde-brace-options* is a string inserted before the opening "{"; it defaults to the null string, but allows you to insert colon and/or atsign. The line-width of the stream is computed the same way that the ~:; command computes it; it is not possible to override the natural line-width of the stream.

21.6.2 The Output Subsystem

The formatting functions associated with the output subsystem allow you to do formatted output using Lisp-style control structure. Instead of a directive in a format control string, there is one formatting function for each kind of formatted output.

The calling conventions of the formatting functions are all similar. The first argument is usually the datum to be output. The second argument is usually the minimum number of columns to use. The remaining arguments are options—alternating keywords and values.

Options which most functions accept include :padchar, followed by a character to use for padding; :minpad, followed by the minimum number of padding characters to output after the data; and :tab-period, followed by the distance between allowable places to stop padding. To make the meaning of :tab-period clearer, if the value of :tab-period is 5, the minimum size of the field is 10, and the value of :minpad is 2, then a datum that takes 9 characters will be padded out to 15 characters. The requirement to use at least two characters of padding means it can't fit into 10 characters, and the :tab-period of 5 means the next allowable stopping place is at 10+5 characters. The default values for :minpad and :tab-period, if they are not specified, are zero and one. The default value for :padchar is space.

The formatting functions always output to standard-output and do not require an argument to specify the stream. The macro format:output allows you to specify the stream or a string, just as format does, and also makes it convenient to concatenate constant and variable output.

format:output *stream string-or-form...* *Macro*

format:output makes it convenient to intersperse arbitrary output operations with printing of constant strings. standard-output is bound to *stream*, and each *string-or-form* is processed in succession from left to right. If it is a string it is printed, otherwise it is a form which is evaluated and the value is discarded. Presumably the forms will send output to standard-output.

If *stream* is written as nil, then the output is put into a string which is returned by format:output. If *stream* is written as t, then the output goes to the prevailing value of standard-output. Otherwise *stream* is a form which must evaluate to a stream.

Here is an example:

```
(format:output t "FOO is " (prin1 foo) " now." (terpri))
```

Because format:output is a macro, what matters about *stream* is not whether it *evaluates* to t or nil, but whether it is actually written as t or nil.

format:outfmt *string-or-form...* *Macro*

Some system functions ask for a format control string and arguments, to be printed later. If you wish to generate the output using the formatted output functions, you can use format:outfmt, which produces a control argument which will eventually make format print the desired output (this is a list whose one element is a string containing the output). A call to format:outfmt can be used as the second argument to ferror, for example:

```
(ferror nil (format:outfmt "Foo is " (format:onum foo)
                           " which is too large"))
```

format:onum *number* &optional *radix minwidth* &rest *options*

format:onum outputs *number* in base *radix*, padding to at least *minwidth* columns and obeying the other padding options specified as described above.

radix can be a number, or it can be :roman, :english, or :ordinal. The default *radix* is 10. (decimal).

Two special keywords are allowed as options: :signed and :commas. :signed with value t means print a sign even if the number is positive. :commas with value t means print a comma every third digit in the customary way. These options are meaningful only with numeric radices.

format:ofloat *number* &optional *n-digits force-exponential-notation minwidth* &rest *options*

format:ofloat outputs *number* as a floating point number using *n-digits* digits. If *force-exponential-notation* is non-nil, then an exponent is always used. *minwidth* and *options* are used to control padding as usual.

format:ostring *string* &optional *minwidth* &rest *options*

format:ostring outputs *string*, padding to at least *minwidth* columns if *minwidth* is not nil, and obeying the other padding options specified as described above.

Normally the contents of the string are left-justified; any padding follows the data. The special option :right-justify causes the padding to come before the data. The amount of padding is not affected.

The argument need not really be a string. Any Lisp object is allowed, and it is output by princ.

format:oprint *object* &optional *minwidth* &rest *options*

format:oprint prints *object*, any Lisp object, padding to at least *minwidth* columns if *minwidth* is not nil, and obeying the padding options specified as described above.

Normally the data are left justified; any padding follows. The special option :right-justify causes the padding to come before the data. The amount of padding is not affected.

The printing of the object is done with prin1.

format:ochar *character* &optional *style top-explain minwidth* &rest *options*

format:ochar outputs *character* in one of three styles, selected by the *style* argument. *minwidth* and *options* control padding as usual.

If *style* is :read, nil, or not specified, then the character is printed using #/ or #\ so that it could be read back in.

If *style* is :editor, then the output is in the style of the string "Meta-Rubout".

If *style* is :sail, a somewhat more abbreviated style is used in which alpha, beta, etc. are used to represent "Control" and "Meta", and shorter names for characters are also used when possible. See section 21.1, page 314.

top-explain is useful with the :editor and :sail styles. It says that any character which has to be typed using the Top or Greek keys should be followed by an explanation of how to type it. For example: "α (Top-Z)" or "α (Greek-a)", depending on the type of keyboard in use.

format:tab *mincol* &rest *options*

format:tab outputs padding at least until column *mincol*. It is the only formatting function which bases its actions on the actual cursor position rather than the width of what is being output. The padding options :padchar, :minpad, and :tab-period are obeyed. Thus, at least the :minpad number of padding characters are output even if that goes past *mincol*, and once past *mincol*, padding can only stop at a multiple of :tab-period characters past *mincol*.

In addition, if the :terpri option is t, then if column *mincol* is passed, format:tab starts a new line and indents it to *mincol*.

The :unit option specifies the units of horizontal position. The default is to count in units of characters. If :unit is specified as :pixel, then the computation (and the argument *mincol* and the :minpad and :tab-period options) are in units of pixels.

format:pad (*minwidth option...*) *body...* *Macro*

format:pad is used for printing several items in a fixed amount of horizontal space, padding between them to use up any excess space. Each of the *body* forms prints one item. The padding goes between items. The entire format:pad always uses at least *minwidth* columns; any columns that the items don't need are distributed as padding between the items. If that isn't enough space, then more space is allocated in units controlled by the :tab-period option until there is enough space. If it's more than enough, the excess is used as padding.

If the :minpad option is specified, then at least that many pad characters must go between each pair of items.

Padding goes only between items. If you want to treat several actual pieces of output as one item, put a progn around them. If you want padding before the first item or after the last, as well as between the items, include a dummy item nil at the beginning or the end.

If there is only one item, it is right justified. One item followed by nil is left-justified. One item preceded and followed by nil is centered. Therefore, format:pad can be used to provide the usual padding options for a function which does not provide them itself.

format:plural *number singular* &optional *plural*

format:plural outputs either the singular or the plural form of a word depending on the value of *number*. The singular is used if and only if *number* is 1. *singular* specifies the singular form of the word. string-pluralize is used to compute the plural, unless *plural* is explicitly specified.

It is often useful for *number* to be a value returned by format:onum, which returns its argument. For example:

```
        (format:plural (format:onum n-frobs) " frob")
```

will print "1 frob" or "2 frobs".

format:breakline *linel print-if-terpri print-always...* *Macro*

format:breakline is used to go to the next line if there is not enough room for something to be output on the current line. The *print-always* forms print the text which is supposed to fit on the line. *linel* is the column before which the text must end. If it doesn't end before that column, then format:breakline moves to the next line and executes the *print-if-terpri* form before doing the *print-always* forms.

Constant strings are allowed as well as forms for *print-if-terpri* and *print-always*. A constant string is just printed.

To go to a new line unconditionally, simply call terpri.

Here is an example which prints the elements of a list, separated by commas, breaking lines between elements when necessary.

```
(defun pcl (list linel)
  (do ((l list (cdr l))) ((null l))
    (format:breakline linel "  "
      (princ (car l))
      (and (cdr l) (princ ", ")))))
```

21.6.3 Formatting Lisp Code

grindef *function-spec...* *Special Form*

Prints the definitions of one or more functions, with indentation to make the code readable. Certain other "pretty-printing" transformations are performed: The quote special form is represented with the ' character. Displacing macros are printed as the original code rather than the result of macro expansion. The code resulting from the backquote (`) reader macro is represented in terms of '.

The subforms to grindef are the function specs whose definitions are to be printed; the usual way grindef is used is with a form like (grindef foo) to print the definition of foo. When one of these subforms is a symbol, if the symbol has a value its value is prettily printed also. Definitions are printed as defun special forms, and values are printed as setq special forms.

If a function is compiled, grindef will say so and try to find its previous interpreted definition by looking on an associated property list (see uncompile (page 197). This will only work if the function's interpreted definition was once in force; if the definition of the function was simply loaded from a QFASL file, grindef will not find the interpreted definition and will not be able to do anything useful.

With no subforms, grindef assumes the same arguments as when it was last called.

grind-top-level *obj* &optional *width* (*stream* standard-output) (*untyo-p* nil) (*displaced* 'si:displaced) (*terpri-p* t) *notify-fun loc*

Pretty-prints *obj* on *stream*, putting up to *width* characters per line. This is the primitive interface to the pretty-printer. Note that it does not support variable-width fonts. If the *width* argument is supplied, it is how many characters wide the output is to be. If *width* is unsupplied or nil, grind-top-level will try to figure out the "natural width" of the stream, by sending a :size-in-characters message to the stream and using the first returned value. If the stream doesn't handle that message, a width of 95. characters is used instead.

The remaining optional arguments activate various strange features and usually should not be supplied. These options are for internal use by the system, and are only documented here for completeness. If *untyo-p* is t, the :untyo and :untyo-mark operations will be used on *stream*, speeding up the algorithm somewhat. *displaced* controls the checking for displacing macros; it is the symbol which flags a place that has been displaced, or nil to disable the feature. If *terpri-p* is nil, grind-top-level does not advance to a fresh line before printing.

If *notify-fun* is non-nil, it is a function of three arguments which is called for each "token" in the pretty-printed output. Tokens are atoms, open and close parentheses, and reader macro characters such as '. The arguments to *notify-fun* are the token, its "location" (see next paragraph), and t if it is an atom or nil if it is a character.

loc is the "location" (typically a cons) whose car is *obj*. As the grinder recursively descends through the structure being printed, it keeps track of the location where each thing came from, for the benefit of the *notify-fun*, if any. This makes it possible for a program to correlate the printed output with the list structure. The "location" of a close parenthesis is t, because close parentheses have no associated location.

21.7 Rubout Handling

The rubout handler is a feature of all interactive streams, that is, streams which connect to terminals. Its purpose is to allow the user to edit minor mistakes in typein. At the same time, it is not supposed to get in the way; input is to be seen by Lisp as soon as a syntactically complete form has been typed. The definition of "syntactically complete form" depends on the function that is reading from the stream; for read, it is a Lisp expression.

Some interactive streams ("editing Lisp listeners") have a rubout handler which allows input to be edited with the full power of the Zwei editor. Other streams have a simple rubout handler which just allows rubbing out of single characters, and a few simple commands like clearing the screen and erasing the entire input typed so far. This section describes the general protocol used to deal with any rubout handler, and it also discusses the simple rubout handler and what commands it deals with.

The tricky thing about the rubout handler is the need for it to figure out when you are all done. The idea of a rubout handler is that you can type in characters, and they are saved up in a buffer so that if you change your mind, you can rub them out and type different characters. However, at some point, the rubout handler has to decide that the time has come to stop putting characters into the buffer, and let the function, such as read, start processing the characters. This is called "activating". The right time to activate depends on the function calling the rubout handler, and may be very complicated (if the function is read, figuring out when one Lisp expression has been typed requires knowledge of all the various printed representations, what all currently-defined reader macros do, and so on). Rubout handlers should not have to know how to parse the characters in the buffer to figure out what the caller is reading and when to activate; only the caller should have to know this. The rubout handler interface is organized so that the calling function can do all the parsing, while the rubout handler does all the handling of rubouts, and the two are kept completely separate.

The basic way that the rubout handler works is as follows. When an input function that reads an "object", such as read or readline (but not tyi), is called to read from a stream which has :rubout-handler in its :which-operations list, that function "enters" the rubout handler. It then goes ahead :tyi'ing characters from the stream. Because control is inside the rubout handler, the stream will echo these characters so the user can see what he is typing. (Normally echoing is considered to be a higher-level function outside of the province of streams, but when the higher-level function tells the stream to enter the rubout handler it is also handing it the responsibility for echoing). The rubout handler is also saving all these characters in a buffer, for reasons disclosed in the following paragraph. When the function, read or whatever, decides it has enough input, it returns and control "leaves" the rubout handler. That was the easy case.

If the user types a rubout, a *throw is done, out of all recursive levels of read, reader macros, and so forth, back to the point where the rubout handler was entered. Also the rubout is echoed by erasing from the screen the character which was rubbed out. Now the read is tried over again, re-reading all the characters which had been typed and not rubbed out, not echoing them this time. When the saved characters have been exhausted, additional input is read from the user in the usual fashion.

The effect of this is a complete separation of the functions of rubout handling and parsing, while at the same time mingling the execution of these two functions in such a way that input is always "activated" at just the right time. It does mean that the parsing function (in the usual case, read and all macro-character definitions) must be prepared to be thrown through at any time and should not have non-trivial side-effects, since it may be called multiple times.

If an error occurs while inside the rubout handler, the error message is printed and then additional characters are read. When the user types a rubout, it rubs out the error message as well as the character that caused the error. The user can then proceed to type the corrected expression; the input will be reparsed from the beginning in the usual fashion.

The simple rubout handler also recognizes the special characters Clear-Input, Clear-Screen, and Delete. (These are Clear, Form, and VT on old keyboards.) Clear-Screen clears the screen and echoes back the buffered input. Clear-Input is like hitting enough rubouts to flush all the buffered input. Delete is like Clear-Screen in that it echoes back the input, but it does not clear the screen. [It should be moved to a different key, shouldn't it?]

If a character with control shifts (Control, Meta, Super, or Hyper) is typed at a rubout handler that does not support the full set of editing commands, such as the simple rubout handler, it beeps and ignores the character. These characters are reserved in this context for editing use. The rubout handler based on the Zwei editor interprets control characters in the usual Zwei way: as editing commands, allowing you to edit your buffered input. When not inside the rubout handler, and when typing at a program that uses control characters for its own purposes, control characters are treated the same as ordinary characters.

The following explanation tells you how to write your own function that invokes the rubout handler. The functions read and readline both work this way. You should use the readline1 example, below, as a template for writing your own function.

The way that the rubout handler is entered is complicated, since a *catch must be established. The variable rubout-handler is non-nil if the current process is inside the rubout handler. This is used to handle recursive calls to read from inside reader macros and the like. If rubout-handler is nil, and the stream being read from has :rubout-handler in its :which-operations, functions such as read send the :rubout-handler message to the stream with arguments of a list of options, the function, and its arguments. The rubout handler initializes itself and establishes its *catch, then calls back to the specified function with rubout-handler bound to t. User-written input reading functions should follow this same protocol, to get the same input editing benefits as read and readline.

rubout-handler *Variable*
t if control is inside the rubout handler in this process.

As an example of how to use the rubout handler, here is a simplified version of the readline function. It doesn't bother about end-of-file handling, use of :line-in for efficiency, etc.

```
(defun readline1 (stream)
  ;; If stream does rubout handling, get inside rubout handler
  (cond ((and (not rubout-handler)
              (memq ':rubout-handler
                    (funcall stream ':which-operations)))
         (funcall stream ':rubout-handler '() #'readline1 stream))
        ;; Accumulate characters until return
        (t (do ((ch (funcall stream ':tyi)
                    (funcall stream ':tyi))
                (len 100)
                (string (make-array 100 ':type 'art-string))
                (idx 0))
               ((or (null ch) (= ch #\cr))
                (adjust-array-size string idx)
                string)
             (if (= idx len)
                 (adjust-array-size string (setq len (+ len 40))))
             (aset ch string idx)
             (setq idx (1+ idx))))))
```

The first argument to the :rubout-handler message is a list of options. The second argument is the function that the rubout handler should call to do the reading, and the rest of the arguments are passed to that function. Note that in the example above, readline1 is sending the :rubout-handler message passing itself as the function, and its own arguments as the arguments. This is the usual thing to do. It isn't passing any options. The returned values of the message are normally the returned values of the function (except sometimes when the :full-rubout option is used; see below).

Each option in the list of options given as the first argument to the :rubout-handler message consists of a list whose first element is a keyword and whose remaining elements are "arguments" to that keyword. Note that this is not the same format as the arguments to a typical function that takes keyword arguments; rather this is an a-list of options. The standard options are:

(:full-rubout *val*)
: If the user rubs out all the characters he typed, then control will be returned from the rubout handler immediately. Two values are returned; the first is nil and the second is *val*. (if the user doesn't rub out all the characters, then the rubout handler propagates multiple values back from the function that it calls, as usual.) In the absence of this option, the rubout handler would simply wait for more characters to be typed in, and would ignore any additional rubouts.

(:pass-through *char1 char2*...)
: The characters *char1*, *char2*, etc. are not to be treated as special by the rubout handler. You can use this to override the default processing of characters such as Clear-Input and to receive control characters. Any function that reads input and uses non-printing characters for anything should list them in a :pass-through option. This way, if input is being rubout-handled by the editor, those non-printing characters will get their desired meaning rather than their meaning as editor commands.

(:prompt *function*)
(:reprompt *function*)

When it is time for the user to be prompted, *function* is called with two arguments. The first is a stream it may print on; the second is the character which caused the need for prompting, e.g. #\clear-input or #\clear-screen, or nil if the rubout handler was just entered.

The difference between :prompt and :reprompt is that the latter does not call the prompt function when the rubout handler is first entered, but only when the input is redisplayed (e.g. after a screen clear). If both options are specified then :reprompt overrides :prompt except when the rubout handler is first entered.

(:initial-input *string*)

Pretends that the user typed *string*. When the rubout handler is entered, *string* is typed out. The user can add more characters to it or rubout characters from it.

(:do-not-echo *char-1 char-2...*)

The characters *char-1*, *char-2*, etc. are not to be echoed when the user types them. The comparison is done with =, not char-equal. You can use this to suppress echoing of the return character that terminated a readline, for example.

21.8 The :read and :print Stream Operations

A stream can specially handle the reading and printing of objects by handling the :read and :print stream operations. Note that these operations are optional and most streams do not support them.

If the read function is given a stream which has :read in its which-operations, then instead of reading in the normal way it sends the :read message to the stream with one argument, read's *eof-option* if it had one or a magic internal marker if it didn't. Whatever the stream returns is what read returns. If the stream wants to implement the :read operation by internally calling read, it must use a different stream which does not have :read in its which-operations.

If a stream has :print in its which-operations, it may intercept all object printing operations, including those due to the print, prin1, and princ functions, those due to format, and those used internally, for instance in printing the elements of a list. The stream receives the :print message with three arguments: the object being printed, the *prindepth* (for comparison against the prinlevel variable), and *slashify-p* (t for prin1, nil for princ). If the stream returns nil, then normal printing takes place as usual. If the stream returns non-nil, then print does nothing; the stream is assumed to have output an appropriate printed representation for the object. The two following functions are useful in this connection; however, they are in the system-internals package and may be changed without much notice.

si:print-object *object prindepth slashify-p stream* &optional *which-operations*
Outputs the printed-representation of *object* to *stream*, as modified by *prindepth* and *slashify-p*. This is the internal guts of the Lisp printer. When a stream's **:print** handler calls this function, it should supply the list (**:string-out**) for *which-operations*, to prevent itself from being called recursively. Or it can supply **nil** if it does not want to receive **:string-out** messages.

If you want to customize the behavior of all printing of Lisp objects, advising (see section 26.4, page 460) this function is the way to do it. See section 21.2.1, page 321.

si:print-list *list prindepth slashify-p stream which-operations*
This is the part of the Lisp printer that prints lists. A stream's **:print** handler can call this function, passing along its own arguments and its own which-operations, to arrange for a list to be printed the normal way and the stream's **:print** hook to get a chance at each of the list's elements.

21.9 Accessing Files

The Lisp Machine can access files on a variety of remote file servers, which are typically (but not necessarily) accessed through the Chaosnet, as well as accessing files on the Lisp Machine itself, if the machine has its its own file system. This section tells you how to get a stream which reads or writes a given file, and what the device-dependent operations on that stream are. Files are named with *pathnames*. Since pathnames are quite complex they have their own chapter; see chapter 22, page 376.

with-open-file (*stream pathname options*) *body...* *Special Form*
Evaluates the *body* forms with the variable *stream* bound to a stream which reads or writes the file named by the value of *pathname*. *options* should evaluate to a keyword or list of keywords. These options control whether the stream is for input from a existing file or output to a new file, whether the file is text or binary, etc.

When control leaves the body, either normally or abnormally (via ***throw**), the file is closed. If a new output file is being written, and control leaves abnormally, the file is aborted and it is as if it were never written. Because it always closes the file, even when an error exit is taken, **with-open-file** is preferred over **open**. Opening a large number of files and forgetting to close them tends to break some remote file servers, ITS's for example.

pathname is the name of the file to be opened; it can be a pathname object, a string, a symbol, or a Maclisp-compatible "namelist". It can be anything acceptable to **fs:parse-pathname**; the complete rules for parsing pathnames are explained in chapter 22, page 376.

If an error, such as file not found, occurs the user is asked to supply an alternate pathname, unless this is overridden by *options*. At that point he can quit out or enter the error handler, if the error was not due to a misspelled pathname.

options is either a single symbol or a (possibly-null) list of symbols. The following option symbols are recognized:

:in, :read	Select opening for input (the default).
:out, :write, :print	Select opening for output; a new file is to be created.
:fixnum	Select binary mode, otherwise character mode is used. Note that fixnum mode uses 16-bit binary words and is not compatible with Maclisp fixnum mode which uses 36-bit words. On the pdp-10, fixnum files are stored with two 16-bit words per pdp-10 word, left-justified and in pdp-10 byte order.
:ascii	The opposite of :fixnum. This is the default.
:single, :block	Ignored for compatibility with the Maclisp open function.
:byte-size	Must be followed by a number in the options list, and must be used in combination with :fixnum. The number is the number of bits per byte, which can be from 1 to 16. On a pdp-10 file server these bytes will be packed into words in the standard way defined by the ILDB instruction. The :tyi stream operation will (of course) return the bytes one at a time.
:probe	The file is not being opened to do I/O, but only to find out information about it. A stream is returned but the normal I/O messages should not be sent to it. The special file-attribute messages described below (see section 21.9.3, page 371) may be sent. It is as if the stream were immediately closed after opening it. :probe implies :noerror and :fixnum.
:noerror	If the file cannot be opened, then instead of returning a stream, a string containing the error message is returned. If :noerror was not specified, this error string would have been displayed and the user would have been asked to supply an alternate pathname.
:super-image	Disables the special treatment of rubout in ascii files. Normally rubout is an escape which causes the following character to be interpreted specially, allowing all characters from 0 through 376 to be stored. This applies to pdp-10 file servers only.
:raw	Disables all character set translation in ascii files. This applies to pdp-10 file servers only.
:deleted :temporary	These options are for TOPS-20 file servers only. They specify TOPS-20-specific attributes of the file to be opened.

For compatibility with the Maclisp open function (see below), which uses the same keywords as with-open-file, the keyword symbols can be in any package and need not be prefixed with colons. Zetalisp programs should include the colons for consistency of style.

For example, evaluating any of the forms

```
        (with-open-file (foo "info;dir >" ':in) ...)
        (with-open-file (foo "INFO;DIR >" '(:read)) ...)
        (with-open-file (foo "DIR > INFO;" ':read) ...)
```
will open the file "AI: INFO; DIR >" (assuming AI is the current default file server), and will return an input stream which will return successive characters of the file.

open *pathname options*
Returns a stream which is connected to the specified file. Unlike Maclisp, the **open** function only creates streams for *files*; streams for other devices are created by other functions. The *pathname* and *options* arguments are the same as in **with-open-file**; see above. If an error, such as file not found, occurs, the user is asked to supply an alternate pathname, unless this is overridden by *options*.

When the caller is finished with the stream, it should close the file by using the **:close** operation or the **close** function. The **with-open-file** special form does this automatically, and so is usually preferred. **open** should only be used when the control structure of the program necessitates opening and closing of a file in some way more complex than the simple way provided by **with-open-file**. Any program that uses **open** should set up **unwind-protect** handlers (see page 49) to close its files in the event of an abnormal exit.

close *stream*
The **close** function simply sends the **:close** message to *stream*.

renamef *file new-name* &optional (*error-p* **t**)
file can be a pathname or a stream which is open to a file. The specified file is renamed to *new-name* (a pathname). If *error-p* is **t**, then if an error occurs it will be signalled as a Lisp error. If *error-p* is **nil** and an error occurs, the error message will be returned as a string, otherwise **t** will be returned.

deletef *file* &optional (*error-p* **t**)
file can be a pathname or a stream which is open to a file. The specified file is deleted. If *error-p* is **t**, then if an error occurs it will be signalled as a Lisp error. If *error-p* is **nil** and an error occurs, the error message will be returned as a string, otherwise **t** will be returned.

probef *pathname*
Returns **nil** if there is no file named *pathname*, otherwise returns a pathname which is the true name of the file, which can be different from *pathname* because of file links, version numbers, etc.

fs:close-all-files
Closes all open files. This is useful when a program has run wild opening files and not closing them.

21.9.1 Loading Files

To *load* a file is to read through the file, evaluating each form in it. Programs are typically stored in files; the expressions in the file are mostly special forms such as **defun** and **defvar** which define the functions and variables of the program.

Loading a compiled (or QFASL) file is similar, except that the file does not contain text but rather pre-digested expressions created by the compiler which can be loaded more quickly.

These functions are for loading single files. There is a system for keeping track of programs which consist of more than one file; for further information refer to chapter 24, page 406.

load *pathname* &optional *pkg nonexistent-ok dont-set-default*

This function loads the file named by *pathname* into the Lisp environment. If the file is a QFASL file, it calls fasload; otherwise it calls readfile. Normally the file is read into its "home" package, but if *pkg* is supplied it is the package in which the file is to be read. *pkg* can be either a package or the name of a package as a string or a symbol. If *pkg* is not specified, load prints a message saying what package the file is being loaded into. If *nonexistent-ok* is specified, load just returns if the file cannot be opened.

pathname can be anything acceptable to fs:parse-pathname; pathnames and the complete rules for parsing them are explained in chapter 22, page 376. *pathname* is defaulted from fs:load-pathname-defaults (see page 381), which is the set of defaults used by load, qc-file, and similar functions. Normally load updates the pathname defaults from *pathname*, but if *dont-set-default* is specified this is suppressed.

If *pathname* contains an FN1 but no FN2, load will first look for the file with an FN2 of QFASL, then it will look for an FN2 of >. For non-ITS file systems, this generalizes to: if *pathname* specifies a type and/or a version, load loads that file. Otherwise it first looks for a type-QFASL file, then a type-LISP file, in both cases looking for the newest version.

readfile *pathname* &optional *pkg no-msg-p*

readfile is the version of load for text files. It reads and evaluates each expression in the file. As with load, *pkg* can specify what package to read the file into. Unless *no-msg-p* is t, a message is printed indicating what file is being read into what package. The defaulting of *pathname* is the same as in load.

fasload *pathname* &optional *pkg no-msg-p*

fasload is the version of load for QFASL files. It defines functions and performs other actions as directed by the specifications inserted in the file by the compiler. As with load, *pkg* can specify what package to read the file into. Unless *no-msg-p* is t, a message is printed indicating what file is being read into what package. The defaulting of *pathname* is the same as in load.

21.9.2 File Property Lists

Any text file can contain a "property list" which specifies several attributes of the file. The above loading functions, the compiler, and the editor look at this property list. File property lists are especially useful in program source files, i.e. a file that is intended to be loaded (or compiled and then loaded).

If the first non-blank line in the file contains the three characters "-*-", some text, and "-*-" again, the text is recognized as the file's property list. Each property consists of the property name, a colon, and the property value. If there is more than one property they are separated by semicolons. An example of such a property list is:

```
    ; -*- Mode:Lisp; Package:Cellophane; Base:10 -*-
```

The semicolon makes this line look like a comment rather than a Lisp expression. This defines three properties: mode, package, and base. Another example is:

```
    .c This is part of the Lisp machine manual.  -*- Mode:Bolio -*-
```

A property name is made up of letters, numbers, and otherwise-undefined punctuation characters such as hyphens. A property value can be such a name, or a decimal number, or several such items separated by commas. Spaces may be used freely to separate tokens. Upper and lower-case letters are not distinguished. There is *no* quoting convention for special characters such as colons and semicolons. Thus file property lists are similar in spirit to Lisp property lists.

The file property list format actually has nothing to do with Lisp; it is just a convention for placing some information into a file that is easy for a program to interpret. The Emacs editor on the pdp-10 knows how to interpret these property lists (primarily in order to look at the **Mode** property).

Within the Lisp Machine, there exists a parser for file property lists that creates some Lisp data structure that corresponds to the file property list. When a file property list is read in and given to the parser (the **fs:file-read-property-list** function, see below), it is converted into Lisp objects as follows: Property names are interpreted as Lisp symbols, and interned on the keyword package. Numbers are interpreted as Lisp fixnums, and are read in decimal. If a property value contains any commas, then the commas separate several expressions which are formed into a list.

When a file is edited, loaded, or compiled, its file property list is read in and the properties are stored on the property list of the generic pathname (see section 22.3, page 381) for that file, where they can be retrieved with the **:get** and **:plist** messages. So the way you examine the properties of a file is usually to use messages to a pathname object that represents the generic pathname of a file. Note that there other properties there, too. The function **fs:file-read-property-list** (see below) reads the file property list of a file and sets up the properties on the generic pathname; editing, loading, or compiling a file will call this function, but you can call it yourself if you want to examine the properties of an arbitrary file.

If the property list text contains no colons, it is an old Emacs format, containing only the value of the **Mode** property.

The following are some of the property names allowed and what they mean.

Mode The editor major mode to be used when editing this file. This is typically the name of the language in which the file is written. The most common values are

	Lisp and Text.
Package	The name of the package into which the file is to be loaded. See chapter 23, page 392 for information about packages.
Base	The number base in which the file is written. This affects both **ibase** and **base**, since it is confusing to have the input and output bases be different. The most common values are 8 and 10.
Lowercase	If the property value is not **nil**, the file is written in lower-case letters and the editor does not translate to upper case. (The editor does not translate to upper case by default unless the user selects "Electric Shift Lock" mode.)
Fonts	The property value is a list of font names, separated by commas. The editor uses this for files which are written in more than one font.
Backspace	If the property value is not **nil**, the file may contain backspaces which cause characters to overprint on each other. The default is to disallow overprinting and display backspaces the way other special function keys are displayed. This default is to prevent the confusion that can be engendered by overstruck text.
Patch-File	If the property value is not **nil**, the file is a "patch file". When it is loaded the system will not complain about function redefinitions. Furthermore, the remembered source file names for functions defined in this file will not be changed to this file, but will be left as whatever file the function came from originally. In a patch file, the **defvar** special-form turns into **defconst**; thus patch files will always reinitialize variables.

You are free to define additional file properties of your own. Howver, you should choose names that are different from all the names above, and from any names likely to be defined by anybody else's programs, to avoid accidental name conflicts.

The following function is the parser for file property lists.

fs:file-read-property-list *pathname stream*

pathname should be a pathname object (*not* a string or namelist, but an actual pathname); usually it is a generic pathname (see section 22.3, page 381). *stream* should be a stream that has been opened and is pointing to the beginning of the file whose file property list is to be parsed. This function reads from the stream until it gets the file property list, parses it, puts corresponding properties onto the property list of *pathname*, and finally sets the stream back to the beginning of the file by using the **:set-pointer** file stream operation (see page 372).

The fundamental way that programs in the Lisp Machine react to the presence of properties on a file's file property list is to examine the property list in the generic pathname. However, there is another way that is more convenient for some applications. File properties can cause special variables to be bound whenever Lisp expressions are being read from the file—when the file is being loaded, when it is being compiled, when it is being read from by the editor, and when its QFASL file is being loaded. This is how the Package and Base properties work. You can also deal with properties this way, by using the following function:

fs:file-property-bindings *pathname*

This function examines the property list of *pathname*, and finds all those property names that have fs:file-property-bindings properties. Each such property name specifies a set of variables to bind, and a set of values to which to bind them. This function returns two values: a list of all the variables, and a list of all the corresponding values. Usually you use this function by calling it on a generic pathname what has had fs:file-read-property-list done on it, and then you use the two returned values as the first two subforms to a progv special form (see page 18). Inside the body of the progv the specified bindings will be in effect.

Usually *pathname* is a generic pathname. It can also be a locative, in which case it is interpreted to be the property list itself.

Of the standard property names, the following ones have fs:file-property-bindings, with the following effects. Package binds the variable package (see page 398) to the package. Base binds the variables base (see page 319) and ibase (see page 322) to the value. Patch-file binds fs:this-is-a-patch-file to the value.

Any properties whose names do not have a fs:file-property-bindings property are ignored completely.

You can also add your own property names that affect bindings. If an indicator symbol has an fs:file-property-bindings property, the value of that property is a function which is called when a file with a file property of that name is going to be read from. The function is given three arguments: the file pathname, the property name, and the property value. It must return two values: a list of variables to be bound and a list of values to bind them to. The function for the Base keyword could have been defined by:

```
(defun (:base file-property-bindings) (file ignore bse)
  (if (not (and (typep bse 'fixnum)
                (> bse 1)
                (< bse 37.)))
      (ferror nil "File ~A has an illegal -*- Base:~s -*-"
                  file bse))
  (values (list 'base 'ibase) (list bse bse)))
```

21.9.3 File Stream Operations

The following messages may be sent to file streams, in addition to the normal I/O messages which work on all streams. Note that several of these messages are useful to send to a file stream which has been closed. Some of these messages use pathnames; refer to chapter 22, page 376 for an explanation of pathnames.

:pathname

Returns the pathname that was opened to get this stream. This may not be identical to the argument to open, since missing components will have been filled in from defaults, and the pathname may have been replaced wholesale if an error occurred in the attempt to open the original pathname.

:truename

Returns the pathname of the file actually open on this stream. This can be different from what :pathname returns because of file links, logical devices, mapping of "newest" version to a particular version number, etc. For an output stream the truename is not meaningful until after the stream has been closed, at least when the file server is an ITS.

:qfaslp

Returns t if the file has a magic flag at the front that says it is a QFASL file, nil if it is an ordinary file.

:length

Returns the length of the file, in bytes or characters. For text files on pdp-10 file servers, this is the number of pdp-10 characters, not Lisp Machine characters. The numbers are different because of character-set translation; see page 316 for a full explanation. For an output stream the length is not meaningful until after the stream has been closed, at least when the file server is an ITS.

:creation-date

Returns the creation date of the file, as a number which is a universal time. See the chapter on the time package (chapter 30, page 493).

:info

Returns a string which contains the version number and creation date of the file. This can be used to tell if the file has been modified between two opens. For an output stream the info is not meaningful until after the stream has been closed, at least when the file server is an ITS.

:set-byte-size *new-byte-size*

This is only allowed on binary ("fixnum mode") file streams. The byte size can be changed to any number of bits from 1 to 16.

:delete &optional (*error-p* t)

Deletes the file open on this stream. For the meaning of *error-p*, see the deletef function. The file doesn't really go away until the stream is closed.

:rename *new-name* &optional (*error-p* t)

Renames the file open on this stream. For the meaning of *error-p*, see the renamef function.

:read-pointer

Returns the current position within the file, in characters (bytes in fixnum mode). For text files on pdp-10 file servers, this is the number of Lisp Machine characters, not pdp-10 characters. The numbers are different because of character-set translation.

:set-pointer *new-pointer*

Sets the reading position within the file to *new-pointer* (bytes in fixnum mode). For text files on pdp-10 file servers, this will not do anything reasonable unless *new-pointer* is 0, because of character-set translation. This operation is for input streams only.

:rewind
This is the same as :set-pointer 0. This operation is for input streams only.

:get-input-buffer &optional *eof*
Returns three values: a buffer array, the index in that array of the next input byte, and a count of the number of bytes remaining in the array. If the end of the file has been reached, returns nil or signals an error, based on the *eof* argument, just like the :tyi message. After reading as many bytes from the array as you care to, you must send the :advance-input-buffer message. This operation is for input streams only. It is a kludge to provide for faster input from files.

:advance-input-buffer &optional *new-pointer*
If *new-pointer* is non-nil, it is the index in the buffer array of the next byte to be read. If *new-pointer* is nil, the entire buffer has been used up. This operation is for input streams only.

File output streams implement the :finish and :force-output messages.

21.10 Accessing Directories

To understand the functions in this section, it helps to have read the following chapter, on *pathnames*.

fs:directory-list *pathname* &rest *options*
Finds all the files that match *pathname* and returns a list with one element for each file. Each element is a list whose car is the pathname of the file and whose cdr is a list of the properties of the file; thus the element is a "disembodied" property list and get may be used to access the file's properties. The car of one element is nil; the properties in this element are properties of the file system as a whole rather than of a specific file.

The matching is done using both host-independent and host-dependent conventions. Any component of *pathname* which is :wild matches anything; all files that match the remaining components of *pathname* will be listed, regardless of their value for the wild component. In addition, there is host-dependent matching. Typically this uses the asterisk character (*) as a wild-card character. A pathname component that consists of just a * matches any value of that component (the same as :wild). A pathname component that contains * and other characters matches any character (on ITS) or any string of characters (on TOPS-20) in the starred positions and requires the specified characters otherwise. Other hosts will follow similar but not necessarily identical conventions.

The *options* are keywords which modify the operation. The following options are currently defined:

:noerror If a file-system error (such as no such directory) occurs during the operation, normally an error will be signalled and the user will be asked to supply a new pathname. However, if :noerror is specified then in the event of an error a string describing the error will be returned as the result of fs:directory-list. This is identical to the :noerror option to open.

:deleted
: This is for TOPS-20 file servers. It specifies that deleted (but not yet expunged) files are to be included in the directory listing.

The properties that may appear in the list of property lists returned by **fs:directory-list** are host-dependent to some extent. The following properties are those that are defined for both ITS and TOPS-20 file servers. This set of properties is likely to be extended or changed in the future.

:length-in-bytes
: The length of the file expressed in terms of the basic units in which it is written (characters in the case of a text file).

:byte-size
: The number of bits in one of those units.

:length-in-blocks
: The length of the file in terms of the file system's unit of storage allocation.

:block-size
: The number of bits in one of those units.

:creation-date
: The date the file was created, as a universal time. See chapter 30, page 493.

:reference-date
: The most recent date that the file was used, as a universal time.

:author
: The name of the person who created the file, as a string.

:not-backed-up
: **t** if the file exists only on disk, **nil** if it has been backed up on magnetic tape.

fs:change-file-properties *pathname error-p* &rest *properties*

Some of the properties of a file may be changed; for instance, its creation date or its author. Exactly which properties may be changed depends on the host file system; a list of the changeable property names is the :settable-properties property of the file system as a whole, returned by **fs:directory-list** as explained above.

fs:change-file-properties changes one or more properties of a file. *pathname* names the file. The *properties* arguments are alternating keywords and values. The *error-p* argument is the same as with **renamef**; if an error occurs and it is **nil** a string describing the error will be returned; if it is **t** a Lisp error will be signalled. If no error occurs, **fs:change-file-properties** returns **t**.

fs:file-properties *pathname* &optional (*error-p* **t**)

Returns a disembodied property list for a single file (compare this to **fs:directory-list**). The car of the returned list is the truename of the file and the cdr is an alternating list of indicators and values. The *error-p* argument is the same as with **renamef**; if an error occurs and it is **nil** a string describing the error will be returned; if it is **t** (the default) a Lisp error will be signalled.

fs:complete-pathname *defaults string type version* &rest *options*

string is a partially-specified file name. (Presumably it was typed in by a user and terminated with the altmode key or the END key to request completion.) fs:complete-pathname looks in the file system on the appropriate host and returns a new, possibly more specific string. Any unambiguous abbreviations are expanded out in a host-dependent fashion.

defaults, *type*, and *version* are the arguments that will be given to fs:merge-pathname-defaults (see page 382) when the user's input is eventually parsed and defaulted.

options are keywords (without following values) which control how the completion will be performed. The following option keywords are allowed:

:deleted	Look for files which have been deleted but not yet expunged.
:read or :in	The file is going to be read. This is the default.
:print or :write or :out	The file is going to be written (i.e. a new version is going to be created).
:old	Look only for files that already exist. This is the default.
:new-ok	Allow either a file that already exists, or a file that does not yet exist. An example of the use of this is the c-X c-F (Find File) command in the editor.

The first value returned is always a string containing a file name; either the original string, or a new, more specific string. The second value returned indicates the success or failure status of the completion. It is nil if an error occurred. One possible error is that the file is on a file system that does not support completion, in which case the original string will be returned unchanged. Other possible second values are :old, which means that the string completed to the name of a file that exists, :new, which means that the string completed to the name of a file which could be created, and nil again, which means that there is no possible completion.

22. Naming of Files

A Lisp Machine generally has access to many file systems. While it may have its own file system on its own disks, usually a community of Lisp Machine users want to have a shared file system accessible by any of the Lisp Machines over a network. These shared file systems can be implemented by any computer that is capable of providing file system service. A file server computer may be a special-purpose computer that does nothing but service file system requests from computers on a network, or it might be an existing time-sharing system.

Programs need to use names to designate files within these file systems. The main difficulty in dealing with names of files is that different file systems have different naming formats for files. For example, in the ITS file system, a typical name looks like:

```
DSK: GEORGE; FOO QFASL
```

with DSK being a device name, GEORGE being a directory name, FOO being the first file name and QFASL being the second file name. However, in TOPS-20, a similar file name is expressed as:

```
PS:<GEORGE>FOO.QFASL
```

It would be unreasonable for each program that deals with file names to be expected to know about each different file name format that exists; in fact, new formats could get added in the future, and existing programs should retain their abilities to manipulate the names.

The functions and flavors described in this chapter exist to solve this problem. They provide an interface through which a program can deal with names of files and manipulate them without depending on anything about their syntax. This lets a program deal with multiple remote file servers simultaneously, using a uniform set of conventions.

22.1 Pathnames

All file systems dealt with by the Lisp Machine are mapped into a common model, in which files are named by something called a *pathname*. A pathname always has six components, described below. These components are the common interface that allows programs to work the same way with different file systems; the mapping of the pathname components into the concepts peculiar to each file system is taken care of by the pathname software. This mapping is described for each file system later in this chapter.

These are the components of a pathname. They will be clarified by an example below.

host	The name of the file system machine on which the file resides.
device	Corresponds to the "device" or "file structure" concept in many host file systems.
directory	The name of a group of related files belonging to a single user or project. Corresponds to the "directory" concept in many host file systems.
name	The name of a group of files which can be thought of as conceptually the "same" file.
type	Corresponds to the "filetype" or "extension" concept in many host file systems. This says what kind of file this is.

version	Corresponds to the "version number" concept in many host file systems. This is a number which increments every time the file is modified.

As an example, consider a Lisp program named CONCH. If it belongs to GEORGE, who uses the FISH machine, the host would be FISH, the device would be the default probably, and the directory would be GEORGE. On this directory would be a number of files related to the CONCH program. The source code for this program would live in a set of files with name CONCH, type LISP, and versions 1, 2, 3, etc. The compiled form of the program would live in files named CONCH with type QFASL; each would have the same version number as the source file that it came from. If the program had a documentation file, it would have type **INFO**.

Note that a pathname is not necessarily the name of a specific file. Rather, it is a way to get to a file; a pathname need not correspond to any file that actually exists, and more than one pathname can refer to the same file. For example, the pathname with a version of "newest" will refer to the same file as a pathname with the same components except a certain number as the version. In systems with links, multiple file names, logical devices, etc. two pathnames that look quite different may really turn out to address the same file. To get from a pathname to a file requires doing a file system operation such as **open**.

A pathname is an instance of a flavor (see chapter 20, page 279); exactly which flavor depends on what the host of the pathname is. If *p* is a pathname, then **(typep *p* 'fs:pathname)** will return **t**. (**fs** is the file-system package.) There are functions for manipulating pathnames, and there are also messages that can be sent to them. These are described later in this chapter.

Two important operations of the pathname system are *parsing* and *merging*. Parsing is the conversion of a string—which might be something typed in by the user when asked to supply the name of a file—into a pathname object. This involves finding out what host the pathname is for, then using the file name syntax conventions of that host to parse the string into the standard pathname components. Merging is the operation which takes a pathname with missing components and supplies values for those components from a set of defaults.

Since each kind of file server can have its own character string representation of names of its files, there has to be a different parser for each of these representations, capable of examining such a character string and figuring out what each component is. The parsers all work differently. How can the parsing operation know which parser to use? The first thing that the parser does is to figure out which host this filename belongs to. A filename character string may specify a host explicitly, by having the name of the host, followed by a colon, either at the beginning or the end of the string. For example, the following strings all specify hosts explicitly:

```
AI: COMMON; GEE WHIZ            ; This specifies host AI.
COMMON; GEE WHIZ AI:            ; So does this.
AI: ARC: USERS1; FOO BAR        ; So does this.
ARC: USERS1; FOO BAR AI:        ; So does this.
EE:PS:<COMMON>GEE.WHIZ.5        ; This specifies host EE.
PS:<COMMON>GEE.WHIZ.5 EE:       ; So does this.
```

If the string does not specify a host explicitly, the parser will assume some particular host is the one in question, and will use the parser for that host's file system. The optional arguments passed to the parsing function (**fs:parse-pathname**) tell it which host to assume. Note: the parser won't be confused by strings starting with "DSK:" or "PS:" because it knows that neither of those is a valid host name. (If some file system's syntax allowed file names that start with the name of a valid host followed by a colon, there could be problems.)

Pathnames, like symbols, are *interned*. This means that there is only one pathname object with a given set of components. If a character string is parsed into components, and some pathname object with exactly those components already exists, then the parser returns the existing pathname object rather than creating a new one. The main reason for this is that a pathname has a property list (see section 5.8, page 71). The system stores properties on pathnames to remember information about the file or family of files to which that pathname refers. So you can parse a character-string that represents a filename, and then look at its property list to get various information known about that pathname. The components of a pathname are never modified once the pathname has been created, just as the print-name of a symbol is never modified. The only thing that can be modified is the property list.

When using property lists of pathnames, you have to be very careful which pathname you use to hold properties, in order to avoid a subtle problem: many different pathnames can refer to the same file, because of things like the :newest component, file system links, multiple naming in the file system, and so on. If you put a property on one of these pathnames because you want to associate some information with the file itself, somebody else might look at another pathname that refers to the same file, and not find the information there. If you really want to associate information with the file itself rather than some particular pathname, you can get a canonical pathname for the file by using the :truename message to a stream opened to that file (see page 372). You may also want to store properties on "generic" pathnames (see section 22.3, page 381).

A pathname can be converted into a string, which is in the file name syntax of its host's file system, except that the name of the host followed by a colon is inserted at the front. **prin1** of a pathname (~S in **format**) prints it like a Lisp object (using the usual "#<" syntax), while **princ** of a pathname (~A in **format**) prints it like a file name of the host file system. The **string** function, applied to a pathname, returns the string that **princ** would print. Thus pathnames may be used as arguments to functions like **string-append**.

Not all of the components of a pathname need to be specified. If a component of a pathname is missing, its value is **nil**. Before a file server can do anything interesting with a file, such as opening the file, all the missing components of a pathname must be filled in from defaults. But pathnames with missing components are often handed around inside the machine, since almost all pathnames typed by users do not specify all the components explicitly. The host is not allowed to be missing from any pathname; since the behavior of a pathname is host-dependent to some extent, it has to know what its host is. All pathnames have host attributes, even if the string being parsed does not specify one explicitly.

A component of a pathname can also be the special symbol :unspecific. This means that the component has been explicitly determined not to be there, as opposed to being missing. One way this can occur is with *generic* pathnames, which refer not to a file but to a whole family of files. The version, and usually the type, of a generic pathname are :unspecific. Another way :unspecific is used has to do with mapping of pathnames into file systems such as ITS that do not have all six components. A component that is really not there will be :unspecific in the pathname. When a pathname is converted to a string, **nil** and :unspecific both cause the component not to appear in the string. The difference occurs in the merging operation, where **nil** will be replaced with the default for that component, while :unspecific is left alone.

A component of a pathname can also be the special symbol :wild. This is only useful when the pathname is being used with a directory primitive such as fs:directory-list (see page 373), where it means that this pathname component matches anything. The printed representation of a pathname usually designates :wild with an asterisk; however, this is host-dependent.

What values are allowed for components of a pathname depends, in general, on the pathname's host. However, in order for pathnames to be usable in a system-independent way certain global conventions are adhered to. These conventions are stronger for the type and version than for the other components, since the type and version are actually understood by many programs, while the other components are usually just treated as something supplied by the user which just needs to be remembered.

The type is always a string (unless it is one of the special symbols nil, :unspecific, and :wild). Many programs that deal with files have an idea of what type they want to use. For example, Lisp source programs are "lisp", compiled Lisp programs are "qfasl", text files are "text", tags files are "tags", etc. Just what characters are allowed in the type, and how many, is system dependent.

The version is either a number (specifically, a positive fixnum), or a special symbol. nil, :unspecific, and :wild have been explained above. :newest refers to the largest version number that exists when reading a file, or that number plus one when writing a new file. :oldest refers to the smallest version number that exists. Some file systems may define other special version symbols, such as :installed for example.

The host is always a string. The Lisp Machine has a fixed list of hosts that it knows about.

The device, directory, and name are more system-dependent. These can be strings (with host-dependent rules on allowed characters and length), or they can be *structured*. A structured component is a list of strings. This is used for file system features such as hierarchical directories. The system is arranged so that programs do not need to know about structured components unless they do host-dependent operations. Giving a string as a pathname component to a host that wants a structured value converts the string to the appropriate form. Giving a structured component to a host that does not understand them converts it to a string by taking the first element and ignoring the rest.

Some host file systems have features that do not fit into this pathname model. For instance, directories might be accessible as files, there might be complicated structure in the directories or names, or there might be relative directories, such as "<" in Multics. These features appear in the parsing of strings into pathnames, which is one reason why the strings are written in host-dependent syntax. Pathnames for hosts with these features are also likely to handle additional messages besides the common ones documented in this chapter, for the benefit of host-dependent programs which want to access those features. However, note that once your program depends on any such features, it will only work for certain file servers and not others; in general, it is a good idea to make your program work just as well no matter what file server is being used.

22.2 Defaults and Merging

When the user is asked to type in a pathname, it is of course unreasonable to require the user to type a complete pathname, containing all components. Instead there are defaults, so that components not specified by the user can be supplied automatically by the system. Each program that deals with pathnames typically has its own set of defaults.

The system defines an object called a *defaults a-list*. Functions are provided to create one, get the default pathname out of one, merge a pathname with one, and store a pathname back into one. A defaults a-list can remember more than one default pathname if defaults are being kept separately for each host; this is controlled by the variable fs:*defaults-are-per-host*. The main primitive for using defaults is the function fs:merge-pathname-defaults (see page 382).

In place of a defaults a-list, you may use just a pathname. Defaulting one pathname from another is useful for cases such as a program that has an input file and an output file, and asks the user for the name of both, letting the unsupplied components of one name default from the other. Unspecified components of the output pathname will come from the input pathname, except that the type should default not to the type of the input but to the appropriate default type for output from this program.

The implementation of a defaults a-list is an association list of host names and default pathnames. The host name nil is special and holds the defaults for all hosts, when defaults are not per-host.

The *merging* operation takes as input a pathname, a defaults a-list (or another pathname), a default type, and a default version, and returns a pathname. Basically, the missing components in the pathname are filled in from the defaults a-list, except that if no type is specified the default type is used, and if no version is specified the default version is used. By default, the default type is :unspecific, meaning that if the input pathname has no type, the user really wants a file with no type. Programs that have a default type for the files they manipulate will supply it to the merging operation. The default version is usually :newest; if no version is specified the newest version in existence should be used. The default type and version can be nil, to preserve the information that they were missing in the input pathname.

The full details of the merging rules are as follows. First, if the pathname explicitly specifies a host and does not supply a device, then the device will be the default file device for that host. Next, if the pathname does not specify a host, device, directory, or name, that component comes from the defaults.

The merging rules for the type and version are more complicated, and depend on whether the pathname specifies a name. If the pathname doesn't specify a name, then the type and version, if not provided, will come from the defaults, just like the other components. However, if the pathname does specify a name, then the type and version are not affected by the defaults. The reason for this is that the type and version "belong to" some other filename, and are unlikely to have anything to do with the new one you are typing in. Finally, if this process leaves the type or version missing, the default type or default version is used (these were inputs to the merging operation).

The effect of all this is that if the user supplies just a name, the host, device, and directory will come from the defaults, but the type and version will come from the default type and default version arguments to the merging operation. If the user supplies nothing, or just a directory, the name, type, and version will come over from the defaults together. If the host's file name syntax provides a way to input a type or version without a name, the user can let the name default but supply a different type or version than the one in the defaults.

The following special variables are parts of the pathname interface that are relevant to defaults.

fs:*defaults-are-per-host* *Variable*
: This is a user customization option intended to be set by a user's LISPM INIT file (see section 31.5, page 506). The default value is nil, which means that each program's set of defaults contains only one default pathname. If you type in just a host name and a colon, the other components of the name will default from the previous host, with appropriate translation to the new host's pathname syntax. If fs:*defaults-are-per-host* is set to t, each program's set of defaults will maintain a separate default pathname for each host. If you type in just a host name and a colon, the last file that was referenced on that host will be used.

fs:*default-pathname-defaults* *Variable*
: This is the default defaults a-list; if the pathname primitives that need a set of defaults are not given one, they use this one. Most programs, however, should have their own defaults rather than using these.

fs:load-pathname-defaults *Variable*
: This is the defaults a-list for the load and qc-file functions. Other functions may share these defaults if they deem that to be an appropriate user interface.

fs:last-file-opened *Variable*
: This is the pathname of the last file that was opened. Occasionally this is useful as a default. Since some programs deal with files without notifying the user, you must not expect the user to know what the value of this symbol is. Using this symbol as a default may cause unfortunate surprises, and so such use is discouraged.

22.3 Generic Pathnames

A generic pathname stands for a whole family of files. The property list of a generic pathname is used to remember information about the family, some of which (such as the package) comes from the -*- line (see section 21.9.2, page 369) of a source file in the family. All types of files with that name, in that directory, belong together. They are different members of the same family; for example, they might be source code, compiled code, and documentation for a program. All versions of files with that name, in that directory, belong together.

The generic pathname of pathname *p* has the same host, device, directory, and name as *p* does. However, it has a version of :unspecific. Furthermore, if the type of *p* is one of the elements of fs:*known-types*, then it has a type of :unspecific; otherwise it has the same type as *p*. The reason that the type of the generic pathname works this way is that in some file systems, like ITS, the type component may actually be part of the file name; ITS files named "DIRECT IONS" and "DIRECT ORY" do not belong together.

The :generic-pathname message to a pathname returns its corresponding generic pathname. See page 385.

fs:*known-types* *Variable*

This is a list of the file types which are "not important"; constructing a generic pathname will strip off the file type if it is in this list. File types not in this list are really part of the name in some sense. The initial list is

```
    ("lisp" "qfasl" "text" nil :unspecific)
```

Some users may need to add to this list.

22.4 Pathname Functions

These functions are what programs use to parse and default file names that have been typed in or otherwise supplied by the user.

fs:parse-pathname *thing* &optional *host defaults*

This turns *thing*, which can be a pathname, a string, a symbol, or a Maclisp-style name list, into a pathname. Most functions which are advertised to take a pathname argument call fs:parse-pathname on it so that they will accept anything that can be turned into a pathname.

This function does *not* do defaulting, even though it has an argument named *defaults*; it only does parsing. The *host* and *defaults* arguments are there because in order to parse a string into a pathname, it is necessary to know what host it is for so that it can be parsed with the file name syntax peculiar to that host. If *thing* does not contain a manifest host name, then if *host* is non-nil, it is the host name to use, as a string. If *thing* is a string, a manifest host name may be at the beginning or the end, and consists of the name of a host followed by a colon. If *host* is nil then the host name is obtained from the default pathname in *defaults*. If *defaults* is not supplied, the default defaults (fs:*default-pathname-defaults*) are used.

Note that if *host* is specified, and *thing* contains a host name, an error is signalled if they are not the same host.

fs:merge-pathname-defaults *pathname* &optional *defaults default-type default-version*

Fills in unspecified components of *pathname* from the defaults, and returns a new pathname. This is the function that most programs should call to process a file name supplied by the user. *pathname* can be a pathname, a string, a symbol, or a Maclisp namelist. The returned value will always be a pathname. The merging rules are documented on page 380.

If *defaults* is a pathname, rather than a defaults a-list, then the defaults are taken from its components. This is how you merge two pathnames (in Maclisp that operation is called mergef).

defaults defaults to the value of fs:*default-pathname-defaults* if unsupplied. *default-type* defaults to :unspecific. *default-version* defaults to :newest.

fs:merge-and-set-pathname-defaults *pathname* &optional *defaults default-type default-version*
This is the same as fs:merge-pathname-defaults except that after it is done the result is stored back into *defaults*. This is handy for programs that have "sticky" defaults. (If *defaults* is a pathname rather than a defaults a-list, then no storing back is done.) The optional arguments default the same way as in fs:merge-pathname-defaults.

This function yields a pathname given its components.

fs:make-pathname &rest *options*
The *options* are alternating keywords and values, which specify the components of the pathname. Missing components default to nil, except the host (all pathnames must have a host). The :defaults option specifies what defaults to get the host from if none is specified. The other options allowed are :host, :device, :structured-device, :directory, :structured-directory, :name, :structured-name, :type, and :version.

These functions are used to manipulate defaults a-lists directly.

fs:make-pathname-defaults
Creates a defaults a-list initially containing no defaults. Asking this empty set of defaults for its default pathname before anything has been stored into it will return the file FOO on the user's home directory on the host he logged in to.

fs:default-pathname &optional *defaults host default-type default-version*
This is the primitive function for getting a default pathname out of a defaults a-list. Specifying the optional arguments *host*, *default-type*, and *default-version* to be non-nil forces those fields of the returned pathname to contain those values.

If fs:*defaults-are-per-host* is nil (its default value), this gets the one relevant default from the a-list. If it is t, this gets the default for *host* if one is specified, otherwise for the host most recently used.

If *defaults* is not specified, the default defaults are used.

This function has an additional optional argument *internal-p*, which users should never supply.

fs:set-default-pathname *pathname* &optional *defaults*
This is the primitive function for updating a set of defaults. It stores *pathname* into *defaults*. If *defaults* is not specified, the default defaults are used.

These functions return useful information.

fs:user-homedir &optional *host reset-p*
Returns the pathname of the logged-in user's home directory on *host*, which defaults to the host the user logged in to. Home directory is a somewhat system-dependent concept, but from the point of view of the Lisp Machine it is the directory where the user keeps personal files such as init files and mail. This function returns a pathname without any name, type, or version component (those components are all nil). If *reset-p* is specified non-nil, the machine the user is logged in to is changed to be *host*.

fs:init-file-pathname *program-name* &optional *host*
Returns the pathname of the logged-in user's init file for the program *program-name*, on the *host*, which defaults to the host the user logged in to. Programs that load init files containing user customizations call this function to find where to look for the file, so that they need not know the separate init file name conventions of each host operating system. The *program-name* "LISPM" is used by the login function.

These functions are useful for poking around.

fs:describe-pathname *pathname*
If *pathname* is a pathname object, this describes it, showing you its properties (if any) and information about files with that name that have been loaded into the machine. If *pathname* is a string, this describes all interned pathnames that match that string, ignoring components not specified in the string. One thing this is useful for is finding what directory a file whose name you remember is in. Giving describe (see page 500) a pathname object will do the same thing as this function will.

fs:pathname-plist *pathname*
Parses and defaults *pathname* then returns the list of properties of that pathname.

fs:*pathname-hash-table* *Variable*
This is the hash table in which pathname objects are interned. Applying the function maphash-equal to this will extract all the pathnames in the world.

22.5 Pathname Messages

This section documents the messages a user may send to a pathname object. Pathnames handle some additional messages which are only intended to be sent by the file system itself, and therefore are not documented here. Someone who wanted to add a new host to the system would need to understand those internal messages. This section also does not document messages which are peculiar to pathnames of a particular host; those would be documented under that host.

:host (to pathname)
:device (to pathname)
:directory (to pathname)
:name (to pathname)
:type (to pathname)
:version (to pathname)

These return the components of the pathname. The returned values can be strings, special symbols, or lists of strings in the case of structured components. The type will always be a string or a symbol. The version will always be a number or a symbol.

:new-device *dev* (to pathname)
:new-structured-device *dev* (to pathname)
:new-directory *dir* (to pathname)
:new-structured-directory *dir* (to pathname)
:new-name *name* (to pathname)
:new-structured-name *name* (to pathname)
:new-type *type* (to pathname)
:new-version *version* (to pathname)

These return a new pathname which is the same as the pathname they are sent to except that the value of one of the components has been changed. The "structured" messages expect a list of strings. If the component is not structured on this host, the first string in the list is used and the rest are ignored. The "unstructured" messages expect a string (or a special symbol), but accept a list of strings if this host allows this component to be structured.

:new-pathname &rest *options* (to pathname)

This returns a new pathname which is the same as the pathname it is sent to except that the values of some of the components have been changed. *options* is a list of alternating keywords and values. The keywords all specify values of pathname components; they are :host, :device, :structured-device, :directory, :structured-directory, :name, :structured-name, :type, and :version.

:generic-pathname (to pathname)

Returns the generic pathname for the family of files of which this pathname is a member. See section 22.3, page 381 for documentation on generic pathnames.

Messages to get a path name string out of a pathname object:

:string-for-printing (to pathname)

Returns a string which is the printed representation of the path name. This is the same as what you get if you princ the pathname or take string of it.

:string-for-wholine (to pathname)

Returns a string which may be compressed in order to fit in the wholine.

:string-for-editor (to pathname)

Returns a string which is the path name with its components rearranged so that the name is first. The editor uses this form to name its buffers.

:string-for-dired (to pathname)

Returns a string to be used by the directory editor. The string contains only the name, type, and version.

:string-for-host (to pathname)
Returns a string which is the path name the way the host file system likes to see it.

Messages to manipulate the property list of a pathname:

:get *indicator* (to pathname)
:getl *list-of-indicators* (to pathname)
:putprop *value indicator* (to pathname)
:remprop *indicator* (to pathname)
:plist (to pathname)
These manipulate the pathname's property list analogously to the functions of the same names (see page 72), which don't (currently) work on instances. Please read the paragraph on page 378 explaining some care you must take in using property lists of pathnames.

22.6 Host File Systems Supported

This section lists the host file systems supported, gives an example of the pathname syntax for each system, and discusses any special idiosyncracies. More host types will no doubt be added in the future.

22.6.1 ITS

An ITS pathname looks like "HOST: DEVICE: DIR; FOO 69". The default device is DSK: but other devices such as ML:, ARC:, DVR:, or PTR: may be used.

ITS does not exactly fit the virtual file system model, in that a file name has two components (FN1 and FN2) rather than three (name, type, and version). Consequently to map any virtual pathname into an ITS filename, it is necessary to choose whether the FN2 will be the type or the version. The rule is that usually the type goes in the FN2 and the version is ignored; however, certain types (LISP and TEXT) are ignored and instead the version goes in the FN2. Also if the type is :unspecific the FN2 is the version.

Given an ITS filename, it is converted into a pathname by making the FN2 the version if it is "<", ">", or a number. Otherwise the FN2 becomes the type. ITS pathnames allow the special version symbols :oldest and :newest, which correspond to "<" and ">" respectively. If a version is specified, the type is always :unspecific. If a type is specified, the version is :newest unless the type is a normally-ignored type (such as LISP) in which case the version is :unspecific so that it does not override the type.

Each component of an ITS pathname is mapped to upper case and truncated to six characters.

Special characters (space, colon, and semicolon) in a component of an ITS pathname can be quoted by prefixing them with right horseshoe (⊃) or equivalence sign (≡). Right horseshoe is the same character code in the Lisp Machine character set as control-Q in the ITS character set.

An ITS pathname can have a structured name, which is a list of two strings, the FN1 and the FN2. In this case there is neither a type nor a version.

An ITS pathname with an FN2 but no FN1 (i.e. a type and/or version but no name) is represented with the placeholder FN1 "⇔", because ITS pathname syntax provides no way to write an FN2 without an FN1 before it.

The ITS init file naming convention is "homedir; user program".

fs:*its-uninteresting-types* *Variable*

The ITS file system does not have separate file types and version numbers; both components are stored in the "FN2". This variable is a list of the file types which are "not important"; files with these types use the FN2 for a version number. Files with other types use the FN2 for the type and do not have a version number. The initial list is

```
("lisp" "text" nil :unspecific)
```

Some users may need to add to this list.

:fn1 (to its-pathname)
:fn2 (to its-pathname)

These two messages return a string which is the FN1 or FN2 host-dependent component of the pathname.

22.6.2 TOPS-20 and Tenex

A TOPS-20 pathname looks like "HOST:DEVICE:<DIRECTORY>NAME.TYPE.VERSION". The default device is PS:.

TOPS-20 pathnames are mapped to upper case. Special characters (including lowercase letters) are quoted with the circle-x (⊗) character, which has the same character code in the Lisp Machine character set as control-V in the TOPS-20 character set.

TOPS-20 pathnames allow the special version symbols :oldest and :newest, which correspond to "..-2" and "..0" respectively.

The directory component of a TOPS-20 pathname may be structured. The directory <FOO.BAR> is represented as the list ("FOO" "BAR").

The TOPS-20 init file naming convention is "<user>program.INIT".

When there is an attempt to display a TOPS-20 file name in the who-line and there isn't enough room to show the entire name, the name is truncated and followed by a center-dot character to indicate that there is more to the name than can be displayed.

Tenex pathnames are almost the same as TOPS-20 pathnames, except that the version is preceeded by a semi-colon instead of a period, the default device is DSK instead of PS, and the quoting requirements are slightly different.

22.6.3 Logical Pathnames

There is another kind of pathname that doesn't correspond to any particular file server. It is called a "logical" pathname, and its host is called a "logical" host. Every logical pathname can be translated into a corresponding "physical" pathname; there is a mapping from logical hosts into physical hosts used to effect this translation.

The reason for having logical pathnames is to make it easy to keep bodies of software on more than one file system. An important example is the body of software that constitutes the Lisp Machine system. Every site has a copy of all of the sources of the programs that are loaded into the initial Lisp environment. Some sites may store the sources on an ITS file system, while others might store them on a TOPS-20. However, there is software that wants to deal with the pathnames of these files in such a way that the software will work correctly no matter which site it is run at. The way this is accomplished is that there is a logical host called SYS, and all pathnames for system software files are actually logical pathnames with host SYS. At each site, SYS is defined as a logical host, but translation will work differently at one site than at another. At a site where the sources are stored on a certain TOPS-20, for example, pathnames of the SYS host will be translated into pathnames for that TOPS-20.

Here is how translation is done. For each logical host, there is a mapping that takes the name of a directory on the logical host, and produces a device and a directory for the corresponding physical host. To translate a logical pathname, the system finds the mapping for that pathname's host and looks up that pathname's directory in the mapping. If the directory is found, a new pathname is created whose host is the physical host, and whose device and directory come from the mapping. The other components of the new pathname are left the same. There is also, for each logical host, a "default device". If the directory is not found in the mapping, then the new pathname will have the same directory name as the old one, and its device will be the "default device" for the logical host.

This means that when you invent a new logical device for a certain set of files, you also make up a set of logical directory names, one for each of the directories that the set of files is stored in. Now when you create the mappings at particular sites, you can choose any physical host for the files to reside on, and for each of your logical directory names, you can specify the actual directory name to use on the physical host. This gives you flexibility in setting up your directory names; if you used a logical directory name called fred and you want to move your set of files to a new file server that already has a directory called fred, being used by someone else, you can translate fred to some other name and so avoid getting in the way of the existing directory. Furthermore, you can set up your directories on each host to conform to the local naming conventions of that host.

fs:add-logical-pathname-host *logical-host physical-host translations*

This creates a new logical host named *logical-host*. Its corresponding "physical" host (that is, the host to which it will forward most operations) is *physical-host*. *logical-host* and *physical-host* should both be strings. *translations* should be a list of translation specifications. Each translation specification should be a list of two strings. The first is the name of a directory on the logical host. The second is a pathname whose device component and directory component are the translation of that directory. The default device for the logical host will be the device of the first translation specification. Example:

```
(fs:add-logical-pathname-host "music" "music-10-a"
      '(("melody" 'ss:<melody>")
        ("doc" "ps:<music-documentation>")))
```

This creates a new logical host called music. If you try to read the file music:doc;manual text 2, you will be re-directed to the file music-10-a:ps:<music-documentation>manual.text.2 (assuming that the host music-10-a is a TOPS-20 system).

:translated-pathname (to fs:logical-pathname)
This converts a logical pathname to a physical pathname. It returns the translated pathname of this instance; a pathname whose :host component is the physical host that corresponds to this instance's logical host.

If this message is sent to a physical pathname, it simply returns itself.

:back-translated-pathname *pathname* (to fs:logical-pathname)
This converts a physical pathname to a logical pathname. *pathname* should be a pathname whose host is the physical host corresponding to this instance's logical host. This returns a pathname whose host is the logical host and whose translation is *pathname*.

An example of how this would be used is in connection with truenames. Given a stream which was obtained by opening a logical pathname,

```
(funcall stream ':pathname)
```

returns the logical pathname that was opened.

```
(funcall stream ':truename)
```

returns the true name of the file that is open, which of course is a pathname on the physical host. To get this in the form of a logical pathname, one would do

```
(funcall (funcall stream ':pathname)
         ':back-translated-pathname
         (funcall stream ':truename))
```

If this message is sent to a physical pathname, it simply returns its argument. Thus the above example will work no matter what kind of pathname was opened to create the stream.

A logical pathname looks like "HOST: DIRECTORY; NAME TYPE VERSION". There is no way to specify a device; parsing a logical pathname always returns a pathname whose device component is :unspecific. This is because devices don't have any meaning in logical pathnames.

The equivalence-sign character (≡) can be used for quoting special characters such as spaces and semicolons. The double-arrow character (↔) can be used as a place-holder for unspecified components. Components are not mapped to upper-case. The :newest, :oldest, and :wild values for versions are specified with the strings ">", "<", and "*" respectively.

There isn't any init file naming convention for logical hosts; you can't log into them. The :string-for-host, :string-for-wholine, :string-for-dired, and :string-for-editor messages are all passed on to the translated pathname, but the :string-for-printing is handled by the fs:logical-pathname flavor itself and shows the logical name.

22.7 Maclisp Conversion

This section briefly discusses how to convert from Maclisp I/O and filename functions to the corresponding but often more general Lisp Machine ones.

The functions load, open, probef, renamef, and deletef are upward compatible. Most of them take optional additional arguments to do additional things, usually connected with error handling. Where Maclisp wants to see a file name in the form of a symbol or a list, the Lisp Machine will accept those or a string or a pathname object. probef returns a pathname or nil rather than a namelist or nil.

load keeps defaults, which it updates from the file name it is given.

The old-I/O functions uread, crunit, etc. do not exist in the Lisp Machine. fasload exists but is a function rather than a special form.

There is a special form, with-open-file, which should replace most calls to open. See page 365.

The functions for manipulating file names themselves are different. The system will accept a namelist as a pathname, but will never create a namelist. mergef is replaced by fs:merge-pathname-defaults. defaultf is replaced by fs:default-pathname or fs:set-default-pathname, depending on whether it is given an argument. namestring is replaced by the :string-for-printing message to a pathname, or the string function. namelist is approximately replaced by fs:parse-pathname. (status udir) and (status homedir) are approximately replaced by fs:user-homedir. The truename function is replaced by the :truename stream operation, which returns a pathname containing the actual name of the file open on that stream. The directory and allfiles functions are replaced by fs:directory-list.

22.8 Examples

The following examples illustrate some of the rules of parsing and merging. They assume that the default host is an ITS host named AI.

If we parse the string "AI:COMMON;NOMEN 5" (by calling fs:parse-pathname), we get back a pathname that prints as #<ITS-PATHNAME "AI: COMMON; NOMEN 5">. Its host is "AI", its device is "DSK" (because of the rule that when you specify a host and don't specify a device, the standard file-storage device for that host is used), its directory is "COMMON", its name is "NOMEN", its type is nonexistent (:unspecific), and its version is 5. Call this pathname *p*.

Parsing just the string "foo" returns a pathname that prints as #<ITS-PATHNAME "AI: FOO">. The host is "AI", the name is "FOO", and all the other components are unspecified, i.e. nil.

If we merge this with *p* (by calling fs:merge-pathname-defaults with this pathname as its first argument and *p* as its second), the result is a pathname that prints as #<ITS-PATHNAME "AI: COMMON; FOO >">. with host "AI", device "DSK", directory "COMMON", name "FOO", type :unspecific, and version :newest. This is because of the rule that when a name is explicitly specified, the type and version of the defaults are ignored. The version, 5, was ignored, and the version of the result came from the *default-version* argument to fs:merge-pathname-defaults, which had the value :newest. The type, similarly, came from the *default-type* argument, which had the value :unspecific.

Parsing "FOO BAR" returns a pathname that prints as #<ITS-PATHNAME "AI: FOO BAR">. It has host "AI", name "FOO", and type "BAR"; the directory is nil and the version is :newest. Merging this with *p* gives a pathname that prints as #<ITS-PATHNAME "AI: COMMON; FOO BAR">; it has host "AI", device "DSK", directory "COMMON", name "FOO", type "BAR", and version :newest. If we ask for the generic pathname of this new pathname, what we get prints exactly the same, but one of its components is different: its version is :unspecific. This difference does not appear in the printed representation because ITS filenames cannot convey both a meaningful type and a meaningful version number at the same time.

23. Packages

23.1 The Need for Multiple Contexts

A Lisp program is a collection of function definitions. The functions are known by their names, and so each must have its own name to identify it. Clearly a programmer must not use the same name for two different functions.

The Lisp Machine consists of a huge Lisp environment, in which many programs must coexist. All of the "operating system", the compiler, the editor, and a wide variety of programs are provided in the initial environment. Furthermore, every program which the user uses during his session must be loaded into the same environment. Each of these programs is composed of a group of functions; apparently each function must have its own distinct name to avoid conflicts. For example, if the compiler had a function named pull, and the user loaded a program which had its own function named pull, the compiler's pull would be redefined, probably breaking the compiler.

It would not really be possible to prevent these conflicts, since the programs are written by many different people who could never get together to hash out who gets the privilege of using a specific name such as pull.

Now, if we are to enable two programs to coexist in the Lisp world, each with its own function pull, then each program must have its own symbol named "pull", because there can't be two function definitions on the same symbol. This means that separate "name spaces"—mappings between names and symbols—must be provided for the two programs. The package system is designed to do just that.

Under the package system, the author of a program or a group of closely related programs identifies them together as a "package". The package system associates a distinct name space with each package.

Here is an example: suppose there are two programs named chaos and arpa, for handling the Chaosnet and Arpanet respectively. The author of each program wants to have a function called get-packet, which reads in a packet from the network (or something). Also, each wants to have a function called allocate-pbuf, which allocates the packet buffer. Each "get" routine first allocates a packet buffer, and then reads bits into the buffer; therefore, each version of get-packet should call the respective version of allocate-pbuf.

Without the package system, the two programs could not coexist in the same Lisp environment. But the package feature can be used to provide a separate name space for each program. What is required is to declare a package named chaos to contain the Chaosnet program, and another package arpa to hold the Arpanet program. When the Chaosnet program is read into the machine, its symbols would be entered in the chaos package's name space. So when the Chaosnet program's get-packet referred to allocate-pbuf, the allocate-pbuf in the chaos name space would be found, which would be the allocate-pbuf of the Chaosnet program—the right one. Similarly, the Arpanet program's get-packet would be read in using the arpa package's name space and would refer to the Arpanet program's allocate-pbuf.

To understand what is going on here, you should keep in mind how Lisp reading and loading works. When a file is gotten into the Lisp Machine, either by being read or by being fasloaded, the file itself obviously cannot contain Lisp objects; it contains printed representations of those objects. When the reader encounters a printed representation of a symbol, it calls intern to look up that string in some name space and find a corresponding symbol. The package system arranges that the correct name space is used whenever a file is loaded.

23.2 The Organization of Name Spaces

We could simply let every name space be implemented as one obarray, e.g. one big table of symbols. The problem with this is that just about every name space wants to include the whole Lisp language: car, cdr, and so on should be available to every program. We would like to share the main Lisp system between several name spaces without making many copies.

Instead of making each name space be one big array, we arrange packages in a tree. Each package has a "superpackage" or "parent", from which it "inherits" symbols. Also, each package has a table, or "obarray", of its own additional symbols. The symbols belonging to a package are simply those in the package's own obarray, followed by those belonging to the superpackage. The root of the tree of packages is the package called global, which has no superpackage. global contains car and cdr and all the rest of the standard Lisp system. In our example, we might have two other packages called chaos and arpa, each of which would have global as its parent. Here is a picture of the resulting tree structure:

```
                   global
                     |
        /----------------------------\
        |                            |
       chaos                        arpa
```

In order to make the sharing of the global package work, the intern function is made more complicated than in basic Lisp. In addition to the string or symbol to intern, it must be told which package to do it in. First it searches for a symbol with the specified name in the obarray of the specified package. If nothing is found there, intern looks at its superpackage, and then at the superpackage's superpackage, and so on, until the name is found or a root package such as global is reached. When intern reaches the root package, and doesn't find the symbol there either, it decides that there is no symbol known with that name, and adds a symbol to the originally specified package.

Since you don't normally want to worry about specifying packages, intern normally uses the "current" package, which is the value of the symbol package. This symbol serves the purpose of the symbol obarray in Maclisp.

Here's how that works in the above example. When the Chaos net program is read into the Lisp world, the current package would be the chaos package. Thus all of the symbols in the Chaosnet program would be interned on the chaos package. If there is a reference to some well known global symbol such as append, intern would look for "append" on the chaos package, not find it, look for "append" on global, and find the regular Lisp append symbol, and return that. If, however, there is a reference to a symbol which the user made up himself (say it is called get-packet), the first time he uses it, intern won't find it on either chaos nor global. So intern will make a new symbol named get-packet, and install it on the chaos package. When get-packet is referred to later in the Chaosnet program, intern will find get-packet on the chaos package.

When the Arpanet program is read in, the current package would be arpa instead of chaos. When the Arpanet program refers to append, it gets the global one; that is, it shares the same one that the Chaosnet program got. However, if it refers to get-packet, it will *not* get the same one the Chaosnet program got, because the chaos package is not being searched. Rather, the arpa and global packages are getting searched. So intern will create a new get-packet and install it on the arpa package.

So what has happened is that there are two get-packets: one for chaos and one for arpa. The two programs are loaded together without name conflicts.

23.3 Shared Programs

Now, a very important feature of the Lisp Machine is that of "shared programs"; if one person writes a function to, say, print numbers in Roman numerals, any other function can call it to print Roman numerals. This contrasts sharply with PDP-10 system programs, in which Roman numerals have been independently reimplemented several times (and the ITS filename parser several dozen times).

For example, the routines to manipulate a robot arm might be a separate program, residing in a package named arm. If we have a second program called blocks (the blocks world, of course) which wanted to manipulate the arm, it would want to call functions which are defined on the arm obarray, and therefore not in blocks's own name space. Without special provision, there would be no way for any symbols not in the blocks name space to be part of any blocks functions.

The colon character (":") has a special meaning to the Lisp reader. When the reader sees a colon preceeded by the name of a package, it will read in the next Lisp object with package bound to that package. The way blocks would call a function named go-up defined in arm would be by asking to call arm:go-up, because "go-up would be interned on the arm package. What arm:go-up means precisely is "the symbol named go-up in the name space of the package arm."

Similarly, if the chaos program wanted to refer to the arpa program's allocate-pbuf function (for some reason), it would simply call arpa:allocate-pbuf.

An important question which should occur at this point is how the names of packages are associated with their obarrays and other data. This is done by means of the "refname-alist" which each package has. This alist associates strings called *reference names* or *refnames* with the packages they name. Normally, a package's refname-alist contains an entry for each subpackage, associating the subpackage with its name. In addition, every package has its own name defined as a refname, referring to itself. However, the user can add any other refnames, associating them with any packages he likes. This is useful when multiple versions of a program are loaded into different packages. Of course, each package inherits its superpackage's refnames just as it does symbols.

In our example, since arm is a subpackage of global, the name arm is on global's refname-alist, associated with the arm package. Since blocks is also a subpackage of global, when arm:go-up is seen the string "arm" is found on global's refname alist.

When you want to refer to a symbol in a package which you and your superpackages have no refnames for—say, a subpackage named foo of a package named bar which is under global—you can use multiple colons. For example, the symbol finish in that package foo could be referred to as foo:bar:finish. What happens here is that the second name, bar, is interpreted as a refname in the context of the package foo.

23.4 Declaring Packages

Before any package can be referred to or loaded, it must be declared. This is done with the special form package-declare, which tells the package system all sorts of things, including the name of the package, the place in the package hierarchy for the new package to go, its estimated size, and some of the symbols which belong in it.

Here is a sample declaration:

```
(package-declare foo global 1000
        ()
        (shadow array-push adjust-array-size)
        (extern foo-entry))
```

What this declaration says is that a package named foo should be created as an inferior of global, the package which contains advertised global symbols. Its obarray should initially be large enough to hold 1000 symbols, though it will grow automatically if that isn't enough. Unless there is a specific reason to do otherwise, you should make all of your packages direct inferiors of global. The size you give is increased slightly to be a good value for the hashing algorithm used.

After the size comes the "file-alist", which is given as () in the example. This is an obsolete feature which is not normally used. The "system"-defining facilities should be used instead. See chapter 24, page 406.

Finally, the foo package "shadows" array-push and adjust-array-size, and "externs" foo-entry. What shadowing means is that the foo package should have its own versions of those symbols, rather than inheriting its superpackage's versions. Symbols by these names will be added to the foo package even though there are symbols on global already with those names. This allows the foo package to redefine those functions for itself without redefining them in the global package for everyone else. What externing means is that the foo package is allowed to redefine foo-entry as inherited from the global package, so that it *is* redefined for everybody. If foo attempts to redefine a function such as car which is present in the global package but neither shadowed nor externed, confirmation from the user will be requested.

Note that externing doesn't actually put any symbols into the global package. It just asserts permission to redefine symbols already there. This is deliberate; the intent is to enable the maintainers of the global package to keep control over what symbols are present in it. Because inserting a new symbol into the global package can cause trouble to unsuspecting programs which expect that symbol to be private, this is not supposed to be done in a decentralized manner by programs written by one user and used by another unsuspecting user. Here is an example of the trouble that could be caused: if there were two user programs, each with a function named move-square, and move-square were put on the global package, all of a sudden the two functions would share the same symbol, resulting in a name conflict. While all the definitions of the functions in global are actually supplied by subpackages which extern them (global contains no files of its own), the list of symbol names is centralized in one place, the file "AI: LISPM2; GLOBAL >", and this file is not changed without notifying everyone, and updating the documentation in this manual.

Certain other things may be found in the declarations of various internal system packages. They are arcane and needed only to compensate for the fact that parts of those packages are actually loaded before the package system is. They should not be needed by any user package.

Your package declarations should go into separate files containing only package declarations. Group them however you like, one to a file or all in one file. Such files can be read with load. It doesn't matter what package you load them into, so use user, since that has to be safe.

If the declaration for a package is read in twice, no harm is done. If you edit the size to replace it with a larger one, the package will be expanded. At the moment, however, there is no way to change the list of shadowings or externals; such changes will be ignored. Also, you can't change the superpackage. If you edit the superpackage name and read the declaration in again, you will create a new, distinct package without changing the old one.

package-declare *Macro*

The package-declare macro is used to declare a package to the package system. Its form is:

(package-declare *name superpackage size*
file-alist option-1 option-2 . . .)

The interpretation of the declaration is complicated; see section 23.4, page 395.

describe-package *package-name*

(describe-package *package-name*) is equivalent to (describe (pkg-find-package *package-name*)); that is, it describes the package whose name is *package-name*.

23.5 Packages and Writing Code

The unsophisticated user need never be aware of the existence of packages when writing his programs. He should just load all of his programs into the package user, which is also what console type-in is interned in. Since all the functions which users are likely to need are provided in the global package, which is user's superpackage, they are all available. In this manual, functions which are not on the global package are documented with colons in their names, so typing the name the way it is documented will work.

However, if you are writing a generally useful tool, you should put it in some package other than user, so that its internal functions will not conflict with names other users use. Whether for this reason or for any other, if you are loading your programs into packages other than user there are special constructs that you will need to know about.

One time when you as the programmer must be aware of the existence of packages is when you want to use a function or variable in another package. To do this, write the name of the package, a colon, and then the name of the symbol, as in eine:ed-get-defaulted-file-name. You will notice that symbols in other packages print out that way, too. Sometimes you may need to refer to a symbol in a package whose superior is not global. When this happens, use multiple colons, as in foo:bar:ugh, to refer to the symbol ugh in the package named bar which is under the package named foo.

Another time that packages intrude is when you use a "keyword": when you check for eqness against a constant symbol, or pass a constant symbol to someone else who will check for it using eq. This includes using the symbol as either argument to get. In such cases, the usual convention is that the symbol should reside in the user package, rather than in the package with which its meaning is associated. To make it easy to specify user, a colon before a symbol, as in :select, is equivalent to specifying user by name, as in user:select. Since the user package has no subpackages, putting symbols into it will not cause name conflicts.

Why is this convention used? Well, consider the function make-array, which takes one required argument followed by any number of keyword arguments. For example,

```
(make-array 100 'leader-length 10 'type art-string)
```

specifies, after the first required argument, two options with names leader-length and type and values 10 and art-string. The file containing this function's definition is in the system-internals package, but the function is available to everyone without the use of a colon prefix because the symbol make-array is itself inherited from global. But all the keyword names, such as type, are short and should not have to exist in global. However, it would be a shame if all callers of make-array had to specify system-internals: before the name of each keyword. After all, those callers can include programs loaded into user, which should by rights not have to know about packages at all. Putting those keywords in the user package solves this problem. The correct way to type the above form would be

```
(make-array 100 ':leader-length 10 ':type art-string)
```

Exactly when should a symbol go in user? At least, all symbols which the user needs to be able to pass as an argument to any function in global must be in user if they aren't themselves in global. Symbols used as keywords for arguments by any function should usually be in user, to keep things consistent. However, when a program uses a specific property name to associate its own internal memoranda with symbols passed in from outside, the property name should belong to the program's package, so that two programs using the same property name in that way don't conflict.

23.6 Shadowing

Suppose the user doesn't like the system nth function; he might be a former Interlisp user, and expect a completely different meaning from it. Were he to say (defun nth ---) in his program (call it snail) he would clobber the global symbol named "nth", and so affect the "nth" in everyone else's name space. (Actually, if he had not "externed" the symbol "nth", the redefinition would be caught and the user would be warned.)

In order to allow the snail package to have its own (defun nth ---) without interfering with the rest of the Lisp environment, it must "shadow" out the global symbol "nth" by putting a new symbol named "nth" on its own obarray. Normally, this is done by writing (shadow nth) in the declaration of the snail package. Since intern looks on the subpackage's obarray before global, it will find the programmer's own nth, and never the global one. Since the global one is now impossible to see, we say it has been "shadowed."

Having shadowed nth, if it is sometimes necessary to refer to the global definition, this can be done by writing global:nth. This works because the refname global is defined in the global package as a name for the global package. Since global is the superpackage of the snail package, all refnames defined by global, including "global", are available in snail.

23.7 Packages and Interning

The function intern allows you to specify a package as the second argument. It can be specified either by giving the package object itself, or by giving a string or symbol which is the name of the package. intern returns three values. The first is the interned symbol. The second is t if the symbol is old (was already present, not just added to the obarray). The third is the package in which the symbol was actually found. This can be either the specified package or one of its superiors.

When you don't specify the second argument to intern, the current package, which is the value of the symbol package, is used. This happens, in particular, when you call read. Bind the symbol package temporarily to the desired package, before calling things which call intern, when you want to specify the package. When you do this, the function pkg-find-package, which converts a string into the package it names, may be useful. While most functions that use packages will do this themselves, it is better to do it only once when package is bound. The function pkg-goto sets package to a package specified by a string. You shouldn't usually need to do this, but it can be useful to "put the keyboard inside" a package when you are debugging.

package *Variable*

The value of package is the current package; many functions which take packages as optional arguments default to the value of package, including intern and related functions.

pkg-goto &optional *pkg*

pkg may be a package or the name of a package. *pkg* is made the current package. It defaults to the user package.

pkg-bind *pkg body...* *Macro*

pkg may be a package or a package name. The forms of the *body* are evaluated sequentially with the variable package bound to the package named by *pkg*.
Example:

```
(pkg-bind "zwei"
          (read-from-string function-name))
```

There are actually four forms of the **intern** function: regular **intern**, **intern-soft**, **intern-local**, and **intern-local-soft**. **-soft** means that the symbol should not be added to the package if there isn't already one; in that case, all three values are **nil**. **-local** means that the superpackages should not be searched. Thus, **intern-local** can be used to cause shadowing. **intern-local-soft** is a good low-level primitive for when you want complete control of what to search and when to add symbols. All four forms of **intern** return the same three values, except that the **soft** forms return **nil nil nil** when the symbol isn't found.

intern *string* &optional (*pkg* **package**)

intern searches *pkg* and its superpackages sequentially, looking for a symbol whose print-name is equal to *string*. If it finds such a symbol, it returns three values: the symbol, **t**, and the package on which the symbol is interned. If it does not find one, it creates a new symbol with a print name of *string*, interns it into the package *pkg*, and returns the new symbol, **nil**, and *pkg*.

If *string* is not a string but a symbol, **intern** searches for a symbol with the same print-name. If it doesn't find one, it interns *string*—rather than a newly-created symbol—in *pkg* (even if it is also interned in some other package) and returns it.

intern-local *string* &optional (*pkg* **package**)

intern searches *pkg* (but *not* its superpackages), looking for a symbol whose print-name is equal to *string*. If it finds such a symbol, it returns three values: the symbol, **t**, and *pkg* If it does not find one, it creates a new symbol with a print name of *string*, and returns the new symbol, **nil**, and *pkg*.

If *string* is not a string but a symbol, and no symbol with that print-name is already interned in *pkg*, **intern-local** interns *string*—rather than a newly-created symbol—in *pkg* (even if it is also interned in some other package) and returns it.

intern-soft *string* &optional (*pkg* **package**)

intern searches *pkg* and its superpackages sequentially, looking for a symbol whose print-name is equal to *string*. If it finds such a symbol, it returns three values: the symbol, **t**, and the package on which the symbol is interned. If it does not find one, it returns **nil**, **nil**, and **nil**.

intern-local-soft *string* &optional (*pkg* **package**)

intern searches *pkg* (but *not* its superpackages), looking for a symbol whose print-name is equal to *string*. If it finds such a symbol, it returns three values: the symbol, **t**, and *pkg* If it does not find one, it returns **nil**, **nil**, and **nil**.

Each symbol remembers which package it belongs to. While you can intern a symbol in any number of packages, the symbol will only remember one: normally, the first one it was interned in, unless you clobber it. This package is available as (**symbol-package** *symbol*). If the value is **nil**, the symbol believes that it is uninterned.

The printer also implicitly uses the value of **package** when printing symbols. If slashification is on, the printer tries to print something such that if it were given back to the reader, the same object would be produced. If a symbol which is not in the current name space were just printed as its print name and read back in, the reader would intern it on the wrong package, and return the wrong symbol. So the printer figures out the right colon prefix so that if the symbol's printed representation were read back in to the same package, it would be interned correctly. The prefix is only printed if slashification is on, i.e. prin1 prints it and princ does not.

remob *symbol* &optional *package*

remob removes *symbol* from *package* (the name means "REMove from OBarray"). *symbol* itself is unaffected, but intern will no longer find it on *package*. remob is always "local", in that it removes only from the specified package and not from any superpackages. It returns t if the symbol was found to be removed. *package* defaults to the contents of the symbol's package cell, the package it is actually in. (Sometimes a symbol can be in other packages also, but this is unusual.)

symbol-package *symbol*

Returns the contents of *symbol*'s package cell, which is the package which owns *symbol*, or nil if *symbol* is uninterned.

package-cell-location *symbol*

Returns a locative pointer to *symbol*'s package cell. It is preferable to write

```
(locf (symbol-package symbol))
```

rather than calling this function explicitly.

mapatoms *function* &optional (*package* package) (*superiors-p* t)

function should be a function of one argument. mapatoms applies *function* to all of the symbols in *package*. If *superiors-p* is t, then the function is also applied to all symbols in *package*'s superpackages. Note that the function will be applied to shadowed symbols in the superpackages, even though they are not in *package*'s name space. If that is a problem, *function* can try applying intern in *package* on each symbol it gets, and ignore it if it is not eq to the result of intern; this measure is rarely needed.

mapatoms-all *function* &optional (*package* "global")

function should be a function of one argument. mapatoms-all applies *function* to all of the symbols in *package* and all of *package*'s subpackages. Since *package* defaults to the global package, this normally gets at all of the symbols in all packages. It is used by such functions as apropos and who-calls (see page 499)

Example:

```
(mapatoms-all
  (function
    (lambda (x)
      (and (alphalessp 'z x)
           (print x)))))
```

pkg-create-package *name* &optional (*super* package) (*size* 200)

pkg-create-package creates and returns a new package. Usually packages are created by package-declare, but sometimes it is useful to create a package just to use as a hash table for symbols, or for some other reason.

If *name* is a list, its first element is taken as the package name and the second as the program name; otherwise, *name* is taken as both. In either case, the package name and program name are coerced to strings. *super* is the superpackage for this package; it may be nil, which is useful if you only want the package as a hash table, and don't want it to interact with the rest of the package system. *size* is the size of the package; as in package-declare it is rounded up to a "good" size for the hashing algorithm used.

pkg-kill *pkg*

pkg may be either a package or the name of a package. The package should have a superpackage and no subpackages. pkg-kill takes the package off its superior's subpackage list and refname alist.

pkg-find-package *x* &optional (*create-p* nil) (*under* "global")

pkg-find-package tries to interpret *x* as a package. Most of the functions whose descriptions say "... may be either a package or the name of a package" call pkg-find-package to interpret their package argument.

If *x* is a package, pkg-find-package returns it. Otherwise it should be a symbol or string, which is taken to be the name of a package. The name is looked up on the refname alists of package and its superpackages, the same as if it had been typed as part of a colon prefix. If this finds the package, it is returned. Otherwise, *create-p* controls what happens. If *create-p* is nil, an error is signalled. If *create-p* is :find, nil is returned. If *create-p* is :ask the user is asked whether to create it. Otherwise, a new package is created, and installed as an inferior of *under*.

A package is implemented as a structure, created by defstruct. The following accessor macros are available on the global package:

pkg-name	The name of the package, as a string.
pkg-refname-alist	The refname alist of the package, associating strings with packages.
pkg-super-package	The superpackage of the package.

23.8 Status Information

The current package—where your type-in is being interned—is always the value of the symbol package. A package is a named structure which prints out nicely, so examining the value of package is the best way to find out what the current package is. (It is also displayed in the who-line.) Normally, it should be user, except when inside compilation or loading of a file belonging to some other package.

To get more information on the current package or any other, use the function describe-package. Specify either a package object or a string which is a refname for the desired package as the argument. This will print out everything except a list of all the symbols in the package. If you want *that*, use (mapatoms 'print *package* nil). describe of a package will call describe-package.

23.9 Packages, Loading, and Compilation

It's obvious that every file has to be loaded into the right package to serve its purpose. It may not be so obvious that every file must be compiled in the right package, but it's just as true. Luckily, this usually happens automatically.

When you have mentioned a file in a package's file-alist, requesting to compile that file with qc-file or loading it with load automatically selects that package to perform the operation.

The system can get the package of a source file from its "file property list" (see section 21.9.2, page 369). For instance, you can put at the front of your file a line such as "; -*- Mode:Lisp; Package:System-Internals -*-". The compiler puts the package name into the QFASL file for use when it is loaded. If a file is not mentioned in a package's file-alist and doesn't have such a package specification in it, the system loads it into the current package, and tells you what it did.

23.10 Subpackages

Usually, each independent program occupies one package, which is directly under global in the hierarchy. But large programs, such as Macsyma, are usually made up of a number of sub-programs, which are maintained by a small number of people. We would like each sub-program to have its own name space, since the program as a whole has too many names for anyone to remember. So, we can make each sub-program into its own package. However, this practice requires special care.

It is likely that there will be a fair number of functions and symbols which should be shared by all of the sub-programs of Macsyma. These symbols should reside in a package named macsyma, which would be directly under global. Then, each part of macsyma (which might be called sin, risch, input, and so on) would have its own package, with the macsyma package as its superpackage. To do this, first declare the macsyma package, and then declare the risch, sin, etc. packages, specifying macsyma as the superpackage for each of them. This way, each sub-program gets its own name space. All of these declarations would probably be together in a file called something like "macpkg".

However, to avoid a subtle pitfall, it is necessary that the macsyma package itself contain no files; only a set of symbols specified at declaration time. This list of symbols is specified using shadow in the declaration of the macsyma package. At the same time, the file-alist specified in the declaration must be nil (otherwise, you will not be allowed to create the subpackages). The symbols residing in the macsyma package can have values and definitions, but these must all be supplied by files in macsyma's subpackages (which must "extern" those symbols as necessary). Note that this is exactly the same treatment that global receives: all its functions are actually defined in files which are loaded into system-internals (si), compiler, etc.

To demonstrate the full power and convenience of this scheme, suppose there were a second huge program called owl which also had a subprogram called input (which, presumably, does all of the inputting for owl), and one called database. Then a picture of the hierarchy of packages would look like this:

```
                          global
                            |
                 /-----------------------------------\
                 |                                   |
              macsyma                               owl
                 |                                   |
   ------------------------------       -----------------------

   | | |      |       |       |          |         |          | | |
 (others)   risch    sin    input      input   database    (others)
```

Now, the risch program and the sin program both do integration, and so it would be natural for each to have a function called integrate. From inside sin, sin's integrate would be referred to as "integrate" (no prefix needed), while risch's would be referred to as "risch:integrate". Similarly, from inside risch, risch's own integrate would be called "integrate", whereas sin's would be referred to as "sin:integrate".

If sin's integrate were a recursive function, the implementor would be referring to it from within sin itself, and would be happy that he need not type out "sin:integrate" every time; he can just say "integrate".

From inside the macsyma package or any of its other sub-packages, the two functions would be referred to as "sin:integrate" and as "risch:integrate". From anywere else in the hierarchy, they would have to be called "macsyma:sin:integrate" and "macsyma:risch:integrate".

Similarly, assume that each of the input packages has a function called get-line. From inside macsyma or any of macsyma's subprograms (other than input), the relevant function would be called input:get-line, and the irrelevant one owl:input:get-line. The converse is true for owl and its sub-programs. Note that there is no problem arising from the fact that both owl and macsyma have subprograms of the same name (input).

You might also want to put Macsyma's get-line function on the macsyma package. Then, from anywehere inside Macsyma, the function would be called get-line; from the owl package and subpackages it could be referred to as macsyma:get-line.

23.11 Initialization of the Package System

This section describes how the package system is initialized when generating a new software release of the Lisp Machine system; none of this should affect users.

When the world begins to be loaded, there is no package system. There is one "obarray", whose format is different from that used by the package system. After sufficiently much of the Lisp environment is present for it to be possible to initialize the package system, that is done. At that time, it is necessary to split the symbols of the old-style obarray up among the various initial packages.

The first packages created by initialization are the most important ones: global, system, user, and system-internals. All of the symbols already present are placed in one of those packages. By default, a symbol goes into system-internals. Only those placed on special lists go into one of the others. These lists are the file "AI: LISPM2; GLOBAL >" of symbols which belong in global, and the file "AI: LISPM2; SYSTEM >" of symbols which go in system.

After the four basic packages exist, the package system's definition of intern is installed, and packages exist. Then, the other initial packages format, compiler, zwei, etc. are declared and loaded in almost the normal manner. The exception is that a few of the symbols present before packages exist really belong in one of these packages. Their package declarations contain calls to forward and borrow, which exist only for this purpose and are meaningful only in package declarations, and are used to move the symbols as appropriate. These declarations are kept in the file "AI: LISPM; PKGDCL >".

globalize *symbol* &optional (*package* "global")

Sometimes it will be discovered that a symbol which ought to be in global is not there, and the file defining it has already been loaded, thus mistakenly creating a symbol with that name in a package which ought just to inherit the one from global. When this happens, you can correct the situation by doing (globalize "*symbol-name*"). This function creates a symbol with the desired name in global, merges whatever value, function definition, and properties can be found on symbols of that name together into the new symbol (complaining if there are conflicts), and forwards those slots of the existing symbols to the slots of the new one using one-q-forward pointers, so that they will appear to be one and the same symbol as far as value, function definition, and property list are concerned. They cannot all be made eq to each other, but globalize does the next-best thing: it takes an existing symbol from user, if there is one, to put it in global. Since people who check for eq are normally supposed to specify user anyway, they will not perceive any effect from moving the symbol from user into global.

If globalize is given a symbol instead of a string as argument, the exact symbol specified is put into global. You can use this when a symbol in another package, which should have been inherited from global, is being checked for with eq—as long as there are not *two* different packages doing so. But, if the symbol is supposed to be in global, there usually should not be.

If the argument *package* is specified, then the symbol is moved into that package from all its subpackages, rather than into global.

23.12 Initial Packages

The initially present packages include:

global	Contains advertised global functions.
user	Used for interning the user's type-in. Contains all keyword symbols.
sys or system	Contains internal global symbols used by various system programs. global is for symbols global to the Lisp language, while system is for symbols global to the Lisp Machine "operating system".
si or system-internals	Contains subroutines of many advertised system functions. si is a subpackage of sys.

compiler	Contains the compiler. **compiler** is a subpackage of **sys.**
zwei	Contains the editor.
chaos	Contains the Chaosnet controller.
tv	Contains the window system.
format	Contains the function **format** and its associated subfunctions.

There are quite a few others, it would be pointless to list them all.

Packages which are used for special sorts of data:

fonts	Contains the names of all fonts.
format	Contains the keywords for **format**, as well as the code.

Here is a picture depicting the initial package hierarchy:

```
                              global
                                |
    /----------------------------------------------------\
    |      |      |           |           |          |        |       |
   user  zwei   chaos      system        tv       format   fonts   (etc)
                              |
                       /-------------\
                       |             |
              system-internals    compiler
```

24. Maintaining Large Systems

When a program gets large, it is often desirable to split it up into several files. One reason for this is to help keep the parts of the program organized, to make things easier to find. It's also useful to have the program broken into small pieces that are more convenient to edit and compile. It is particularly important to avoid the need to recompile all of a large program every time any piece of it changes; if the program is broken up into many files, only the files that have changes in them need to be recompiled.

The apparent drawback to splitting up a program is that more "commands" are needed to manipulate it. To load the program, you now have to load several files separately, insead of just loading one file. To compile it, you have to figure out which files need compilation, by seeing which have been edited since they were last compiled, and then you have to compile those files.

What's even more complicated is that files can have interdependencies. You might have a file called "DEFS" that contains some macro definitions (or flavor or structure definitions), and functions in other files might use those macros. This means that in order to compile any of those other files, you must first load the file "DEFS" into the Lisp environment, so that the macros will be defined and can be expanded at compile time. You'd have to remember this whenever you compile any of those files. Furthermore, if "DEFS" has changed, other files of the program might need to be recompiled because the macros might have changed and need to be re-expanded.

This chapter describes the *system* facility, which takes care of all these things for you. The way it works is that you define a set of files to be a *system*, using the **defsystem** special form, described below. This system definition says which files make up the system, which ones depend on the presence of others, and so on. You put this system definition into its own little file, and then all you have to do is load that file and the Lisp environment will know about your system and what files are in it. You can then use the **make-system** function (see page 411) to load in all the files of the system, or recompile all the files that need compiling, and so on.

The system facility is very general and extensible. This chapter explains how to use it and how to extend it. This chapter also explains the *patch* facility, which lets you conveniently update a large program with incremental changes, and how to save away Lisp environments in disk partitions.

24.1 Defining a System

defsystem *name* (*keyword args...*)... *Special Form*

Defines a system named *name*. The options selected by the keywords are explained in detail later. In general, they fall into two categories: properties of the system and *transformations*. A transformation is an operation such as compiling or loading which takes one or more files and does something to them. The simplest system is a set of files and a transformation to be performed on them.

Here are a few examples.

```
(defsystem mysys
  (:compile-load ("AI: GEORGE; PROG1" "AI: GEORG2; PROG2")))

(defsystem zmail
  (:name "ZMail")
  (:pathname-default "AI: ZMAIL;")
  (:package zwei)
  (:module defs "DEFS")
  (:module mult "MULT" :package tv)
  (:module main ("TOP" "COMNDS" "MAIL" "USER" "WINDOW"
                 "FILTER" mult "COMETH"))
  (:compile-load defs)
  (:compile-load main (:fasload defs)))

(defsystem bar
  (:module reader-macros "RDMAC")
  (:module other-macros "MACROS")
  (:module main-program "MAIN")
  (:compile-load reader-macros)
  (:compile-load other-macros (:fasload reader-macros))
  (:compile-load main-program (:fasload reader-macros
                                        other-macros)))
```

The first example defines a new *system* called mysys, which consists of two files, both of which are to be compiled and loaded. The second example is somewhat more complicated. What all the options mean will be specified shortly, but the primary difference is that there is a file defs which must be loaded before the rest of the files (main) can be compiled. The final example has two levels of dependency. reader-macros must be compiled and loaded before other-macros can be compiled. Both reader-macros and other-macros must then be loaded before main-program can be compiled.

The defsystem options other than transformations are:

:name Specifies a "pretty" version of the name for the system, for use in printing.

:short-name
: Specified an abbreviated name used in constructing disk label comments and in patch file names for some file systems.

:component-systems
: Specifies the names of other systems used to make up this system. Performing an operation on a system with component systems is equivalent to performing the same operation on all the individual systems. The format is (:component-systems *names...*).

:package
: Specifies the package in which transformations are performed. A package specified here will override one in the -*- line of the file in question.

:pathname-default
: Gives a local default within the definition of the system for strings to be parsed into pathnames. Typically this specifies the directory, when all the files of a system are on the

same directory.

:patchable

Makes the system be a patchable system (see section 24.7, page 416). An optional argument specifies the directory to put patch files in. The default is the :pathname-default of the system.

:initial-status

Specifies what the status of the system should be when make-system is used to create a new major version. The default is :experimental. See section 24.7.5, page 421 for further details.

:not-in-disk-label

Make a patchable system not appear in the disk label comment. This should probably never be specified for a user system. It is used by patchable systems internal to the main Lisp system, to avoid cluttering up the label.

:module

Allows assigning a name to a set of files within the system. This name can then be used instead of repeating the filenames. The format is (:module *name files options*...). *files* is a *module-specification*, which can be any of the following:

a string This is a file name.

a symbol

This is a module name. It stands for all of the files which are in that module of this system.

an *external module component*

This is a list of the form (*system-name module-names*...), to specify modules in another system. It stands for all of the files which are in all of those modules.

a list of *module components*

A module component is any of the above, or the following:

a list of file names

This is used in the case where the names of the input and output files of a transformation are not related according to the standard naming conventions, for example when a QFASL file has a different name or resides on a different directory than the source file. The file names in the list are used from left to right, thus the first name is the source file. Each file name after the first in the list is defaulted from the previous one in the list.

To avoid syntactic ambiguity, this is allowed as a module component but not as a module specification.

The currently defined options for the :module clause are

:package Overrides any package specified for the whole system for transformations performed on just this module.

In the second defsystem example above, there are three modules. The first two each have only one file, and the third one (main) is made up both of files and another module. To take examples of the other possibilities,

```
(:module prog (("AI: GEORGE; PROG" "AI: GEORG2; PROG")))
(:module foo (defs (zmail defs)))
```

The prog module consists of one file, but it lives in two directories, GEORGE and GEORG2. If this were a Lisp program, that would mean that the file "AI: GEORGE; PROG >" would be compiled into "AI: GEORG2; PROG QFASL". The foo module consists of two other modules, the defs module in the same system, and the defs module in the zmail system. It is not generally useful to compile files that belong to other systems, thus this foo module would not normally be the subject of a transformation. However, *dependencies* (defined below) use modules and need to be able to refer to (depend on) modules of other systems.

24.2 Transformations

Transformations are of two types, simple and complex. A simple transformation is a single operation on a file, such as compiling it or loading it. A complex transformation takes the output from one transformation and performs another transformation on it, for example, loading the results of compilation.

The general format of a simple transformation is (*name input dependencies condition*). *input* is usually a module specification or another transformation whose output is used. The transformation *name* is to be performed on all the files in the module, or all the output files of the other transformation.

dependencies and *condition* are optional.

dependencies is a *transformation specification*, either a list (*transformation-name module-names...*), or a list of such lists. A *module-name* is either a symbol which is the name of a module in the current system, or a list (*system-name module-names...*). A dependency declares that all of the indicated transformations must be performed on the indicated modules before the current transformation itself can take place. Thus in the zmail example above, the defs module must have the :fasload transformation performed on it before the :compile transformation can be performed on main.

The dependency has to be a tranformation that was explicitly specified as a transformation in the system definition, not just an action that might have been performed by anything. That is, if you have a dependency (:fasload foo), it means that (fasload foo) is a tranformation of your system and you depend on that tranformation; it does not simply mean that you depend on foo's being loaded. Furthermore, it doesn't work if (:fasload foo) was an implicit piece of another tranformation. For example, the following is correct and will work:

```
(defsystem foo
  (:module foo "FOO")
  (:module bar "BAR")
  (:compile-load (foo bar)))
```

but this doesn't work:

```
(defsystem foo
  (:module foo "FOO")
  (:module bar "BAR")
  (:module blort "BLORT")
  (:compile-load (foo bar))
  (:compile-load blort (:fasload foo)))
```

because foo's :fasload is not mentioned explicitly (i.e. at top level) but is only implicit in the (:compile-load (foo bar)). One must instead write:

```
(defsystem foo
  (:module foo "FOO")
  (:module bar "BAR")
  (:module blort "BLORT")
  (:compile-load foo)
  (:compile-load bar)
  (:compile-load blort (:fasload foo)))
```

condition is a predicate which specifies when the transformation should take place. Generally it defaults according to the type of the transformation. Conditions are discussed further on page 415.

The defined simple transformations are:

:fasload
: Calls the fasload function to load the indicated files, which must be QFASL files. The *condition* defaults to si:file-newer-than-installed-p, which is t if a newer version of the file exists on the file computer than was read into the current environment.

:readfile
: Calls the readfile function to read in the indicated files. Use this for files that are not to be compiled. *condition* defaults to si:file-newer-than-installed-p.

:compile
: Calls the qc-file function to compile the indicated files. *condition* defaults to si:file-newer-than-file-p which returns t if the source file has been written more recently than the binary file.

A special simple transformation is

:do-components
: (:do-components *dependencies*) inside a system with component systems will cause the *dependencies* to be done before anything in the component systems. This is useful when you have a module of macro files used by all of the component systems.

The defined complex transformations are

:compile-load
: (:compile-load *input compile-dependencies load-dependencies compile-condition load-condition*) is the same as (:fasload (:compile *input compile-dependencies compile-condition*) *load-dependencies load-condition*). This is the most commonly-used transformation. Everything after *input* is optional.

:compile-load-init
: See page 416.

As was explained above, each filename in an input specification can in fact be a list of strings for the case where the source file of a program differs from the binary file in more than just the file type. In fact, every filename is treated as if it were an infinite list of filenames with the last filename, or in the case of a single string the only filename, repeated forever at the end. Each simple transformation takes some number of input filename arguments, and some number of output filename arguments. As transformations are performed, these arguments are taken from the front of the filename list. The input arguments are actually removed and the output arguments left as input arguments to the next higher transformation. To make this clearer, consider the prog module above having the :compile-load transformation performed on it. This means that prog is given as the input to the :compile transformation and the output from this transformation is given as the input to the :fasload transformation. The :compile transformation takes one input filename argument, the name of a lisp source file, and one output filename argument, the name of the qfasl file. The :fasload transformation takes one input filename argument, the name of a qfasl file, and no output filename arguments. So, for the first and only file in the prog module, the filename argument list looks like ("AI: GEORGE; PROG" "AI: GEORG2; PROG" "AI: GEORG2; PROG" ...). The :compile transformation is given arguments of "AI: GEORGE; PROG" and "AI: GEORG2; PROG" and the filename argument list which it outputs as the input to the :fasload transformation is ("AI: GEORG2; PROG" "AI: GEORG2; PROG" ...). The :fasload transformation then is given its one argument of "AI: GEORG2; PROG".

Note that dependencies are not "transitive" nor "inherited". For example, if module a depends on macros defined in module b, and therefore needs b to be loaded in order to compile, and b has a similar dependency on c, c will not be loaded during compilation of a. Transformations with these dependencies would be written

```
(:compile-load a (:fasload b))
(:compile-load b (:fasload c))
```

To say that compilation of a depends on both b and c, you would instead write

```
(:compile-load a (:fasload b c))
(:compile-load b (:fasload c))
```

If in addition a depended on c (but not b) during loading (perhaps a contains defvars whose initial values depend on functions or special variables defined in c) you would write the transformations

```
(:compile-load a (:fasload b c) (:fasload c))
(:compile-load b (:fasload c))
```

24.3 Making a System

make-system *name* &rest *keywords*

The make-system function does the actual work of compiling and loading. In the example above, if PROG1 and PROG2 have both been compiled recently, then

```
(make-system 'mysys)
```

will load them as necessary. If either one might also need to be compiled, then

```
(make-system 'mysys ':compile)
```

will do that first as necessary.

make-system lists what transformations it is going to perform on what files, asks the user for confirmation, then performs the transformations. Before each transformation a message is printed listing the transformation being performed, the file it is being done to, and the package. This behavior can be altered by *keywords*.

These are the keywords recognized by the make-system function and what they do.

:noconfirm	Assumes a yes answer for all questions that would otherwise be asked of the user.
:selective	Asks the user whether or not to perform each transformation that appears to be needed for each file.
:silent	Avoids printing out each transformation as it is performed.
:reload	Bypasses the specified conditions for performing a transformation. Thus files are compiled even if they haven't changed and loaded even if they aren't newer than the installed version.
:noload	Does not load any files except those required by dependencies. For use in conjunction with the :compile option.
:compile	Compiles files also if need be. The default is to load but not compile.
:no-increment-patch	When given along with the :compile option, disables the automatic incrementing of the major system version that would otherwise take place. See section 24.7, page 416.
:increment-patch	Increment a patchable system's major version without doing any compilations. See section 24.7, page 416.
:batch	Allows a large compilation to be done unattended. It acts like :noconfirm with regard to questions, turns off more-processing and fdefine-warnings (see inhibit-fdefine-warnings, page 149), and saves the compiler warnings in an editor buffer and a file (it asks you for the name).
:print-only	Just prints out what transformations would be performed, does not actually do any compiling or loading.
:noop	Is ignored. This is mainly useful for programs that call make-system, so that such programs can include forms like `(make-system 'mysys (if compile-p ':compile ':noop))`

24.4 Adding New Keywords to make-system

make-system keywords are defined as functions on the si:make-system-keyword property of the keyword. The functions are called with no arguments. Some of the relevant variables they can use are

si:*system-being-made* *Variable*

The internal data structure which represents the system being made.

si:*make-system-forms-to-be-evaled-before* *Variable*
A list of forms which are evaluated before the transformations are performed.

si:*make-system-forms-to-be-evaled-after* *Variable*
A list of forms which are evaluated after the transformations have been performed.

si:*make-system-forms-to-be-evaled-finally* *Variable*
A list of forms which are evaluated after the body of make-system has completed. This differs from si:*make-system-forms-to-be-evaled-after* in that these forms are evaluated outside of the "compiler context", which sometimes makes a difference.

si:*query-type* *Variable*
Controls how questions are asked. Its normal value is :normal. :noconfirm means no questions will be asked and :selective asks a question for each individual file transformation.

si:*silent-p* *Variable*
If t, no messages are printed out.

si:*batch-mode-p* *Variable*
If t, :batch was specified.

si:*redo-all* *Variable*
If t, all transformations are performed, regardless of the condition functions.

si:*top-level-transformations* *Variable*
A list of the names of transformations that will be performed, such as (:fasload :readfile).

si:*file-transformation-function* *Variable*
The actual function that gets called with the list of transformations that need to be performed. The default is si:do-file-transformations.

si:define-make-system-special-variable *Special Form*
variable value [defvar-p]
Causes *variable* to be bound to *value*, which is evaluated at make-system time, during the body of the call to make-system. This allows you to define new variables similar to those listed above. If *defvar-p* is specified as (or defaulted to) t, *variable* is defined with defvar. It is not given an initial value. If *defvar-p* is specified as *nil*, *variable* belongs to some other program and is not defvar'ed here.

The following simple example adds a new keyword to make-system called :just-warn, which means that fdefine warnings (see page 149) regarding functions being overwritten should be printed out, but the user should not be queried.

```
(si:define-make-system-special-variable
   inhibit-fdefine-warnings inhibit-fdefine-warnings nil)

(defun (:just-warn si:make-system-keyword) ()
  (setq inhibit-fdefine-warnings ':just-warn))
```

(See the description of the inhibit-fdefine-warnings variable, on page 149.)

make-system keywords can have effect either directly when called, or by pushing a form to be evaluated onto si:*make-system-forms-to-be-evaled-after* or one of the other two similar lists. In general, the only useful thing to do is to set some special variable defined by si:define-make-system-special-variable. In addition to the ones mentioned above, user-defined transformations may have their behavior controlled by new special variables, which can be set by new keywords. If you want to get at the list of transformations to be performed, for example, the right way would be to set si:*file-transformation-function* to a new function, which then might call si:do-file-transformations with a possibly modified list. That is how the :print-only keyword works.

24.5 Adding New Options for defsystem

Options to defsystem are defined as macros on the si:defsystem-macro property of the option keyword. Such a macro can expand into an existing option or transformation, or it can have side effects and return nil. There are several variables they can use; the only one of general interest is

si:*system-being-defined* *Variable*

The internal data structure which represents the system which is currently being constructed.

si:define-defsystem-special-variable *variable value* *Special Form*

Causes *value* to be evaluated and *variable* to be bound to the result during the expansion of the defsystem special form. This allows you to define new variables similar to the one listed above.

si:define-simple-transformation *Special Form*

This is the most convenient way to define a new simple transformation. The form is

(si:define-simple-transformation *name function*
default-condition input-file-types output-file-types
pretty-names compile-like load-like)

For example,

```
(si:define-simple-transformation :compile si:qc-file-1
        si:file-newer-than-file-p ("LISP") ("QFASL"))
```

input-file-types and *output-file-types* are how a transformation specifies how many input filenames and output filenames it should receive as arguments, in this case one of each. They also, obviously, specify the default file type for these pathnames. The si:qc-file-1 function is mostly like qc-file, except for its interface to packages. It takes input-file and output-file arguments.

pretty-names, *compile-like*, and *load-like* are optional.

pretty-names specifies how messages printed for the user should print the name of the transformation. It can be a list of the imperative ("Compile"), the present participle ("Compiling"), and the past participle ("compiled"). Note that the past participle is not capitalized, because it is not used at the beginning of a sentence. *pretty-names* can be just a string, which is taken to be the imperative, and the system will conjugate the participles itself. If *pretty-names* is omitted or nil it defaults to the name of the transformation.

compile-like and *load-like* say when the transformation should be performed. Compile-like transformations are performed when the :compile keyword is given to make-system. Load-like transformations are performed unless the :noload keyword is given to make-system. By default *compile-like* is t but *load-like* is nil.

Complex transformations are just defined as normal macro expansions, for example,

```
(defmacro (:compile-load si:defsystem-macro)
                  (input &optional com-dep load-dep
                                   com-cond load-cond)
  '(:fasload (:compile ,input ,com-dep ,com-cond)
             ,load-dep ,load-cond))
```

24.6 More Esoteric Transformations

It is sometimes useful to specify a transformation upon which something else can depend, but which is not performed by default, but rather only when requested because of that dependency. The transformation nevertheless occupies a specific place in the hierarchy. The :skip defsystem macro allows specifying a transformation of this type. For example, suppose there is a special compiler for the read table which is not ordinarily loaded into the system. The compiled version should still be kept up to date, and it needs to be loaded if ever the read table needs to be recompiled.

```
(defsystem reader
  (:pathname-default "AI: LMIO;")
  (:package system-internals)
  (:module defs "RDDEFS")
  (:module reader "READ")
  (:module read-table-compiler "RTC")
  (:module read-table "RDTBL")
  (:compile-load defs)
  (:compile-load reader (:fasload defs))
  (:skip :fasload (:compile read-table-compiler))
  (:rtc-compile-load read-table (:fasload read-table-compiler)))
```

Assume that there is a complex transformation :rtc-compile-load which is like :compile-load, except that is is built on a transformation called something like :rtc-compile, which uses the read table compiler rather than the Lisp compiler. In the above system, then, if the :rtc-compile transformation is to be performed, the :fasload transformation must be done on read-table-compiler first, that is the read table compiler must be loaded if the read table is to be recompiled. If you say (make-system 'reader ':compile), then the :compile transformation will still happen on the read-table-compiler module, compiling the read table compiler if need be. But if you say (make-system 'reader) the reader and the read table will be loaded, but the :skip keeps this from happening to the read table compiler.

So far nothing has been said about what can be given as a *condition* for a transformation except for the default functions which check for a source file being newer than the binary and so on. In general, any function which takes the same arguments as the transformation function (e.g. qc-file) and returns t if the transformation needs to be performed, can be in this place as a symbol, including for example a closure. To take an example, suppose there is a file which contains compile-flavor-methods for a system, and which should therefore be recompiled if any of the flavor method definitions change. In this case, the condition function for compiling that file should return t if either the source of that file itself or any of the files that define the flavors have changed. This is what the :compile-load-init complex transformation is for. It is defined like this:

```
    (defmacro (:compile-load-init si:defsystem-macro)
                     (input add-dep &optional com-dep load-dep
                      &aux function)
      (setq function (let-closed ((*additional-dependent-modules*
                                    add-dep))
                       'compile-load-init-condition))
      '(:fasload (:compile ,input ,com-dep ,function) ,load-dep))

    (defun compile-load-init-condition (source-file qfasl-file)
      (or (si:file-newer-than-file-p source-file qfasl-file)
          (local-declare ((special *additional-dependent-modules*))
            (si:other-files-newer-than-file-p
                              *additional-dependent-modules*
                              qfasl-file))))
```

The condition function which will be generated when this macro is used returns t either if si:file-newer-than-file-p would with those arguments, or if any of the other files in add-dep, which presumably is a *module specification*, are newer than the qfasl file. Thus the file (or module) to which the :compile-load-init transformation applies will be compiled if it or any of the source files it depends on has been changed, and will be loaded under the normal conditions. In most (but not all cases), com-dep would be a :fasload transformation of the same files as add-dep specifies, so that all the files this one depends on would be loaded before compiling it.

24.7 The Patch Facility

The patch facility allows a system maintainer to manage new releases of a large system and issue patches to correct bugs. It is designed to be used to maintain both the Lisp Machine system itself, and applications systems that are large enough to be loaded up and saved on a disk partition.

When a system of programs is very large, it needs to be maintained. Often problems are found and need to be fixed, or other little changes need to be made. However, it takes a long time to load up all of the files that make up such a system, and so rather than having every user load up all the files every time he wants to use the system, usually the files just get loaded once into a Lisp world, and then the Lisp world is saved away on a disk partition. Users then use this disk partition, and copies of it are distributed. The problem is that since the users don't load up the system every time they want to use it, they don't get all the latest changes.

The purpose of the patch system is to solve this problem. A *patch* file is a little file that, when you load it, updates the old version of the system into the new version of the system. Most often, patch files just contain new function definitions; old functions are redefined to do their new thing. When you want to use a system, you first use the Lisp environment saved on the disk, and then you load all the latest patches. Patch files are very small, so loading them doesn't take much time. You can even load the saved environment, load up the latest patches, and then save it away, to save future users the trouble of even loading the patches. (Of course, new patches may be made later, and then these will have to be loaded if you want to get the very latest version.)

For every system, there is a series of patches that have been made to that system. To get the latest version of the system, you load each patch file in the series, in order. Sooner or later, the maintainer of a system will want to stop building more and more patches, and recompile everything, starting afresh. A complete recompilation is also necessary when a system is changed in a far-reaching way, that can't be done with a small patch; for example, if you completely reorganize a program, or change a lot of names or conventions, you might need to completely recompile it to make it work again. After a complete recompilation has been done, the old patch files are no longer suitable to use; loading them in might even break things.

The way all this is kept track of is by labelling each version of a system with a two-part number. The two parts are called the *major version number* and the *minor version number*. The minor version number is increased every time a new patch is made; the patch is identified by the major and minor version number together. The major version number is increased when the program is completely recompiled, and at that time the minor version number is reset to zero. A complete system version is identified by the major version number, followed by a dot, followed by the minor version number.

To clarify this, here is a typical scenario. A new system is created; its initial version number is 1.0. Then a patch file is created; the version of the program that results from loading the first patch file into version 1.0 is called 1.1. Then another patch file might be created, and loading that patch file into system 1.1 creates version 1.2. Then the entire system is recompiled, creating version 2.0 from scratch. Now the two patch files are irrelevant, because they fix old software; the changes that they reflect are integrated into system 2.0.

Note that the second patch file should only be loaded into system 1.1 in order to create system 1.2; you shouldn't load it into 1.0 or any other system besides 1.1. It is important that all the patch files be loaded in the proper order, for two reasons. First, it is very useful that any system numbered 1.1 be exactly the same software as any other system numbered 1.1, so that if somebody reports a bug in version 1.1, it is clear just which software is being complained about. Secondly, one patch might patch another patch; loading them in some other order might have the wrong effect.

The patch facility keeps track, in the file system, of all the patch files that exist, remembering which version each one creates. There is a separate numbered sequence of patch files for each major version of each system. All of them are stored in the file system, and the patch facility keeps track of where they all are. In addition to the patch files themselves, there are "patch directory" files which contain the patch facility's data base by which it keeps track of what minor versions exist for a major version, and what the last major version of a system is. These files and how to make them are described below.

In order to use the patch facility, you must define your system with defsystem (see chapter 24, page 406) and declare it as patchable with the :patchable option. When you load your system (with make-system, see page 411) it is added to the list of all systems present in the world. The patch facility keeps track of which version of each patchable system is present, and where the data about that system reside in the file system. This information can be used to update the Lisp world automatically to the latest versions of all the systems it contains. Once a system is present, you can ask for the latest patches to be loaded, ask which patches are already loaded, and add new patches.

You can also load in patches or whole new systems and then save the entire Lisp environment away in a disk partition. This is explained on section 24.8, page 422.

When a Lisp Machine is booted, it prints out a line of information telling you what systems are present, and which version of each system is loaded. This information is returned by the function si:system-version-info. It is followed by a text string containing any additional information that was requested by whomever created the current disk partition (see **disk-save**, page 424).

print-system-modifications &rest *system-names*

With no arguments, this lists all the systems present in this world and, for each system, all the patches that have been loaded into this world. For each patch it shows the major version number (which will always be the same since a world can only contain one major version), the minor version number, and an explanation of what the patch does, as typed in by the person who made the patch.

If print-system-modifications is called with arguments, only the modifications to the systems named are listed.

si:get-system-version &optional *system*

Returns two values, the major and minor version numbers of the version of *system* currently loaded into the machine, or nil if that system is not present. *system* defaults to "System".

si:system-version-info &optional (*brief-p* nil)

This returns a string giving information about which systems and what versions of the systems are loaded into the machine, and what microcode version is running. A typical string for it to produce is:

```
"System 65.12, ZMail 19.1, Vision 10.23, microcode 739"
```

If *brief-p* is t, it uses short names, suppresses the microcode version, any systems which should not appear in the disk label comment, the name System, and the commas:

```
"65.12 Vis 10.23"
```

24.7.1 Defining a System

In order to use the patch facility, you must declare your system as patchable by giving the :patchable option to defsystem (see chapter 24, page 406). The major version of your system in the file system will be incremented whenever make-system is used to compile it. Thus a major version is associated with a set of QFASL files. The major version of your system that is remembered as having been loaded into the Lisp environment will be set to the major version in the file system whenever make-system is used to load your system and the major version in the file system is greater than what you had loaded before.

After loading your system, you can save it with the disk-save function (see page 424). disk-save will ask you for any additional information you want printed as part of the greeting when the machine is booted. This is in addition to the names and versions of all the systems present in this world. If the system version will not fit in the 16-character field allocated in the disk label, disk-save will ask you to type in an abbreviated form.

24.7.2 Patch files

The patch system will maintain several different types of files in the directory associated with your system. This directory is specified to defsystem via either the :patchable option or the :pathname-default option. These files are maintained automatically, but so that you will know what they are and when they are obsolete (because they are associated with an obsolete version of your system), they are described here.

The file that tells the system's current major version has a name of the form AI: MYDIR; PATCH (PDIR) (on Tops-20, EE:PS:<MYDIR>PATCH.DIRECTORY), where the host, device, and directory (AI:MYDIR; or EE:PS:<MYDIR> in this example) come from the system definition as explained above.

For each major version of the system, there is a *patch directory file*, of the form AI: MYDIR; PAT259 (PDIR), which describes the individual patches for that version, where 259 is the major version number in this example. (On Tops-20, this is EE:PS:<MYDIR>PATCH-259.DIRECTORY).

Then for each minor version of the system, the source of the patch file itself has a name of the form AI: MYDIR; P59.69 >, for minor version 69 of major version 259. Note that 259 has been truncated to 59 to fit into six characters for ITS. On Tops-20 this would be EE:PS:<MYDIR>PATCH-259-69.LISP. Patch files get compiled, so there will also be files like AI: MYDIR; P59.69 QFASL (on Tops-20, EE:PS:<MYDIR>PATCH-259-69.QFASL).

If the :patchable option to defsystem is given an argument, telling it to put the patch files in a different directory than the one which holds the other files of the system, then a slightly different set of file name conventions are used.

On ITS, the file that tells the current major version is of the form AI: PATDIR; *system* (PDIR), where *system* is the name of the system and PATDIR is the directory specified in the :patchable option to defsystem. The patch directory file for major version *nnn* is of the form AI: PATDIR; *sysnnn* (PDIR), where *sys* is the short name specified with the :short-name option to defsystem. A patch file has a name of the form AI: PATDIR; *nnn.mm*; note that the major version is truncated to three digits instead of two. In this set of file name conventions, the patch files don't all fall together in alphabetical order, as they do in the first set.

On TOPS-20, the file names take the forms EE:PS:<PATDIR>*system*.PATCH-DIRECTORY, EE:PS:<PATDIR>*system-nnn*.PATCH-DIRECTORY, and EE:PS:<PATDIR>*system-nnn-mmm*.LISP (or .QFASL). These file name conventions allow the patches for multiple systems to coexist in the same directory.

24.7.3 Loading Patches

load-patches &rest *options*

This function is used to bring the current world up to the latest minor version of whichever major version it is, for all systems present, or for certain specified systems. If there are any patches available, load-patches will offer to read them in. With no arguments, load-patches updates all the systems present in this world.

options is a list of keywords. Some keywords are followed by an argument. The following options are accepted:

:systems *list*	*list* is a list of names of systems to be brought up to date. If this option is not specified, all systems are processed.
:verbose	Print an explanation of what is being done. This is the default.
:selective	For each patch, say what it is and then ask the user whether or not to load it. This is the default. If the user answers "P", selective mode is turned off for any remaining patches to the current system.
:noselective	Turns off :selective.
:silent	Turns off both :selective and :verbose. In :silent mode all necessary patches are loaded without printing anything and without querying the user.

Currently load-patches is not called automatically, but the system may be changed to offer to load patches when the user logs in, in order to keep things up to date.

24.7.4 Making Patches

There are two editor commands that are used to create patch files. During a typical maintenance session on a system you will make several edits to its source files. The patch system can be used to copy these edits into a patch file so that they can be automatically incorporated into the system to create a new minor version. Edits in a patch file can be modified function definitions, new functions, modified defvar's and defconst's, or arbitrary forms to be evaluated, even including load's of new files.

Meta-X Add Patch adds the region (if there is one) or else the current "defun" to the patch file currently being constructed. The first time you give this command it will ask you what system you are patching, allocate a new minor version number, and start constructing the patch file for that version. If you change a function, you should recompile it, test it, then once it works use Add Patch to put it in the patch file.

The patch file being constructed is in an editor buffer. If you mistakenly Add Patch something which doesn't work, you can select the buffer containing the patch file and delete it. Then later you can Add Patch the corrected version.

While you are making your patch file, the minor version number that has been allocated for you is reserved so that nobody else can use it. This way if two people are patching a system at the same time, they will not both get the same minor version number.

After making and testing all of your patches, use meta-X Finish Patch to install the patch file so that other users can load it. This will compile the patch file if you have not done so yourself (patches are always compiled). It will ask you for a comment describing the reason for the patch; load-patches and print-system-modifications print these comments.

After finishing your patch, if you do another Add Patch it will ask you which system again and start a new minor version. Note that you can only be putting together patches for one system at a time.

If you start making a patch file and for some reason never do a Finish Patch (you decide to give up or your machine crashes), the minor version number that you were working on will remain reserved. Since patch files must always be loaded in strictly sequential order, nobody will be able to load any further patches made to this major version past this point. You must manually edit the patch directory file for this major version, removing the line corresponding to the aborted patch. It is OK for a minor version number to be skipped.

24.7.5 System Status

The patch system has the concept of the "status" of a major version of a system. The status is displayed when the system version is displayed, in places such as the system greeting message and the disk partition comment. This status allows users of the system to know what is going on. The status of a system changes as patches are made to it.

The status is one of the following keywords:

:experimental	The system has been built but has not yet been fully debugged and released to users. This is the default status when a new major version is created, unless it is overridden with the :initial-status option to defsystem.
:released	The system is released for general use. This status produces no extra text in the system greeting and the disk partition comment.
:obsolete	The system is no longer supported.
:broken	This is like :experimental but is used when the system was thought incorrectly to have been debugged, and hence was :released for a while.

si:set-system-status *system status* &optional *major-version*

Changes the status of a system. *system* is the name of the system. *major-version* is the number of the major version to be changed; if unsupplied it defaults to the version currently loaded into the Lisp world. *status* should be one of the keywords above.

24.8 Saving New Versions: Disk Partitions

24.8.1 Concepts

The make-system and load-patches functions, described above, load software into the Lisp world. This takes time; it is wasteful for everyone to sit through this loading of software every time the software is to be used. Usually someone loads up software into a Lisp world and then saves away the whole Lisp world in a partition on a disk. This section explains how to do this and other things.

A Lisp Machine disk is divided into several named *partitions* (also called "bands" sometimes). Partitions can be used for many things. Every disk has a partition named PAGE, which is used to implement the virtual memory of the Lisp Machine. When you run Lisp, this is where the Lisp world actually resides. There are also partitions that hold saved images of the Lisp Machine microcode, conventionally named MCR*n* (where *n* is a digit), and partitions that hold saved images of Lisp worlds, conventionally named LOD*n*. A saved image of a Lisp world is also called a "virtual memory load" or "system load".

The directory of partitions is in a special block on the disk called the label. When you "cold-boot" a Lisp Machine by typing CTRL/META/CTRL/META-Rubout, the machine checks the label to see which two partitions contain two important "files": the current microcode load, and the current saved image of the Lisp world. These are kept separate so that the microcode can be easily changed without going through the time-consuming process of generating a new system load. When you "cold-boot", the contents of the current microcode band are loaded into the microcode memory, and then the contents of the current saved image of the Lisp world is copied into the PAGE partition. Then Lisp starts running.

For each partition, the directory of partitions contains a brief textual description of the contents of the partition. For microcode partitions, a typical description might be "UCADR 739"; this means that version 739 of the microcode is in the partition. For saved Lisp images, it is a little more complicated. Ideally, the description would say which versions of which systems are loaded into the band. Unfortunately, there isn't enough room for that in most cases. A typical description is "65.8 ZMail 19.1", meaning that this band contains version 65.8 of System and version 19.1 of ZMail. The description is created when a Lisp world is saved away by disk-save (see below).

24.8.2 Manipulating the Label

print-disk-label &optional (*unit* 0) (*stream* standard-output)
Print a description of the label of the disk specified by *unit* onto *stream*. The description starts with the name of the disk pack, various information about the disk that is generally uninteresting, and the names of the two current load partitions (microcode and saved Lisp image). This is followed by one line of description for each partition. Each one has a name, disk address, size, and textual description. The two partitions that are the current load partitions, used when you cold-boot, are preceeded by asterisks. *unit* may be the unit number of the disk (most Lisp machines just have one unit, numbered 0), or the "host name" of another Lisp Machine on the Chaosnet (in which case the label of unit 0

on that machine will be printed, and the user of that machine will be notified that you are looking at his label).

set-current-band *partition-name*
Set the current saved Lisp image partition to be *partition-name*. If *partition-name* is a number, the name LOD*n* will be used.

set-current-microload *partition-name*
Set the current microcode partition to be *partition-name*. If *partition-name* is a number, the name MCR*n* will be used.

When using the functions to set the current load partitions, be extra sure that you are specifying the correct partition. Having done it, cold-booting the machine will reload from those partitions. Some versions of the microcode will not work with some versions of the Lisp system, and if you set the two current partitions incompatibly, cold-booting the machine will fail; you will need an expert to fix this.

si:edit-disk-label *unit* &optional *init-p*
This runs an interactive label editor on the specified unit. This editor allows you to change any field in the label. The HELP key documents the commands. You have to be an expert to need this and to understand what it does, so the commands are not documented here. Ask someone if you need help.

print-loaded-band &optional *format-dest*
Tells you what you are currently running. This includes where it came from on the disk and what version of each system is present in your Lisp environment. *format-dest* defaults to t; if it is nil the answer will be returned as a string rather than printed out.

disk-restore &optional *partition*
Allows booting from a band other than the current one. *partition* may be the name or the number of a disk partition containing a virtual-memory load, or nil or omitted, meaning to use the current partition. The specified partition is copied into the paging area of the disk and then started.

Although you can use this to boot a different Lisp image than the installed one, this does not provide a way to boot a different microcode image. disk-restore brings up the new band with the currently running microcode.

disk-restore asks the user for confirmation before doing it.

24.8.3 Updating Software

Of all the procedures described in this section, the most common one is to take a partition containing a Lisp image, update it to have all the latest patches, and save it away into a partition.

The way you do this is to start by cold-booting the machine, to get a fresh, empty system. Next, you must log in as something whose INIT file does not affect the Lisp world noticably (so that when you save away the Lisp image, the side-effects of the INIT file won't get saved too); you can log in as "LISPM". Now you can load in any new software you want; usually you just do (load-patches) and answer the questions, to bring all the present patchable systems up to date, but you might also add a new system and load it up. You may also want to call si:set-system-status to change the release status of the system.

When you're done loading everything, do (print-disk-label) to find a band in which to save your new Lisp world. It is best not to reuse the current band, since if something goes wrong during the saving of the partition, while you have written, say, half of the band that is current, it may be impossible to cold-boot the machine. Once you have found the partition, you use the disk-save function to save everything into that partition.

disk-save *partition-name*

Save the current Lisp world in the designated partition. *partition-name* may be a partition name (a string), or it may be a number in which case the name LOD*n* is used.

It first asks you for yes-or-no confirmation that you really want to reuse the named partition. Then it tries to figure out what to put into the textual description of the label. It starts with the brief version of si:system-version-info (see page 418). Then it asks you for an "additional comment" to append to this; usually you just type a return here, but you can also add a comment that will be returned by si:system-version-info (and thus printed when the system is booted) from then on. If this doesn't fit into the fixed size available for the textual description, it asks you to retype the whole thing (the version info as well as your comment) in a compressed form that will fit. The compressed version will appear in the textual description in print-disk-label.

The Lisp environment is then saved away into the designated partition, and then the equivalent of a cold-boot from that partition is done.

Once the patched system has been successfully saved and the system comes back up, you can make it current with set-current-band.

Please don't save patched systems that have had the editor or the compiler run. This works, but it makes the saved system a lot bigger. You should try to do as little as possible between the time you cold-boot and the time you save the partition, in order to produce a clean saved environment.

si:login-history *Variable*

The value of si:login-history is a list of entries, one for each person who has logged into this world since it was created. This makes it possible to tell who disk-saved a band with something broken in it. Each entry is a list of the user ID, the host logged into, the Lisp machine on which the world was being executed, and the date and time.

24.8.4 Installing New Software

The version numbers of the current microcode and system are announced to the INFO-LISPM mailing list. When a new system becomes available, mail is sent to the list explaining where to find the new system and what is new about it. Sometimes a microcode and a system go together, and the new system will not work with the old microcode and vice versa. When this happens extra care is required to avoid getting incompatible loads current at the same time so that the machine will not be able to boot itself.

All of the extant microcode versions can be found on the LISPM1 directory on AI. Microcode version *nnn* is in AI: LISPM1; UCADR *nnn*MCR. To copy a new microcode version into one of the microcode load partitions, first do a (print-disk-label) to ensure that the partition you intend to bash is not the current one; if it was, and something went wrong in the middle of loading the new microcode, it would be impossible to cold-boot, and this is hard to fix.

Then, install the microcode (on the non-current partition) by using si:load-mcr-file.

si:load-mcr-file *microcode-file partition-name*

Load the contents of the file *microcode-file* into the designated partition. Usually *microcode-file* looks like "AI: LISPM1; UCADR *nnn*MCR", and *partition-name* is "MCR1" or "MCR2". This takes about 30 seconds.

The system load, unlike the microcode load, is much too large to fit in an AI file. Therefore, the only way to install an updated system on a machine is to copy it from another machine that already has it. So the first step is to find a machine that is not in use and has the desired system. We will call this the source machine. The machine where the new system will be installed is the target machine. You can see who is logged into which machines, see which ones are free, and use print-disk-label with an argument to examine the label of that machine's disk and see if it has the system you want.

The function for actually copying a system load partition off of another machine is called as follows. Before doing this, double-check the partition names by printing the labels of both machines, and make sure no one is using the source machine.

si:receive-band *source-host source-band target-band*

Copy the partition on *source-host*'s partition named *source-band* onto the local machine's partition named *target-band*. ("Band" means "partition".) This takes about ten minutes. It types out the size of the partition in pages, and types a number every 100 pages telling how far it has gotten. It puts up a display on the remote machine saying what's going on.

To go the other direction, use si:transmit-band.

si:transmit-band *source-band target-host target-band*
This is just like si:receive-band, except you use it on the source machine instead of the target machine. It copies the local machine's partition named *source-band* onto *target-machine*'s partition named *target-band*.

After transferring the band, it is good practice to make sure that it really was copied successfully by comparing the original and the copy. All of the known reasons for errors during band transfer have (of course) been corrected, but peace of mind is valuable. If the copy was not perfectly faithful, you might not find out about it until a long time later, when you use whatever part of the system that had not been copied properly.

si:compare-band *source-host source-band target-band*
This is like si:receive-band, except that it does not change anything. It compares the two bands and complains about any differences.

Having gotten the current microcode load and system load copied into partitions on your machine, you can make them current using set-current-microload and set-current-band. Double-check everything with print-disk-label. Then cold-boot the machine, and the new system should come up in a half-minute or so.

If the microcode you installed is not the same version as was installed on the source machine from which you got the system load, you will need to follow the procedure given below under "installing new microcode". This can happen if someone hasn't installed the current microcode yet on that other machine.

24.8.5 Installing New Microcode

When an existing system is to be used with a new microcode, certain changes need to be made to the system, and it should then be dumped back out with the changes. Usually new microcode is released only along with a new system, so you hardly ever have to do this. The error handler has a table of errors that are detected by microcode. The hardware/microcode debugger (CC) has a microcode symbol table. These symbols are used when debugging other machines, and are also used by certain metering programs. These tables should be updated when a new microcode is installed.

The error-handler will automatically update its table (from a file on the AI:LISPM1; directory) when the machine is booted with the new microcode. The CC symbol table is updated by the following procedure:

```
(login 'lispm)
(pkg-goto 'cadr)
(cc-load-ucode-symbols "AI: LISPM1; UCADR nnnSYM")
(pkg-goto)
```

where *nnn* is the microcode version number. This operation will take a minute or two; after it has read in most of the file the machine will stop for a long time while it sorts the symbols. It will look like it has crashed, but it hasn't, really, and will eventually come back.

After booting the system with the new microcode and following the above procedure, the updated system should be saved with disk-save as explained above. Note that this operation does not change the system version number. Once the new band is verified to work, the old band can be removed from the label with si:edit-disk-label if desired.

25. Processes

The Lisp Machine supports *multi-processing*; several computations can be executed "concurrently" by placing each in a separate *process*. A process is like a processor, simulated by software. Each process has its own "program counter", its own stack of function calls and its own special-variable binding environment in which to execute its computation. (This is implemented with stack groups, see chapter 12, page 163.)

If all the processes are simply trying to compute, the machine time-slices between them. This is not a particularly efficient mode of operation since dividing the finite memory and processor power of the machine among several processes certainly cannot increase the available power and in fact wastes some of it in overhead. The way processes are normally used is different; there can be several on-going computations, but at a given moment only one or two processes will be trying to run. The rest will be either *waiting* for some event to occur, or *stopped*, that is, not allowed to compete for resources.

A process waits for an event by means of the process-wait primitive, which is given a predicate function which defines the event being waited for. A module of the system called the process scheduler periodically calls that function. If it returns nil the process continues to wait; if it returns t the process is made runnable and its call to process-wait returns, allowing the computation to proceed.

A process may be *active* or *stopped*. Stopped processes are never allowed to run; they are not considered by the scheduler, and so will never become the current process until they are made active again. The scheduler continually tests the waiting functions of all the active processes, and those which return non-nil values are allowed to run. When you first create a process with make-process, it is inactive.

A process has two sets of Lisp objects associated with it, called its *run reasons* and its *arrest reasons*. These sets are implemented as lists. Any kind of object can be in these sets; typically keyword symbols and active objects such as windows and other processes are found. A process is considered *active* when it has at least one run reason and no arrest reasons. A process that is not active is *stopped*, is not referenced by the processor scheduler, and does not compete for machine resources.

To get a computation to happen in another process, you must first create a process, and then say what computation you want to happen in that process. The computation to be executed by a process is specified as an *initial function* for the process and a list of arguments to that function. When the process starts up it applies the function to the arguments. In some cases the initial function is written so that it never returns, while in other cases it performs a certain computation and then returns, which stops the process.

To *reset* a process means to throw (see *throw, page 48) out of its entire computation, then force it to call its initial function again. Resetting a process clears its waiting condition, and so if it is active it will become runnable. To *preset* a function is to set up its initial function (and arguments), and then reset it. This is how you start up a computation in a process.

All processes in a Lisp Machine run in the same virtual address space, sharing the same set of Lisp objects. Unlike other systems which have special restricted mechanisms for inter-process communication, the Lisp Machine allows processes to communicate in arbitrary ways through shared Lisp objects. One process can inform another of an event simply by changing the value of a global variable. Buffers containing messages from one process to another can be implemented as lists or arrays. The usual mechanisms of atomic operations, critical sections, and interlocks are provided (see **%store-conditional** (page 178), **without-interrupts** (page 430), and **process-lock** (page 432).)

A process is a Lisp object, an instance of one of several flavors of process (see chapter 20, page 279). The remainder of this chapter describes the messages you can send to a process, the functions you can apply to a process, and the functions and variables a program running in a process can use to manipulate its process.

25.1 The Scheduler

At any time there is a set of *active processes*; as described above, these are all the processes which are not stopped. Each active process is either currently running, trying to run, or waiting for some condition to become true. The active processes are managed by a special stack group called the *scheduler*, which repeatedly cycles through the active processes, determining for each process whether it is ready to be run, or whether it is waiting. The scheduler determines whether a process is ready to run by applying the process's *wait-function* to its *wait-argument-list*. If the wait-function returns a non-**nil** value, then the process is ready to run; otherwise, it is waiting. If the process is ready to run, the scheduler resumes the current stack group of the process.

When a process's wait-function returns non-**nil**, the scheduler will resume its stack group and let it proceed. The process is now the *current process*, that is, the one process that is running on the machine. The scheduler sets the variable **current-process** to it. It will remain the current process and continue to run until either it decides to wait, or a *sequence break* occurs. In either case, the scheduler stack group will be resumed and it will continue to cycle through the active processes. This way, each process that is ready to run will get its share of time in which to execute.

A process can wait for some condition to become true by calling **process-wait** (see page 431), which will set up its wait-function and wait-argument-list accordingly, and resume the scheduler stack group. A process can also wait for just a moment by calling **process-allow-schedule** (see page 431), which resumes the scheduler stack group but leaves the process runnable; it will run again as soon as all other runnable processes have had a chance.

A sequence break is a kind of interrupt that is generated by the Lisp system for any of a variety of reasons; when it occurs, the scheduler is resumed. The function **si:sb-on** (see page 432) can be used to control when sequence breaks occur. The default is to sequence break once a second. Thus if a process runs continuously without waiting, it will be forced to return control to the scheduler once a second so that any other runnable processes will get their turn.

The system does not generate a sequence break when a page fault occurs; thus time spent waiting for a page to come in from the disk is "charged" to a process the same as time spent computing, and cannot be used by other processes. It is done this way for the sake of simplicity;

this allows the whole implementation of the process system to reside in ordinary virtual memory, and not to have to worry specially about paging. The performance penalty is small since Lisp Machines are personal computers, not multiplexed among a large number of processes. Usually only one process at a time is runnable.

A process's wait function is free to touch any data structure it likes and to perform any computation it likes. Of course, wait functions should be kept simple, using only a small amount of time and touching only a small number of pages, or system performance will be impacted since the wait function will consume resources even when its process is not running. If a wait function gets an error, the error will occur inside the scheduler. All scheduling will come to a halt and the user will be thrown into the error handler. Wait functions should be written in such a way that they cannot get errors. Note that process-wait calls the wait function once before giving it to the scheduler, so an error due simply to bad arguments will not occur inside the scheduler.

Note well that a process's wait function is executed inside the scheduler stack-group, *not* inside the process. This means that a wait function may not access special variables bound in the process. It is allowed to access global variables. It could access variables bound by a process through the closure mechanism (chapter 11, page 158), but more commonly any values needed by the wait function are passed to it as arguments.

current-process *Variable*

The value of current-process is the process which is currently executing, or nil while the scheduler is running. When the scheduler calls a process's wait-function, it binds current-process to the process so that the wait-function can access its process.

without-interrupts *body...* *Special Form*

The *body* forms are evaluated with inhibit-scheduling-flag bound to t. This is the recommended way to lock out multi-processing over a small critical section of code to prevent timing errors. In other words the body is an *atomic operation*. The value(s) of a without-interrupts is/are the value(s) of the last form in the body.
Examples:

```
(without-interrupts
  (push item list))

(without-interrupts
  (cond ((memq item list)
         (setq list (delq item list))
         t)
        (t nil)))
```

inhibit-scheduling-flag *Variable*

The value of inhibit-scheduling-flag is normally nil. If it is t, sequence breaks are deferred until inhibit-scheduling-flag becomes nil again. This means that no process other than the current process can run.

process-wait *whostate function* &rest *arguments*

This is the primitive for waiting. The current process waits until the application of *function* to *arguments* returns non-nil (at which time process-wait returns). Note that *function* is applied in the environment of the scheduler, not the environment of the process-wait, so bindings in effect when process-wait was called will *not* be in effect when *function* is applied. Be careful when using any free references in *function*. *whostate* is a string containing a brief description of the reason for waiting. If the who-line at the bottom of the screen is looking at this process, it will show *whostate*.

Examples:

```
(process-wait "sleep"
        #'(lambda (now)
             (> (time-difference (time) now) 100.))
        (time))

(process-wait "Buffer"
        #'(lambda (b) (not (zerop (buffer-n-things b))))
        the-buffer)
```

process-sleep *interval*

This simply waits for *interval* sixtieths of a second, and then returns. It uses process-wait.

process-wait-with-timeout *whostate interval function* &rest *arguments*

This is like process-wait except that if *interval* sixtieths of a second go by and the application of *function* to *arguments* is still returning nil, then process-wait-with-timeout returns t. If the application of *function* to *arguments* does return non-nil during the interval, process-wait-with-timeout returns nil.

process-allow-schedule

This function simply waits momentarily; all other processes will get a chance to run before the current process runs again.

sys:scheduler-stack-group *Variable*

This is the stack group in which the scheduler executes.

sys:clock-function-list *Variable*

This is a list of functions to be called by the scheduler (with no arguments) 60 times a second. These functions implement various system overhead operations such as blinking the blinking cursor on the screen.

sys:active-processes *Variable*

This is the scheduler's data-structure. It is a list of lists, where the car of each element is an active process or nil and the cdr is information about that process.

sys:all-processes *Variable*
This is a list of all the processes in existence. It is mainly for debugging.

si:initial-process *Variable*
This is the process in which the system starts up when it is booted.

si:sb-on &optional *when*
si:sb-on controls what events cause a sequence break, i.e. when re-scheduling occurs. The following keywords are names of events which can cause a sequence break.

:clock	This event happens periodically based on a clock. The default period is one second. See sys:%tv-clock-rate, page 189.
:keyboard	Happens when a character is received from the keyboard.
:chaos	Happens when a packet is received from the Chaosnet, or transmission of a packet to the Chaosnet is completed.

Since the keyboard and Chaosnet are heavily buffered, there is no particular advantage to enabling the :keyboard and :chaos events, unless the :clock event is disabled.

With no argument, si:sb-on returns a list of keywords for the currently enabled events.

With an argument, the set of enabled events is changed. The argument can be a keyword, a list of keywords, nil (which disables sequence breaks entirely since it is the empty list), or a number which is the internal mask, not documented here.

25.2 Locks

A *lock* is a software construct used for synchronization of two processes. A lock is either held by some process, or is free. When a process tries to seize a lock, it waits until the lock is free, and then it becomes the process holding the lock. When it is finished, it unlocks the lock, allowing some other process to seize it. A lock protects some resource or data structure so that only one process at a time can use it.

In the Lisp Machine, a lock is a locative pointer to a cell. If the lock is free, the cell contains nil; otherwise it contains the process that holds the lock. The process-lock and process-unlock functions are written in such a way as to guarantee that two processes can never both think that they hold a certain lock; only one process can ever hold a lock at a time.

process-lock *locative* &optional (*lock-value* current-process) (*whostate* "Lock")
This is used to seize the lock which *locative* points to. If necessary, process-lock will wait until the lock becomes free. When process-lock returns, the lock has been seized. *lock-value* is the object to store into the cell specified by *locative*, and *whostate* is passed on to process-wait.

process-unlock *locative* &optional (*lock-value* current-process)

This is used to unlock the lock which *locative* points to. If the lock is free or was locked by some other process, an error is signaled. Otherwise the lock is unlocked. *lock-value* must have the same value as the *lock-value* parameter to the matching call to process-lock, or else an error is signalled.

It is a good idea to use unwind-protect to make sure that you unlock any lock that you seize. For example, if you write

```
(unwind-protect
   (progn (process-lock lock-3)
          (function-1)
          (function-2))
   (process-unlock lock-3))
```

then even if function-1 or function-2 does a *throw, lock-3 will get unlocked correctly. Particular programs that use locks often define special forms which package this unwind-protect up into a convenient stylistic device.

process-lock and process-unlock are written in terms of a sub-primitive function called %store-conditional (see page 178), which is sometimes useful in its own right.

25.3 Creating a Process

There are two ways of creating a process. One is to create a "permanent" process which you will hold on to and manipulate as desired. The other way is to say simply, "call this function on these arguments in another process, and don't bother waiting for the result." In the latter case you never actually use the process itself as an object.

make-process *name* &rest *options*

Creates and returns a process named *name*. The process will not be capable of running until it has been reset or preset in order to initialize the state of its computation.

The *options* are alternating keywords and values which allow you to specify things about the process, however no options are necessary if you aren't doing anything unusual. The following options are allowed:

:simple-p	Specifying t here gives you a simple process (see page 438).
:flavor	Specifies the flavor of process to be created. See section 25.5, page 438 for a list of all the flavors of process supplied by the system.
:stack-group	The stack group the process is to use. If this option is not specified a stack group will be created according to the relevant options below.
:warm-boot-action	What to do with the process when the machine is booted. See page 436.
:quantum	See page 436.
:priority	See page 436.
:run-reasons	Lets you supply an initial run reason. The default is nil.

:arrest-reasons
Lets you supply an initial arrest reason. The default is **nil**.

:sg-area
The area in which to create the stack group. The default is the value of **default-cons-area**.

:regular-pdl-area
The area in which to create the stack group's regular pdl. The default is **sys:linear-pdl-area**.

:special-pdl-area
The area in which to create the stack group's special binding pdl. The default is the value of **default-cons-area**.

:regular-pdl-size
How big to make the stack group's regular pdl. The default is large enough for most purposes.

:special-pdl-size
How big to make the stack group's special binding pdl. The default is large enough for most purposes.

:swap-sv-on-call-out
:swap-sv-of-sg-that-calls-me
:trap-enable
Specify those attributes of the stack group. You don't want to use these.

If you specify **:flavor**, there can be additional options provided by that flavor.

The following three functions allow you to call a function and have its execution happen asynchronously in another process. This can be used either as a simple way to start up a process which will run "forever", or as a way to make something happen without having to wait for it complete. When the function returns, the process is returned to a pool of free processes, making these operations quite efficient. The only difference between these three functions is in what happens if the machine is booted while the process is still active.

Normally the function to be run should not do any I/O to the terminal. Refer to section 12.4, page 167 for a discussion of the issues.

process-run-function *name function* &rest *args*
Creates a process named *name*, presets it so it will apply *function* to *args*, and starts it running. If the machine is booted, the process is flushed (see page 438). If it is then reset, *function* will be called again.

process-run-temporary-function *name function* &rest *args*
Creates a process named *name*, presets it so it will apply *function* to *args*, and starts it running. If the machine is booted, the process is killed (returned to the free pool).

process-run-restartable-function *name function* &rest *args*
Creates a process named *name*, presets it so it will apply *function* to *args*, and starts it running. If the machine is booted, the process is reset and restarted.

25.4 Process Messages

These are the messages that can be sent to any flavor of process. Certain process flavors may define additional messages. Not all possible messages are listed here, only those "of interest to the user".

25.4.1 Process Attributes

:name (to process)
Returns the name of the process, which was the first argument to make-process or process-run-function when the process was created. The name is a string which appears in the printed-representation of the process, stands for the process in the who-line and the peek display, etc.

:stack-group (to process)
Returns the stack group currently executing on behalf of this process. This can be different from the initial-stack-group if the process contains several stack groups which coroutine among themselves, or if the process is in the error-handler, which runs in its own stack group.

Note that the stack-group of a *simple* process (see page 438) is not a stack group at all, but a function.

:initial-stack-group (to process)
Returns the stack group the initial-function is called in when the process starts up or is reset.

:initial-form (to process)
Returns the initial "form" of the process. This isn't really a Lisp form; it is a cons whose car is the initial-function and whose cdr is the list of arguments to which that function is applied when the process starts up or is reset.

In a simple process (see page 438), the initial form is a list of one element, the process's function.

To change the initial form, send the :preset message (see page 437).

:wait-function (to process)
Returns the process's current wait-function, which is the predicate used by the scheduler to determine if the process is runnable. This is #'true if the process is running, and #'false if the process has no current computation (just created, initial function has returned, or "flushed" (see page 438).

:wait-argument-list (to process)
Returns the arguments to the process's current wait-function. This will frequently be the &rest argument to process-wait in the process's stack, rather than a true list. The system always uses it in a safe manner, i.e. it forgets about it before process-wait returns.

`:whostate` (to process)
: Returns a string which is the state of the process to go in the who-line at the bottom of the screen. This is "run" if the process is running or trying to run, otherwise the reason why the process is waiting. If the process is stopped, then this whostate string is ignored and the who-line displays arrest if the process is arrested or stop if the process has no run reasons.

`:quantum` (to process)
`:set-quantum` *60ths* (to process)
: Return or change the number of 60ths of a second this process is allowed to run without waiting before the scheduler will run someone else. The quantum defaults to 1 second.

`:quantum-remaining` (to process)
: Returns the amount of time remaining for this process to run, in 60ths of a second.

`:priority` (to process)
`:set-priority` *priority-number* (to process)
: Return or change the priority of this process. The larger the number, the more this process gets to run. Within a priority level the scheduler runs all runnable processes in a round-robin fashion. Regardless of priority a process will not run for more than its quantum. The default priority is 0, and no normal process uses other than 0.

`:warm-boot-action` (to process)
`:set-warm-boot-action` *action* (to process)
: Return or change the process's warm-boot-action, which controls what happens if the machine is booted while this process is active. (Contrary to the name, this applies to both cold and warm booting.) This can be nil, which means to "flush" the process (see page 438), or a function to call. The default is si:process-warm-boot-restart, which resets the process, causing it to start over at its initial function. You can also use si:process-warm-boot-reset, which throws out of the process' computation and kills the process.

`:simple-p` (to process)
: Returns nil for a normal process, t for a simple process. See page 438.

25.4.2 Run and Arrest Reasons

`:run-reasons` (to process)
: Returns the list of run reasons, which are the reasons why this process should be active (allowed to run).

`:run-reason` *object* (to process)
: Adds *object* to the process's run reasons. This can activate the process.

`:revoke-run-reason` *object* (to process)
: Removes *object* from the process's run reasons. This can stop the process.

:arrest-reasons (to process)
Returns the list of arrest reasons, which are the reasons why this process should be inactive (forbidden to run).

:arrest-reason *object* (to process)
Adds *object* to the process's arrest reasons. This can stop the process.

:revoke-arrest-reason *object* (to process)
Removes *object* from the process's arrest reasons. This can activate the process.

:active-p (to process)
:runnable-p (to process)
These two messages are the same. t is returned if the process is active, i.e. it can run if its wait-function allows. nil is returned if the process is stopped.

25.4.3 Bashing the Process

:preset *function* &rest *args* (to process)
Sets the process's initial function to *function* and initial arguments to *args*. The process is then reset so that it will throw out of any current computation and start itself up by applying *function* to *args*. A :preset message to a stopped process will return immediately, but will not activate the process, hence the process will not really apply *function* to *args* until it is activated later.

:reset &optional *no-unwind kill* (to process)
Forces the process to throw out of its present computation and apply its initial function to its initial arguments, when it next runs. The throwing out is skipped if the process has no present computation (e.g. it was just created), or if the *no-unwind* option so specifies. The possible values for *no-unwind* are:

:unless-current nil	Unwind unless the stack group to be unwound is the one we are currently executing in, or belongs to the current process.
:always	Unwind in all cases. This may cause the message to throw through its caller instead of returning.
t	Never unwind.

If *kill* is t, the process is to be killed after unwinding it. This is for internal use by the :kill message only.

A :reset message to a stopped process will return immediately, but will not activate the process, hence the process will not really get reset until it is activated later.

:flush (to process)
: Forces the process to wait forever. A process may not :flush itself. Flushing a process is different from stopping it, in that it is still active and hence if it is reset or preset it will start running again.

:kill (to process)
: Gets rid of the process. It is reset, stopped, and removed from sys:all-processes.

:interrupt *function* &rest *args* (to process)
: Forces the process to apply *function* to *args*. When *function* returns, the process will continue the interrupted computation. If the process is waiting, it wakes up, calls *function*, then waits again when *function* returns.

If the process is stopped it will not apply *function* to *args* immediately, but later when it is activated. Normally the :interrupt message returns immediately, but if the process's stack group is in an unusual internal state it may have to wait for it to get out of that state.

25.5 Process Flavors

These are the flavors of process provided by the system. It is possible for users to define additional flavors of their own.

si:process *Flavor*
: This is the standard default kind of process.

si:simple-process *Flavor*
: A simple process is one which has no stack group; instead it has a function which is applied to no arguments whenever the wait-function returns t. Simple processes run inside the scheduler stack group until they return, and of course cannot have any state since they have no stack-group. Simple processes are a low-overhead mechanism for certain purposes. For example, packets received from the Chaosnet are examined and distributed to the proper receiver by a simple process which wakes up whenever there are any packets in the input buffer.

Asking for the stack group of a simple process does not signal an error, but returns the process's function instead.

A simple process cannot wait by calling process-wait; it must call si:set-process-wait and then return.

25.6 Other Process Functions

process-enable *process*
Activates *process* by revoking all its run and arrest reasons, then giving it a run reason of :enable.

process-reset-and-enable *process*
Resets *process* then enables it.

process-disable *process*
Stops *process* by revoking all its run reasons. Also revokes all its arrest reasons.

The remaining functions in this section are obsolete, since they simply duplicate what can be done by sending a message. They are documented here because their names are in the global package.

process-preset *process function* &rest *args*
Just sends a :preset message.

process-reset *process*
Just sends a :reset message.

process-name *process*
Gets the name of a process, like the :name message.

process-stack-group *process*
Gets the current stack group of a process, like the :stack-group message.

process-initial-stack-group *process*
Gets the initial stack group of a process, like the :initial-stack-group message.

process-initial-form *process*
Gets the initial "form" of a process, like the :initial-form message.

process-wait-function *process*
Gets the current wait-function of a process, like the :wait-function message.

process-wait-argument-list *p*
Gets the arguments to the current wait-function of a process, like the :wait-argument-list message.

process-whostate *p*
Gets the current who-line state string of a process, like the :whostate message.

26. Errors and Debugging

The first section of this chapter explains how programs can handle errors, by means of condition handlers. It also explains how a program can signal an error if it detects something it doesn't like.

The second explains how users can handle errors, by means of an interactive debugger; that is, it explains how to recover if you do something wrong. A new user of the Lisp Machine, or someone who just wants to know how to deal with errors and not how to cause them, should ignore the first section and skip ahead to section 26.2, page 450.

The remaining sections describe some other debugging facilities. Anyone who is going to be writing programs for the Lisp Machine should familiarize himself with these.

The *trace* facility provides the ability to perform certain actions at the time a function is called or at the time it returns. The actions may be simple typeout, or more sophisticated debugging functions.

The *advise* facility is a somewhat similar facility for modifying the behavior of a function.

The *step* facility allows the evaluation of a form to be intercepted at every step so that the user may examine just what is happening throughout the execution of the form.

The *MAR* facility provides the ability to cause a trap on any memory reference to a word (or a set of words) in memory. If something is getting clobbered by agents unknown, this can help track down the source of the clobberage.

26.1 The Error System

26.1.1 Conditions

Programmers often want to control what action is taken by their programs when errors or other exceptional situations occur. Usually different situations are handled in different ways, and in order to express what kind of handling each situation should have, each situation must have an associated name. In Zetalisp there is the concept of a *condition*. Every condition has a name, which is a symbol. When an unusual situation occurs, some condition is *signalled*, and a *handler* for that condition is invoked.

When a condition is signalled, the system (essentially) searches up the stack of nested function invocations looking for a handler established to handle that condition. The handler is a function which gets called to deal with the condition. The condition mechanism itself is just a convenient way for finding an appropriate handler function given the name of an exceptional situation. On top of this is built the error-condition system, which defines what arguments are passed to a handler function and what is done with the values returned by a handler function. Almost all current use of the condition mechanism is for errors, but the user may find other uses for the underlying mechanism.

The search for an appropriate handler is done by the function **signal**:

signal *condition-name* &rest *args*

signal searches through all currently-established condition handlers, starting with the most recent. If it finds one that will handle the condition *condition-name*, then it calls that handler with a first argument of *condition-name*, and with *args* as the rest of the arguments. If the first value returned by the handler is nil, signal will continue searching for another handler; otherwise, it will return the first two values returned by the handler. If signal doesn't find any handler that returns a non-nil value, it will return **nil**.

Condition handlers are established through the condition-bind special form:

condition-bind *Special Form*

The condition-bind special form is used for establishing handlers for conditions. It looks like:

```
(condition-bind ((cond-1 hand-1)
                 (cond-2 hand-2)
                 ...)
   body)
```

Each *cond-n* is either the name of a condition, or a list of names of conditions, or nil. If it is nil, a handler is set up for *all* conditions (this does not mean that the handler really has to handle all conditions, but it will be offered the chance to do so, and can return nil for conditions which it is not interested in). Each *hand-n* is a form which is evaluated to produce a handler function. The handlers are established sequentially such that the *cond-1* handler would be looked at first.

Example:

```
(condition-bind ((:wrong-type-argument 'my-wta-handler)
                 ((lossage-1 lossage-2) lossage-handler))
    (princ "Hello there.")
    (= t 69))
```

This first sets up the function my-wta-handler to handle the :wrong-type-argument condition. Then, it sets up the value of the symbol lossage-handler to handle both the lossage-1 and lossage-2 conditions. With these handlers set up, it prints out a message and then runs headlong into a wrong-type-argument error by calling the function = with an argument which is not a number. The condition handler my-wta-handler will be given a chance to handle the error. condition-bind makes use of ordinary variable binding, so that if the condition-bind form is thrown through, the handlers will be disestablished. This also means that condition handlers are established only within the current stack-group.

26.1.2 Error Conditions

[This section is incorrect. The mechanism by which errors are signalled does not work. It will be redesigned someday.]

The use of the condition mechanism by the error system defines an additional protocol for what arguments are passed to error-condition handlers and what values they may return.

There are basically four possible responses to an error: *proceeding*, *restarting*, *throwing*, or entering the *debugger*. The default action, taken if no handler exists or deigns to handle the error (returns non-nil), is to enter the debugger. A handler may give up on the execution that produced the error by throwing (see *throw, page 48). *Proceeding* means to repair the error and continue execution. The exact meaning of this depends on the particular error, but it generally takes the form of supplying a replacement for an unacceptable argument to some function, and retrying the invocation of that function. *Restarting* means throwing to a special standard catch-tag, error-restart. Handlers cause proceeding and restarting by returning certain special values, described below.

Each error condition is signalled with some parameters, the meanings of which depend on the condition. For example, the condition :unbound-variable, which means that something tried to find the value of a symbol which was unbound, is signalled with at least one parameter, the unbound symbol. It is always all right to signal an error condition with extra parameters beyond those whose meanings are defined by the condition.

An error condition handler is applied to several arguments. The first argument is the name of the condition that was signalled (a symbol). This allows the same function to handle several different conditions, which is useful if the handling of those conditions is very similar. (The first argument is also the name of the condition for non-error conditions.) The second argument is a format control string (see the description of format, on page 346). The third argument is t if the error is *proceedable*; otherwise it is nil. The fourth argument is t if the error is *restartable*; otherwise it is nil. The fifth argument is the name of the function that signalled the error, or nil if the signaller can't figure out the correct name to pass. The rest of the arguments are the parameters with which the condition was signalled. If the format control string is used with these parameters, a readable English message should be produced. Since more information than just the parameters might be needed to print a reasonable message, the program signalling the condition is free to pass any extra parameters it wants to, after the parameters which the condition is *defined* to take. This means that every handler must expect to be called with an arbitrarily high number of arguments, so every handler should have a &rest argument (see page 21).

An error condition handler may return any of several values. If it returns nil, then it is stating that it does not wish to handle the condition after all; the process of signalling will continue looking for a prior handler (established farther down on the stack) as if the handler which returned nil had not existed at all. (This is also true for non-error conditions.) If the handler does wish to handle the condition, it can try to proceed from the error if it is proceedable, or restart from it if it is restartable, or it can throw to a catch tag. Proceeding and restarting are done by returning two values. The first value is one of the following symbols:

:return If the error was signalled by calling cerror, the second value is returned as the value of cerror. If the error was signalled by calling ferror, proceeding is not allowed. If the error was detected by the Lisp system, the error will be proceeded from, using the second value if a data object is needed. For example, for an :undefined-function error, the handler's second value will be used as the function to be called, in place of the non-existent function definition.

eh:return-value
: If the error was signalled by calling ferror or cerror, the second value is returned from that function, regardless of whether the error was proceedable. If the error was detected by the Lisp system, the second value is returned as the result of the function in which the error was detected. It should be obvious that :return-value allows you to do things that are totally unanticipated by the program that got the error.

:error-restart The second value is thrown to the catch tag error-restart.

The condition handler must not return any other sort of values. However, it can legitimately throw to any tag instead of returning at all. If a handler tries to proceed an unproceedable error or restart an unrestartable one, an error is signalled.

Note that if the handler returns nil, it is not said to have handled the error; rather, it has decided not to handle it, but to "continue to signal" it so that someone else may handle it. If an error is signalled and none of the handlers for the condition decide to handle it, the debugger is entered.

Here is an example of an excessively simple handler for the :wrong-type-argument condition.

[Note that this code does not work in system 56.]

```
;;; This function handles the :wrong-type-argument condition,
;;; which takes two defined parameters: a symbol indicating
;;; the correct type, and the bad value.
(defun sample-wta-handler (condition control-string
                           proceedable-flag restartable-flag
                           function correct-type bad-value
                           &rest rest)
 (prog ()
   (format error-output "~%There was an error in ~S~%" function)
   (lexpr-funcall (function format) error-output
                  control-string correct-type bad-value rest)
   (cond ((and proceedable-flag
               (yes-or-no-p "Do you want use nil instead?"))
          (return 'return nil))
         (t (return nil))))) ;don't handle
```

If an error condition reaches the error handler, the RESUME (or control-C) command may be used to continue from it. If the condition name has a eh:proceed property, that property is called as a function with two arguments, the stack-group and the "ete" (an internal error-handler data structure). Usually it will ignore these arguments. If this function returns, its value will be returned from the ferror or cerror that signalled the condition. If no such property exists, the error-handler asks the user for a form, evaluates it, and causes ferror or cerror to return that value. Putting such a property on can be used to change the prompt for this form, avoid asking the user, or change things in more far-reaching ways.

26.1.3 Signalling Errors

Some error conditions are signalled by the Lisp system when it detects that something has gone wrong. Lisp programs can also signal errors, by using any of the functions ferror, cerror, or error. ferror is the most commonly used of these. cerror is used if the signaller of the error wishes to make the error be *proceedable* or *restartable*, or both. error is provided for Maclisp compatibility.

A ferror or cerror that doesn't have any particular condition to signal should use nil as the condition name. The only kind of handler that will be invoked by the signaller in this case is the kind that handles *all* conditions, such as is set up by

```
(condition-bind ((nil something) ...) ...)
```

In practice, the nil condition is used a great deal.

ferror *condition-name control-string* &rest *params*

ferror signals the error condition *condition-name*. The associated error message is obtained by calling format (see section 21.6.1, page 346) on *control-string* and *params*. The error is neither proceedable nor restartable, so ferror will not return unless the user forces it to by intervening with the debugger. In most cases *condition-name* is nil, which means that no condition-handler is likely to be found and the debugger will be entered.

Examples:

```
(cond ((> sz 60)
       (ferror nil
               "The size, ~S, was greater than the maximum"
               sz))
      (t (foo sz)))

(defun func (a b)
   (cond ((and (> a 3) (not (symbolp b)))
          (ferror ':wrong-type-argument
                  "The name, ~1G~S, must be a symbol"
                  'symbolp
                  b))
         (t (func-internal a b))))
```

If the error is not handled and the debugger is entered, the error message is printed by calling format with *control-string* as the control string and the elements of *params* as the additional arguments. Alternatively, the formatted output functions (page 356) can be used to generate the error message:

```
(ferror nil
  (format:outfmt "Frob has "
                 (format:plural (format:onum n-elts)
                                " element")
                 " which is too few"))
```

In this case *params* are not used for printing the error message, and often none are needed. They may still be useful as information for condition handlers, and those that a condition is documented to expect should always be supplied.

cerror *proceedable-flag restartable-flag condition-name control-string* &rest *params*

cerror is just like ferror (see above) except for *proceedable-flag* and *restartable-flag*. If cerror is called with a non-nil *proceedable-flag*, the caller should be prepared to accept the returned value of cerror and use it to retry the operation that failed. Similarly, if he passes cerror a non-nil *restartable-flag*, he should be sure that there is a *catch above him for the tag error-restart.

If *proceedable-flag* is t and the error goes to the debugger, if the user says to proceed from the error he will be asked for a replacement object which cerror will return. If *proceedable-flag* is not t and not nil, the user will not be asked for a replacement object and cerror will return no particular value when the error is proceeded.

Note: Many programs that want to signal restartable errors will want to use the error-restart special form; see page 446.

Example:

```
(do ()
    ((symbolp a))
  ; Do this stuff until a becomes a symbol.
  (setq a (cerror t nil ':wrong-type-argument
              "The argument ~2G~A was ~1G~S, which is not ~3G~A"
              'symbolp a 'a "a symbol")))
```

Note: the form in this example is so useful that there is a standard special form to do it, called check-arg (see page 446).

error *message* &optional *object interrupt*

error is provided for Maclisp compatibility. In Maclisp, the functionality of error is, essentially, that *message* gets printed, preceeded by *object* if present, and that *interrupt*, if present, is a user interrupt channel to be invoked.

In order to fit this definition into Zetalisp way of handling errors, error is defined to be:

```
(cerror (not (null interrupt))
        nil
        (or (get interrupt 'eh:condition-name)
            interrupt)
        (if (missing? object)  ;If no object given
            "~*~A"
            "~S ~A")
        object
        message)
```

Here is what that means in English: first of all, the condition to be signalled is **nil** if *interrupt* is **nil**. If there is some condition whose meaning is close to that of one of the Maclisp user interrupt channels, the name of that channel has an **eh:condition-name** property, and the value of that property is the name of the condition to signal. Otherwise, *interrupt* is the name of the condition to signal; probably there will be no handler and the debugger will be entered.

If *interrupt* is specified, the error will be proceedable. The error will not be restartable. The **format** control string and the arguments are chosen so that the right error message gets printed, and the handler is passed everything there is to pass.

error-restart *body...* *Macro*

error-restart is useful for denoting a section of a program that can be restarted if certain errors occur during its execution. The forms of the body are evaluated sequentially. If an error occurs within the evaluation of the body and is restarted (by a condition handler or the debugger), the evaluation resumes at the beginning of the **error-restart**'s body. The only way a restartable error can occur is if **cerror** is called with a second argument of **t**.

Example:

```
(error-restart
  (setq a (* b d))
  (cond ((> a maxtemp)
         (cerror nil t 'overheat
            "The frammistat will overheat by ~D. degrees!"
            (- a maxtemp))))
  (setq q (cons a a)))
```

If the **cerror** happens, and the handler invoked (or the debugger) restarts the error, then evaluation will continue with the **(setq a (* b d))**, and the condition **(> a maxtemp)** will get checked again.

error-restart is implemented as a macro that expands into:

```
(prog ()
 loop (*catch 'error-restart
              (return (progn
                         form-1
                         form-2
                         ...)))
      (go loop))
```

check-arg *var-name predicate description* [*type-symbol*] *Macro*

The **check-arg** form is useful for checking arguments to make sure that they are valid. A simple example is:

```
(check-arg foo stringp "a string")
```

foo is the name of an argument whose value should be a string. **stringp** is a predicate of one argument, which returns **t** if the argument is a string. **"a string"** is an English description of the correct type for the variable.

The general form of **check-arg** is

```
(check-arg var-name
           predicate
           description
           type-symbol)
```

var-name is the name of the variable whose value is of the wrong type. If the error is proceeded this variable will be setq'ed to a replacement value. *predicate* is a test for whether the variable is of the correct type. It can be either a symbol whose function definition takes one argument and returns non-nil if the type is correct, or it can be a non-atomic form which is evaluated to check the type, and presumably contains a reference to the variable *var-name*. *description* is a string which expresses *predicate* in English, to be used in error messages. *type-symbol* is a symbol which is used by condition handlers to determine what type of argument was expected. It may be omitted if it is to be the same as *predicate*, which must be a symbol in that case.

The use of the *type-symbol* is not really well-defined yet, but the intention is that if it is numberp (for example), the condition handlers can tell that a number was needed, and might try to convert the actual supplied value to a number and proceed.

[We need to establish a conventional way of "registering" the type-symbols to be used for various expected types. It might as well be in the form of a table right here.]

The *predicate* is usually a symbol such as fixp, stringp, listp, or closurep, but when there isn't any convenient predefined predicate, or when the condition is complex, it can be a form. In this case you should supply a *type-symbol* which encodes the type. For example:

```
(check-arg a
           (and (numberp a) (≤ a 10.) (> a 0.))
           "a number from one to ten"
           one-to-ten)
```

If this error got to the debugger, the message

```
The argument a was 17, which is not a number from one to ten.
```

would be printed.

In general, what constitutes a valid argument is specified in three ways in a check-arg. *description* is human-understandable, *type-symbol* is program-understandable, and *predicate* is executable. It is up to the user to ensure that these three specifications agree.

check-arg uses *predicate* to determine whether the value of the variable is of the correct type. If it is not, check-arg signals the :wrong-type-argument condition, with four parameters. First, *type-symbol* if it was supplied, or else *predicate* if it was atomic, or else nil. Second, the bad value. Third, the name of the argument (*var-name*). Fourth, a string describing the proper type (*description*). If the error is proceeded, the variable is set to the value returned, and check-arg starts over, checking the type again. Note that only the first two of these parameters are defined for the :wrong-type-argument condition, and so :wrong-type-argument handlers should only depend on the meaning of these two.

check-arg-type *var-name type-name* [*description*] *Macro*

This is a useful variant of the check-arg form. A simple example is:

```
        (check-arg foo :number)
```

foo is the name of an argument whose value should be a number. :number is a value which is passed as a second argument to typep (see page 10); that is, it is a symbol that specifies a data type. The English form of the type name, which gets put into the error message, is found automatically.

The general form of check-arg-type is:

```
        (check-arg-type var-name
                        type-name
                        description)
```

var-name is the name of the variable whose value is of the wrong type. If the error is proceeded this variable will be setq'ed to a replacement value. *type-name* describes the type which the variable's value ought to have. It can be exactly those things acceptable as the second argument to typep. *description* is a string which expresses *predicate* in English, to be used in error messages. It is optional. If it is omitted, and *type-name* is one of the keywords accepted by :typep, which describes a basic Lisp data type, then the right *description* will be provided correctly. If it is omitted and *type-name* describes some other data type, then the description will be the word "a" followed by the printed representation of *type-name* in lower-case.

26.1.4 Standard Condition Names

Some condition names are used by the kernel Lisp system, and are documented below; since they are of global interest, they are on the keyword package. Programs outside the kernel system are free to define their own condition names; it is intended that the description of a function include a description of any conditions that it may signal, so that people writing programs that call that function may handle the condition if they desire. When you decide what package your condition names should be in, you should apply the same criteria you would apply for determining which package a function name should be in; if a program defines its own condition names, they should *not* be on the keyword package. For example, the condition names chaos:bad-packet-format and arpa:bad-packet-format should be distinct. For further discussion, see chapter 23, page 392.

The following table lists all standard conditions and the parameters they take; more will be added in the future. These are all error-conditions, so in addition to the condition name and the parameters, the handler receives the other arguments described above.

:wrong-type-argument *type-name value*

value is the offending argument, and *type-name* is a symbol for what type is required. Often, *type-name* is a predicate which returns non-nil if applied to an acceptable value. If the error is proceeded, the value returned by the handler should be a new value for the argument to be used instead of the one which was of the wrong type.

:inconsistent-arguments *list-of-inconsistent-argument-values*

These arguments were inconsistent with each other, but the fault does not belong to any particular one of them. This is a catch-all, and it would be good to identify subcases in which a more specific categorization can be made. If the error is proceeded, the value returned by the handler will be returned by the function whose arguments were inconsistent.

:wrong-number-of-arguments *function number-of-args-supplied list-of-args-supplied*
function was invoked with the wrong number of arguments. The elements of *list-of-args-supplied* have already been evaluated. If the error is proceeded, the value returned should be a value to be returned by *function*.

:invalid-function *function-name*
The name had a function definition but it was no good for calling. You can proceed, supplying a value to return as the value of the call to the function.

:invalid-form *form*
The so-called *form* was not a meaningful form for eval. Probably it was of a bad data type. If the error is proceeded, the value returned should be a new form; eval will use it instead.

:undefined-function *function-name*
The symbol *function-name* was not defined as a function. If the error is proceeded, then the value returned will be used instead of the (non-existent) definition of *function-name*.

:unbound-variable *variable-name*
The symbol *variable-name* had no value. If the error is proceeded, then the value returned will be used instead of the (non-existent) value of *variable-name*.

26.1.5 Errset

As in Maclisp, there is an errset facility which allows a very simple form of error handling. If an error occurs inside an errset, and no condition handler handles it, i.e. the debugger would be entered, control is returned (*thrown*) to the errset. The errset can control whether or not the debugger's error message is printed. All errors are caught by errset, whether they are signalled by ferror, cerror, error, or the Lisp system itself.

A problem with errset is that it is *too* powerful; it will apply to any unhandled error at all. If you are writing code that anticipates some specific error, you should find out what condition that error signals and set up a handler. If you use errset and some unanticipated error crops up, you may not be told—this can cause very strange bugs. Note that the variable errset allows all errsets to be disabled for debugging purposes.

errset *form* [*flag*] *Special Form*
The errset special form catches errors during the evaluation of *form*. If an error occurs, the usual error message is printed unless *flag* is nil. Then control is thrown and the errset-form returns nil. *flag* is evaluated first and is optional, defaulting to t. If no error occurs, the value of the errset-form is a list of one element, the value of *form*.

catch-error *form* [*flag*] *Special Form*

catch-error is a variant of errset. This special form catches errors during the evaluation of *form*, and returns two values. Normally the first value is the value of *form* and the second value is nil. If an error occurs, the usual error message is printed unless *flag* is nil, and then control is thrown out of the catch-error-form, which returns two values: first nil and second a non-nil value that indicates the occurrence of an error. *flag* is evaluated first and is optional, defaulting to t.

errset *Variable*

If this variable is non-nil, errset-forms are not allowed to trap errors. The debugger is entered just as if there was no errset. This is intended mainly for debugging. The initial value of errset is nil.

err *Special Form*

This is for Maclisp compatibility only and should not be used.

(err) is a dumb way to cause an error. If executed inside an errset, that errset returns nil, and no message is printed. Otherwise an unseen throw-tag error occurs.

(err *form*) evaluates *form* and causes the containing errset to return the result. If executed when not inside an errset, an unseen throw-tag error occurs.

(err *form flag*), which exists in Maclisp, is not supported.

26.2 The Debugger

When an error condition is signalled and no handlers decide to handle the error, an interactive debugger is entered to allow the user to look around and see what went wrong, and to help him continue the program or abort it. This section describes how to use the debugger.

26.2.1 Entering the Debugger

There are two kinds of errors; those generated by the Lisp Machine's microcode, and those generated by Lisp programs (by using ferror or related functions). When there is a microcode error, the debugger prints out a message such as the following:

```
>>TRAP 5543 (TRANS-TRAP)
The symbol FOOBAR is unbound.
While in the function *EVAL ← SI:LISP-TOP-LEVEL1
```

The first line of this error message indicates entry to the debugger and contains some mysterious internal microcode information: the micro program address, the microcode trap name and parameters, and a microcode backtrace. Users can ignore this line in most cases. The second line contains a description of the error in English. The third line indicates where the error happened by printing a very abbreviated "backtrace" of the stack (see below): in the example, it is saying that the error was signalled inside the function *eval, which was called by si:lisp-top-level1.

Here is an example of an error from Lisp code:

```
>>ERROR: The argument X was 1, which is not a symbol,
While in the function FOO ← *EVAL ← SI:LISP-TOP-LEVEL1
```

Here the first line contains the English description of the error message, and the second line contains the abbreviated backtrace. **foo** signalled the error by calling **ferror**, however **ferror** is censored out of the backtrace.

After the debugger's initial message it prints the function that got the error and its arguments.

The debugger can be manually entered either by causing an error (e.g. by typing a ridiculous symbol name such as **ahsdgf** at the Lisp read-eval-print loop) or by typing the BREAK key with the META shift held down while the program is reading from the terminal. Typing the BREAK key with both CONTROL and META held down will force the program into the debugger immediately, even if it is running. If the BREAK key is typed without META, it puts you into a read-eval-print loop using the **break** function (see page 504) rather into the debugger.

eh *process*

Stops *process* and calls the debugger on it so that you can look at its current state. Exit the debugger with the Control-Z command and **eh** will release the process and return. *process* can be a window, in which case the window's process will be used.

If *process* is not a process but a stack group, the current state of the stack group will be examined. The caller should ensure that no one tries to resume that stack group while the debugger is looking at it.

26.2.2 How to Use the Debugger

Once inside the debugger, the user may give a wide variety of commands. This section describes how to give the commands, and then explains them in approximate order of usefulness. A summary is provided at the end of the listing.

When the error hander is waiting for a command, it prompts with an arrow:

```
→
```

At this point, you may either type in a Lisp expression, or type a command (a Control or Meta character is interpreted as a command, whereas most normal characters are interpreted as the first character of an expression). If you type the HELP key or the ? key, you will get some introductory help with the error handler.

If you type a Lisp expression, it will be interpreted as a Lisp form, and will be evaluated in the context of the function which got the error. That is, all bindings which were in effect at the time of the error will be in effect when your form is evaluated, with certain exceptions explained below. The result of the evaluation will be printed, and the debugger will prompt again with an arrow. If, during the typing of the form, you change your mind and want to get back to the debugger's command level, type the ABORT key or a Control-G; the debugger will respond with an arrow prompt. In fact, at any time that typein is expected from you, while you are in the debugger, you may type ABORT or Control-G to flush what you are doing and get back to command level. This read-eval-print loop maintains the values of +, *, and - just as the top-level one does.

If an error occurs in the evaluation of the Lisp expression you type, you will get into a second error handler looking at the new error. You can abort the computation and get back to the first error by typing the ABORT key (see below). However, if the error is trivial the abort will be done automatically and the original error message will be reprinted.

Various debugger commands ask for Lisp objects, such as an object to return, or the name of a catch-tag. Whenever it tries to get a Lisp object from you, it expects you to type in a *form*; it will evaluate what you type in. This provides greater generality, since there are objects to which you might want to refer that cannot be typed in (such as arrays). If the form you type is non-trivial (not just a constant form), the debugger will show you the result of the evaluation, and ask you if it is what you intended. It expects a Y or N answer (see the function y-or-n-p, page 487), and if you answer negatively it will ask you for another form. To quit out of the command, just type ABORT or Control-G.

When the debugger evaluates a form, the variable bindings at the point of error are in effect with the following exceptions:

terminal-io is rebound to the stream the error handler is using. eh:old-terminal-io is bound to the value terminal-io had at the point of error.

standard-input and standard-output are rebound to be synonymous with terminal-io; their old bindings are saved in eh:old-standard-input and eh:old-standard-output.

+ and * are rebound to the error handler's previous form and previous value. When the debugger is first entered, + is the last form typed, which is typically the one that caused the error, and * is the value of the *previous* form.

evalhook (see page 466) is rebound to nil, turning off the step facility if it had been in use when the error occurred.

Note that the variable bindings are those in effect at the point of error, *not* those of the current frame being looked at. This may be changed in the future.

26.2.3 Debugger Commands

All debugger commands are single characters, usually with the Control or Meta bits. The single most useful command is ABORT (or Control-Z), which exits from the debugger and throws out of the computation that got the error. This is the ABORT key, not a 5-letter command. ITS users should note that Control-Z is not CALL. Often you are not interested in using the debugger at all and just want to get back to Lisp top level; so you can do this in one character.

The ABORT command returns control to the most recent read-eval-print loop. This can be Lisp top level, a break, or the debugger command loop associated with another error. Typing ABORT multiple times will throw back to successively older read-eval-print or command loops until top level is reached. Typing Control-Meta-ABORT, on the other hand, will always throw to top level. Control-Meta-ABORT is not a debugger command, but a system command which is always available no matter what program you are in.

Note that typing ABORT in the middle of typing a form to be evaluated by the debugger aborts that form, and returns to the debugger's command level, while typing ABORT as a debugger command returns out of the debugger and the erring program, to the *previous* command level.

Self-documentation is provided by the HELP or ? command, which types out some documentation on the debugger commands, including any special commands which apply to the particular error currently being handled.

Often you want to try to proceed from the error. To do this, use the RESUME (or Control-C) command. The exact way RESUME works depends on the kind of error that happened. For some errors, there is no standard way to proceed at all, and RESUME will just tell you this and return to the debugger's command level. For the very common "unbound variable" error, it will get a Lisp object from you, which will be used in place of the (nonexistent) value of the symbol. For unbound-variable or undefined-function errors, you can also just type Lisp forms to set the variable or define the function, and then type RESUME; it will proceed without asking anything.

The debugger knows about a "current stack frame", and there are several commands which use it. The initially "current" stack frame is the one which signalled the error; either the one which got the microcode-detected error, or the one which called ferror, cerror, or error. When the debugger starts it up it shows you this frame in the following format:

```
        FOO:
           Arg 0 (X): 13
           Arg 1 (Y): 1
```

and so on. This means that foo was called with two arguments, whose names (in the Lisp source code) are x and y. The current values of x and y are 13 and 1 respectively. These may not be the original arguments if foo happens to setq its argument variables.

The CLEAR-SCREEN (or Control-L) command clears the screen, retypes the error message that was initially printed when the debugger was entered, and then prints out a description of the current frame, in the above format.

Several commands are provided to allow you to examine the Lisp control stack and to make other frames current than the one which got the error. The control stack (or "regular pdl") keeps a record of all functions which are currently active. If you call foo at Lisp's top level, and it calls bar, which in turn calls baz, and baz gets an error, then a backtrace (a backwards trace of the stack) would show all of this information. The debugger has two backtrace commands. Control-B simply prints out the names of the functions on the stack; in the above example it would print

```
        BAZ ← BAR ← FOO ← SI:*EVAL ← SI:LISP-TOP-LEVEL1 ← SI:LISP-TOP-LEVEL
```

The arrows indicate the direction of calling. The Meta-B command prints a more extensive backtrace, indicating the names of the arguments to the functions and their current values; for the example above it might look like:

```
        BAZ:
           Arg 0 (X): 13
           Arg 1 (Y): 1

        BAR:
           Arg 0 (ADDEND): 13

        FOO:
           Arg 0 (FROB): (A B C . D)
```
and so on.

The Control-N command moves "down" to the "next" frame (that is, it changes the current frame to be the frame which called it), and prints out the frame in this same format. Control-P moves "up" to the "previous" frame (the one which this one called), and prints out the frame in the same format. Meta-< moves to the top of the stack, and Meta-> to the bottom; both print out the new current frame. Control-S asks you for a string, and searches the stack for a frame whose executing function's name contains that string. That frame becomes current and is printed out. These commands are easy to remember since they are analogous to editor commands.

Meta-L prints out the current frame in "full screen" format, which shows the arguments and their values, the local variables and their values, and the machine code with an arrow pointing to the next instruction to be executed. Refer to chapter 27, page 469 for help in reading this machine code.

Meta-N moves to the next frame and prints it out in full-screen format, and Meta-P moves to the previous frame and prints it out in full-screen format. Meta-S is like Control-S but does a full-screen display.

Commands such as Control-N and Meta-N, which are meaningful to repeat, take a prefix numeric argument and repeat that many types. The numeric argument is typed by using Control- or Meta- and the number keys, as in the editor.

Control-E puts you into the editor, looking at the source code for the function in the current frame. This is useful when you have found a function which caused the error and needs to be fixed. The editor command Control-Z will return to the error handler, if it is still there.

Meta-C is similar to Control-C, but in the case of an unbound variable or undefined function, actually setqs the variable or defines the function, so that the error will not happen again. Control-C (or RESUME) provides a replacement value but does not actually change the variable.

Control-R is used to return a value from the current frame; the frame that called that frame continues running as if the function of the current frame had returned. This command prompts you for a form, which it will evaluate; it returns the resulting value, possibly after confirming it with you.

The Control-T command does a *throw to a given tag with a given value; you are prompted for the tag and the value.

Control-Meta-R is a variation of Control-R; it starts the current frame over with the same function and arguments. If the function has been redefined in the meantime (perhaps you edited it and fixed its bug) the new definition is used. Control-Meta-R asks for confirmation before doing it.

The Control-Meta-N, Control-Meta-P, and Control-Meta-B commands are like the corresponding Control- commands but don't censor the stack. When running interpreted code, the error handler tries to skip over frames that belong to functions of the interpreter, such as ***eval**, **prog**, and **cond**, and only show "interesting" functions. The Control-Meta-U command goes up the stack to the next interesting function, and makes that the current frame.

Control-Meta-A takes a numeric argument *n*, and prints out the value of the *n*th argument of the current frame. It leaves * set to the value of the argument, so that you can use the Lisp **read-eval-print** loop to examine it. It also leaves + set to a locative pointing to the argument on the stack, so that you can change that argument (by calling **rplacd** on the locative). Control-Meta-L is similar, but refers to the *n*th local variable of the frame. Control-Meta-F is similar but refers to the function executing in the frame; it ignores its numeric argument and doesn't allow you to change the function.

Control-Meta-W calls the window error handler, a display-oriented debugger which is not documented in this manual. It should, however, be usable without further documentation.

26.2.4 Summary of Commands

Control-A	Print argument list of function in current frame.
Control-Meta-A	Examine or change the *n*th argument of the current frame.
Control-B	Print brief backtrace.
Meta-B	Print longer backtrace.
Control-Meta-B	Print longer backtrace with no censoring of interpreter functions.
Control-C or RESUME	Attempt to continue.
Meta-C	Attempt to continue, **setq**ing the unbound variable or otherwise "permanently" fixing the error.
Control-Meta-C	Attempt to restart (see the **error-restart** special form, page 446).
Control-E	Edit the source code for the function in the current frame.
Control-Meta-F	Set * to the function in the current frame.
Control-G or ABORT	Quit to command level. This is not a command, but something you can type to escape from typing in a form.
Control-L or CLEAR SCREEN	Redisplay error message and current frame.
Meta-L	Full-screen typeout of current frame.
Control-Meta-L	Get local variable *n*.

Control-N or LINE	Move to next frame. With argument, move down *n* frames.
Meta-N	Move to next frame with full-screen typeout. With argument, move down *n* frames.
Control-Meta-N	Move to next frame even if it is "uninteresting". With argument, move down *n* frames.
Control-P or RETURN	Move to previous frame. With argument, move up *n* frames.
Meta-P	Move to previous frame with full-screen typeout. With argument, move up *n* frames.
Control-Meta-N	Move to previous frame even if it is "uninteresting". With argument, move up *n* frames.
Control-R	Return a value from the current frame.
Meta-R	Return multiple values from the current frame (doesn't work currently).
Control-Meta-R	Reinvoke the function in the current frame (throw back to it and start it over at its beginning.)
Control-S	Search for a frame containing a specified function.
Meta-S	Same as control-S but does a full display.
Control-T	Throw a value to a tag.
Control-Meta-U	Move up the stack to the previous "interesting" frame.
Control-Meta-W	Call the window error handler.
Control-Z or ABORT	Abort the computation and throw back to the most recent break or debugger, to the program's "command level", or to Lisp top level.
? or Help	Print a help message.
Meta-<	Go to top of stack.
Meta->	Go to bottom of stack.
Control-0 through Control-Meta-9	Numeric arguments to the following command are specified by typing a decimal number with Control and/or Meta held down.

26.3 Tracing Function Execution

The trace facility allows the user to *trace* some functions. When a function is traced, certain special actions will be taken when it is called, and when it returns. The default tracing action is to print a message when the function is called, showing its name and arguments, and another message when the function returns, showing its name and value(s).

The trace facility is closely compatible with Maclisp. One invokes it through the trace and untrace special forms, whose syntax is described below. Alternatively, you can use the trace system by clicking "trace" in the system menu, or by using the "meta-X Trace" command in the editor. This allows you to select the trace options from a menu instead of having to remember the following syntax.

trace *Special Form*

A **trace** form looks like:

```
        (trace spec-1 spec-2 ...)
```

Each *spec* can take any of the following forms:

a symbol
: This is a function name, with no options. The function will be traced in the default way, printing a message each time it is called and each time it returns.

a list (*function-name option-1 option-2* ...)
: *function-name* is a symbol and the *options* control how it is to be traced. The various options are listed below. Some options take "arguments", which should be given immediately following the option name.

a list (:function *function-spec option-1 option-2* ...)
: This is like the previous form except that *function-spec* need not be a symbol (see section 10.2, page 136). It exists because if *function-name* was a list in the previous form, it would instead be interpreted as the following form:

a list ((*function-1 function-2*...) *option-1 option-2* ...)
: All of the functions are traced with the same options. Each *function* can be either a symbol or a general function-spec.

The following **trace** options exist:

:break *pred*	Causes a breakpoint to be entered after printing the entry trace information but before applying the traced function to its arguments, if and only if *pred* evaluates to non-nil. During the breakpoint, the symbol **arglist** is bound to a list of the arguments of the function.
:exitbreak *pred*	This is just like **break** except that the breakpoint is entered after the function has been executed and the exit trace information has been printed, but before control returns. During the breakpoint, the symbol **arglist** is bound to a list of the arguments of the function, and the symbol **values** is bound to a list of the values that the function is returning.
:error	Causes the error handler to be called when the function is entered. Use RESUME (or Control-C) to continue execution of the function. If this option is specified, there is no printed trace output other than the error message printed by the error handler.
:step	Causes the function to be single-stepped whenever it is called. See the documentation on the step facility, section 26.5, page 464.
:entrycond *pred*	Causes trace information to be printed on function entry only if *pred* evaluates to non-nil.
:exitcond *pred*	Causes trace information to be printed on function exit only if *pred* evaluates to non-nil.
:cond *pred*	This specifies both :exitcond and :entrycond together.

:wherein *function*
: Causes the function to be traced only when called, directly or indirectly, from the specified function *function*. One can give several trace specs to trace, all specifying the same function but with different wherein options, so that the function is traced in different ways when called from different functions.

  This is different from advise-within, which only affects the function being advised when it is called directly from the other function. The trace :wherein option means that when the traced function is called, the special tracing actions occur if the other function is the caller of this function, or its caller's caller, or its caller's caller's caller, etc.

:argpdl *pdl*
: This specifies a symbol *pdl*, whose value is initially set to nil by trace. When the function is traced, a list of the current recursion level for the function, the function's name, and a list of arguments is consed onto the *pdl* when the function is entered, and cdr'ed back off when the function is exited. The *pdl* can be inspected from within a breakpoint, for example, and used to determine the very recent history of the function. This option can be used with or without printed trace output. Each function can be given its own pdl, or one pdl may serve several functions.

:entryprint *form*
: The *form* is evaluated and the value is included in the trace message for calls to the function. You can give this option more than once, and all the values will appear, preceded by \\.

:exitprint *form*
: The *form* is evaluated and the value is included in the trace message for returns from the function. You can give this option more than once, and all the values will appear, preceded by \\.

:print *form*
: The *form* is evaluated and the value is included in the trace messages for both calls to and returns from the function. You can give this option more than once, and all the values will appear, preceded by \\.

:entry *list*
: This specifies a list of arbitrary forms whose values are to be printed along with the usual entry-trace. The list of resultant values, when printed, is preceded by \\ to separate it from the other information.

:exit *list*
: This is similar to entry, but specifies expressions whose values are printed with the exit-trace. Again, the list of values printed is preceded by \\.

:arg :value :both nil
: These specify which of the usual trace printouts should be enabled. If :arg is specified, then on function entry the name of the function and the values of its arguments will be printed. If :value is specified, then on function exit the returned value(s) of the function will be printed. If :both is specified, both of these will be printed. If nil is specified, neither will be printed. If none of these four options are specified the default is to :both. If any further *options* appear after one of these, they will not be treated as options! Rather, they will be considered to be arbitrary forms whose values are to be printed on entry and/or exit to the function, along with the normal trace information. The values printed will be preceded by a //, and follow any values specified by :entry or :exit. Note that since these options "swallow" all following options, if one is

given it should be the last option specified.

If the variable **arglist** is used in any of the expressions given for the :cond, :break, :entry, or :exit options, or after the :arg, :value, :both, or nil option, when those expressions are evaluated the value of **arglist** will be bound to a list of the arguments given to the traced function. Thus

```
(trace (foo :break (null (car arglist))))
```

would cause a break in **foo** if and only if the first argument to **foo** is **nil**. If the :break or :error option is used, the variable **arglist** will be valid inside the break-loop. If you **setq arglist**, the arguments seen by the function will change. **arglist** should perhaps have a colon, but it can be omitted because this is the name of a system function and therefore global.

Similarly, the variable **values** will be a list of the resulting values of the traced function. For obvious reasons, this should only be used with the :exit option. If the :exitbreak option is used, the variables **values** and **arglist** are valid inside the break-loop. If you **setq values**, the values returned by the function will change. **values** should perhaps have a colon, but it can be omitted because this is the name of a system function and therefore global.

The trace specifications may be "factored", as explained above. For example,

```
(trace ((foo bar) :break (bad-p arglist) :value))
```

is equivalent to

```
(trace (foo :break (bad-p arglist) :value)
       (bar :break (bad-p arglist) :value))
```

Since a list as a function name is interpreted as a list of functions, non-atomic function names (see section 10.2, page 136) are specified as follows:

```
(trace (:function (:method flavor :message) :break t))
```

trace returns as its value a list of names of all functions it traced. If called with no arguments, as just **(trace)**, it returns a list of all the functions currently being traced.

If you attempt to trace a function already being traced, **trace** calls **untrace** before setting up the new trace.

Tracing is implemented with encapsulation (see section 10.10, page 153), so if the function is redefined (e.g. with **defun** or by loading it from a QFASL file) the tracing will be transferred from the old definition to the new definition.

Tracing output is printed on the stream which is the value of **trace-output**. This is synonymous with **terminal-io** unless you change it.

untrace *Special Form*

untrace is used to undo the effects of **trace** and restore functions to their normal, untraced state. **untrace** will take multiple specifications, e.g. **(untrace foo quux fuphoo)**. Calling **untrace** with no arguments will untrace all functions currently being traced.

Unlike Maclisp, if there is an error **trace** (or **untrace**) will invoke the error system and give an English message, instead of returning lists with question marks in them. Also, the **remtrace** function is not provided, since it is unnecessary.

trace-compile-flag *Variable*

If the value of trace-compile-flag is non-nil, the functions created by trace will get compiled, allowing you to trace special forms such as cond without interfering with the execution of the tracing functions. The default value of this flag is nil.

26.4 Advising a Function

To advise a function is to tell it to do something extra in addition to its actual definition. It is done by means of the function advise. The something extra is called a piece of advice, and it can be done before, after, or around the definition itself. The advice and the definition are independent, in that changing either one does not interfere with the other. Each function can be given any number of pieces of advice.

Advising is fairly similar to tracing, but its purpose is different. Tracing is intended for temporary changes to a function to give the user information about when and how the function is called and when and with what value it returns. Advising is intended for semi-permanent changes to what a function actually does. The differences between tracing and advising are motivated by this difference in goals.

Advice can be used for testing out a change to a function in a way which is easy to retract. In this case, you would call advise from the terminal. It can also be used for customizing a function which is part of a program written by someone else. In this case you would be likely to put a call to advise in one of your source files or your login init file (see page 506), rather than modifying the other person's source code.

Advising is implemented with encapsulation (see section 10.10, page 153), so if the function is redefined (e.g. with defun or by loading it from a QFASL file) the advice will be transferred from the old definition to the new definition.

advise *Special Form*

A function is advised by the special form

```
(advise function class name position
  form1 form2...)
```

None of this is evaluated. *function* is the function to put the advice on. It is usually a symbol, but any function spec is allowed (see section 10.2, page 136). The *forms* are the advice; they get evaluated when the function is called. *class* should be either :before, :after, or :around, and says when to execute the advice (before, after, or around the execution of the definition of the function). The meaning of :around advice is explained a couple of sections below.

name is used to keep track of multiple pieces of advice on the same function. *name* is an arbitrary symbol which is remembered as the name of this particular piece of advice. If you have no name in mind, use nil; then we say the piece of advice is anonymous. A given function and class can have any number of pieces of anonymous advice, but it can have only one piece of named advice for any one name. If you try to define a second one, it replaces the first. Advice for testing purposes is usually anonymous. Advice used for customizing someone else's program should usually be named so that multiple customizations to one function have separate names. Then, if you reload a customization that is already loaded, it does not get put on twice.

position says where to put this piece of advice in relation to others of the same class already present on the same function. If *position* is *nil*, the new advice goes in the default position: it usually goes at the beginning (where it is executed before the other advice), but if it is replacing another piece of advice with the same name, it goes in the same place that the old piece of advice was in.

If you wish to specify the position, *position* can be the numerical index of which existing piece of advice to insert this one before. Zero means at the beginning; a very large number means at the end. Or, *position* can be the name of an existing piece of advice of the same class on the same function; the new advice is inserted before that one.

For example,
```
    (advise factorial :before negative-arg-check nil
      (if (minusp (first arglist))
          (ferror nil 'factorial of negative argument")))
```
This modifies the factorial function so that if it is called with a negative argument it signals an error instead of running forever.

unadvise *Special Form*

`(unadvise` *function class position*`)`

removes pieces of advice. None of its "arguments" are evaluated. *function* and *class* have the same meaning as they do in the function advise. *position* specifies which piece of advice to remove. It can be the numeric index (zero means the first one) or it can be the name of the piece of advice.

unadvise can remove more than one piece of advice if some of its arguments are missing. If *position* is missing or nil, then all advice of the specified class on the specified function is removed. If *class* is missing or nil as well, then all advice on the specified function is removed. (unadvise) removes all advice on all functions, since *function* is not specified.

The following are the primitive functions for adding and removing advice. Unlike the above special forms, these are functions and can be conveniently used by programs. advise and unadvise are actually macros which expand into calls to these two.

si:advise-1 *function class name position forms*

Adds advice. The arguments have the same meaning as in advise. Note that the *forms* argument is *not* a &rest argument.

si:unadvise-1 *function* &optional *class position*

Removes advice. If *class* or *position* is nil or unspecified, all classes of advice or advice at all positions/with all names is removed.

You can find out manually what advice a function has with grindef, which grinds the advice on the function as forms which are calls to advise. These are in addition to the definition of the function.

To poke around in the advice structure with a program, you must work with the encapsulation mechanism's primitives. See section 10.10, page 153.

si:advised-functions *Variable*
A list of all functions which have been advised.

26.4.1 Designing the Advice

For advice to interact usefully with the definition and intended purpose of the function, it must be able to interface to the data flow and control flow through the function. We provide conventions for doing this.

The list of the arguments to the function can be found in the variable arglist. :before advice can replace this list, or an element of it, to change the arguments passed to the definition itself. If you replace an element, it is wise to copy the whole list first with

```
(setq arglist (copylist arglist))
```

After the function's definition has been executed, the list of the values it returned can be found in the variable values. :after advice can set this variable or replace its elements to cause different values to be returned.

All the advice is executed within a prog, so any piece of advice can exit the entire function with return. The arguments of the return will be returned as the values of the function. No further advice will be executed. If a piece of :before advice does this, then the function's definition will not even be called.

26.4.2 :around Advice

A piece of :before or :after advice is executed entirely before or entirely after the definition of the function. :around advice is wrapped around the definition; that is, the call to the original definition of the function is done at a specified place inside the piece of :around advice. You specify where by putting the symbol :do-it in that place.

For example, (+ 5 :do-it) as a piece of :around advice would add 5 to the value returned by the function. This could also be done by (setq values (list (+ 5 (car values)))) as :after advice.

When there is more than one piece of :around advice, they are stored in a sequence just like :before and :after advice. Then, the first piece of advice in the sequence is the one started first. The second piece is substituted for :do-it in the first one. The third one is substituted for :do-it in the second one. The original definition is substituted for :do-it in the last piece of advice.

:around advice can access arglist, but values is not set up until the outermost :around advice returns. At that time, it is set to the value returned by the :around advice. It is reasonable for the advice to receive the values of the :do-it (e.g. with multiple-value-list) and fool with them before returning them (e.g. with values-list).

:around advice can return from the prog at any time, whether the original definition has been executed yet or not. It can also override the original definition by failing to contain :do-it. Containing two instances of :do-it may be useful under peculiar circumstances. If you are careless, the original definition may be called twice, but something like

```
    (if (foo) (+ 5 :do-it) (* 2 :do-it))
```

will certainly work reasonably

26.4.3 Advising One Function Within Another

It is possible to advise the function foo only for when it is called directly from a specific other function bar. You do this by advising the function specifier (:within bar foo). That works by finding all occurrences of foo in the definition of bar and replacing them with altered-foo-within-bar. This can be done even if bar's definition is compiled code. This symbol starts off with foo as its definition; then it, rather than foo, is advised. The system remembers that foo has been replaced inside bar, so that if you change the definition of bar, or advise it, then the replacement is propagated to the new definition or to the advice. If you remove all the advice on (:within bar foo), so that its definition becomes the symbol foo again, then the replacement is unmade and everything returns to its original state.

(grindef bar) will print foo where it originally appeared, rather than altered-foo-within-bar, so the replacement will not be seen. Instead, grindef will print out calls to advise to describe all the advice that has been put on foo or anything else within bar.

An alternate way of putting on this sort of advice is to use advise-within.

advise-within *Special Form*

(advise-within *within-function function-to-advise class name position forms...*)

advises *function-to-advise* only when called directly from the function *within-function*. The other arguments mean the same thing as with advise. None of them are evaluated.

To remove advice from (:within bar foo), you can use unadvise on that function specifier. Alternatively, you can use unadvise-within.

unadvise-within *Special Form*

(unadvise-within *within-function function-to-advise class position*)

removes advice which has been placed on (:within *within-function function-to-advise*). The arguments *class* and *position* are interpereted as for unadvise. For example, if those two are omitted, then all advice placed on *function-to-advise* within *within-function* is removed. Additionally, if *function-to-advise* is omitted, all advise on any function within *within-function* is removed. If there are no arguments, than all advice on one function within another is removed. Other pieces of advice, which have been placed on one function and not limited to within another, are not removed.

(unadvise) removes absolutely all advice, including advice for one function within another.

The function versions of **advise-within** and **unadvise-within** are called **si:advise-within-1** and **si:unadvise-within-1**. **advise-within** and **unadvise-within** are macros which expand into calls to the other two.

26.5 Stepping Through an Evaluation

The Step facility gives you the ability to follow every step of the evaluation of a form, and examine what is going on. It is analogous to a single-step proceed facility often found in machine-language debuggers. If your program is doing something strange, and it isn't obvious how it's getting into its strange state, then the stepper is for you.

There are two ways to enter the stepper. One is by use of the **step** function.

step *form*
This evaluates *form* with single stepping. It returns the value of *form*.

For example, if you have a function named **foo**, and typical arguments to it might be **t** and **3**, you could say

```
(step '(foo t 3))
```

and the form **(foo t 3)** will be evaluated with single stepping.

The other way to get into the stepper is to use the **:step** option of **trace** (see page 457). If a function is traced with the **:step** option, then whenever that function is called it will be single stepped.

Note that any function to be stepped must be interpreted; that is, it must be a lambda-expression. Compiled code cannot be stepped by the stepper.

When evaluation is proceeding with single stepping, before any form is evaluated, it is (partially) printed out, preceded by a forward arrow (→) character When a macro is expanded, the expansion is printed out preceded by a double arrow (↔) character. When a form returns a value, the form and the values are printed out preceded by a backwards arrow (←) character; if there is more than one value being returned, an and-sign (∧) character is printed between the values.

Since the forms may be very long, the stepper does not print all of a form; it truncates the printed representation after a certain number of characters. Also, to show the recursion pattern of who calls whom in a graphic fashion, it indents each form proportionally to its level of recursion.

After the stepper prints any of these things, it waits for a command from the user. There are several commands to tell the stepper how to proceed, or to look at what is happening. The commands are:

Control-N (Next)
Step to the Next thing. The stepper continues until the next thing to print out, and it accepts another command.

Space Go to the next thing at this level. In other words, continue to evaluate at this level, but don't step anything at lower levels. This is a good way to skip over parts of the evaluation that don't interest you.

Control-U (Up)
: Continue evaluating until we go up one level. This is like the space command, only more so; it skips over anything on the current level as well as lower levels.

Control-X (eXit)
: Exit; finish evaluating without any more stepping.

Control-T (Type)
: Retype the current form in full (without truncation).

Control-G (Grind)
: Grind (i.e. prettyprint) the current form.

Control-E (Editor)
: Editor escape (enter the editor).

Control-B (Breakpoint)
: Breakpoint. This command puts you into a breakpoint (i.e. a read-eval-print loop) from which you can examine the values of variables and other aspects of the current environment. From within this loop, the following variables are available:

step-form	which is the current form.
step-values	which is the list of returned values.
step-value	which is the first returned value.

If you change the values of these variables, it will work.

Control-L
: Clear the screen and redisplay the last 10. pending forms (forms which are being evaluated).

Meta-L
: Like Control-L, but doesn't clear the screen.

Control-Meta-L
: Like Control-L, but redisplays all pending forms.

? or Help
: Prints documentation on these commands.

It is strongly suggested that you write some little function and try the stepper on it. If you get a feel for what the stepper does and how it works, you will be able to tell when it is the right thing to use to find bugs.

26.6 Evalhook

The evalhook facility provides a "hook" into the evaluator; it is a way you can get a Lisp form of your choice to be executed whenever the evaluator is called. The stepper uses evalhook, and usually it is the only thing that ever needs to. However, if you want to write your own stepper or something similar, this is the primitive facility that you can use to do so. The way this works is a bit hairy, but unless you need to write your own stepper you don't have to worry about it.

evalhook *Variable*

If the value of evalhook is non-nil, then special things happen in the evaluator. When a form (any form, even a number or a symbol) is to be evaluated, evalhook is bound to nil and the function which was evalhook's value is applied to one argument—the form that was trying to be evaluated. The value it returns is then returned from the evaluator.

evalhook is bound to nil by break and by the error handler, and setq'ed to nil when errors are dismissed by throwing to the Lisp top level loop. This provides the ability to escape from this mode if something bad happens.

In order not to impair the efficiency of the Lisp interpreter, several restrictions are imposed on evalhook. It only applies to evaluation—whether in a read-eval-print loop, internally in evaluating arguments in forms, or by explicit use of the function eval. It *does not* have any effect on compiled function references, on use of the function apply, or on the "mapping" functions. (In Zetalisp, as opposed to Maclisp, it is not necessary to do (*rset t) nor (sstatus evalhook t).) (Also, Maclisp's special-case check for store is not implemented.)

evalhook *form hook*

evalhook is a function which helps exploit the evalhook feature. The *form* is evaluated with evalhook lambda-bound to the function *hook*. The checking of evalhook is bypassed in the evaluation of *form* itself, but not in any subsidiary evaluations, for instance of arguments in the *form*. This is like a "one-instruction proceed" in a machine-language debugger.

Example:

```
;; This function evaluates a form while printing debugging information.
(defun hook (x)
   (terpri)
   (evalhook x 'hook-function))

;; Notice how this function calls evalhook to evaluate the form f,
;; so as to hook the sub-forms.
(defun hook-function (f)
   (let ((v (evalhook f 'hook-function)))
     (format t "form: ~s~%value: ~s~%" f v)
     v))

;; This isn't a very good program, since iff uses multiple
;; values, it will not work.
```

The following output might be seen from (hook '(cons (car '(a . b)) 'c)):

```
        form: (quote (a . b))
        value: (a . b)
        form: (car (quote (a . b)))
        value: a
        form: (quote c)
        value: c
        (a . c)
```

26.7 The MAR

The MAR facility allows any word or contiguous set of words to be monitored constantly, and can cause an error if the words are referenced in a specified manner. The name MAR is from the similar device on the ITS PDP-10's; it is an acronym for "Memory Address Register". The MAR checking is done by the Lisp Machine's memory management hardware, and so the speed of general execution when the MAR is enabled is not significantly slowed down. However, the speed of accessing pages of memory containing the locations being checked is slowed down somewhat, since every reference involves a microcode trap.

These are the functions that control the MAR:

set-mar *location cycle-type* &optional *n-words*

The set-mar function clears any previous setting of the MAR, and sets the MAR on *n-words* words, starting at *location*. *location* may be any object. Often it will be a locative pointer to a cell, probably created with the locf special form. *n-words* currently defaults to 1, but eventually it may default to the size of the object. *cycle-type* says under what conditions to trap. :read means that only reading the location should cause an error, :write means that only writing the location should, t means that both should. To set the MAR to detect setq (and binding) of the variable foo, use

```
        (set-mar (value-cell-location 'foo) ':write)
```

clear-mar

This turns off the MAR. Warm-booting the machine disables the MAR but does not turn it off, i.e. references to the MARed pages are still slowed down. clear-mar does not currently speed things back up until the next time the pages are swapped out; this may be fixed some day.

mar-mode

(mar-mode) returns a symbol indicating the current state of the MAR. It returns one of:

nil	The MAR is not set.
:read	The MAR will cause an error if there is a read.
:write	The MAR will cause an error if there is a write.
t	The MAR will cause an error if there is any reference.

Note that using the MAR makes the pages on which it is set somewhat slower to access, until the next time they are swapped out and back in again after the MAR is shut off. Also, use of the MAR currently breaks the read-only feature if those pages were read-only.

Proceeding from a MAR break allows the memory reference that got an error to take place, and continues the program with the MAR still effective. When proceeding from a write, the error handler asks you whether to allow the write to take place or to inhibit it, leaving the location with its old contents.

Most—but not all—write operations first do a read. **setq** and **rplaca** are examples. This means that if the MAR is in **:read** mode it will catch writes as well as reads, however they will trap during the reading phase, and consequently the data to be written will not be displayed. This also means that setting the MAR to **t** mode causes most writes to trap twice, first for a read and then again for a write. So when the MAR says that it trapped because of a read, this means a read at the hardware level, which may not look like a read in your program.

26.8 Variable Monitoring

monitor-variable *var* &optional *current-value-cell-only-p monitor-function*

Calls a given function just after a given special variable is **setq**'ed (by compiled code or otherwise). Does not trigger on binding of the variable. The function is given both the old and new values as arguments. It does not get the name of the variable as an argument, so it is usually necessary to use a closure as *monitor-function* in order to remember this. The old value will be **nil** if the variable had been unbound.

The default monitoring function just prints the symbol and the old and new values. This behavior can be changed by specifying the *monitor-function* argument.

Normally this feature applies to all **setq**'s, but if *current-value-cell-only-p* is specified **non-nil**, it applies only to those **setq**'s which would alter the variable's currently active value cell. This is only relevant when *var* is subject to a closure.

Don't try to use this with variables that are forwarded to A memory (e.g. **inhibit-scheduling-flag**).

unmonitor-variable &optional *var*

If *var* is being monitored, it is restored to normal. If no *var* is specified, all variables that have been monitored are unmonitored.

27. How to Read Assembly Language

Sometimes it is useful to study the machine language code produced by the Lisp Machine's compiler, usually in order to analyze an error, or sometimes to check for a suspected compiler problem. This chapter explains how the Lisp Machine's instruction set works, and how to understand what code written in that instruction set is doing. Fortunately, the translation between Lisp and this instruction set is very simple; after you get the hang of it, you can move back and forth between the two representations without much trouble. The following text does not assume any special knowledge about the Lisp Machine, although it sometimes assumes some general computer science background knowledge.

27.1 Introduction

Nobody looks at machine language code by trying to interpret octal numbers by hand. Instead, there is a program called the Disassembler which converts the numeric representation of the instruction set into a more readable textual representation. It is called the Disassembler because it does the opposite of what an Assembler would do; however, there isn't actually any assembler that accepts this input format, since there is never any need to manually write assembly language for the Lisp Machine.

The simplest way to invoke the Disassembler is with the Lisp function **disassemble**. Here is a simple example. Suppose we type:

```
(defun foo (x)
  (assq 'key (get x 'propname)))

(compile 'foo)

(disassemble 'foo)
```

This defines the function **foo**, compiles it, and invokes the Disassembler to print out the textual representation of the result of the compilation. Here is what it looks like:

```
22 MOVE D-PDL FEF|6          ;'KEY
23 MOVE D-PDL ARG|0          ;X
24 MOVE D-PDL FEF|7          ;'PROPNAME
25 (MISC) GET D-PDL
26 (MISC) ASSQ D-RETURN
```

The Disassembler is also used by the Error Handler and the Inspector. When you see stuff like the above while using one of these programs, it is disassembled code, in the same format as the **disassemble** function uses. Inspecting a compiled code object shows the disassembled code.

Now, what does this mean? Before we get started, there is just a little bit of jargon to learn.

The acronym PDL stands for Push Down List, and means the same thing as Stack: a last-in first-out memory. The terms PDL and stack will be used interchangeably. The Lisp Machine's architecture is rather typical of "stack machines"; there is a stack that most instructions deal with, and it is used to hold values being computed, arguments, and local variables, as well as flow-of-control information. An important use of the stack is to pass arguments to instructions, though not all instructions take their arguments from the stack.

The acronym "FEF" stands for Function Entry Frame. A FEF is a compiled code object produced by the compiler. After the defun form above was evaluated, the function cell of the symbol foo contained a lambda expression. Then, we compiled the function foo, and the contents of the function cell were replaced by a "FEF" object. The printed representation of the "FEF" object for foo looks like this:

```
#<DTP-FEF-POINTER 11464337 FOO>
```

The FEF has three parts (this is a simplified explanation): a header with various fixed-format fields, a part holding constants and invisible pointers, and the main body holding the machine language instructions. The first part of the FEF, the header, is not very interesting and is not documented here (you can look at it with describe but it won't be easy to understand what it all means). The second part of the FEF holds various constants referred to by the function; for example, our function foo references two constants (the symbols key and propname), and so (pointers to) those symbols are stored in the FEF. This part of the FEF also holds invisible pointers to the value cells of all symbols that the function uses as variables, and invisible pointers to the function cells of all symbols that the function calls as functions. The third part of the FEF holds the machine language code itself.

Now we can read the disassembled code. The first instruction looked like this:

```
22 MOVE D-PDL FEF|6             ;'KEY
```

This instruction has several parts. The 22 is the address of this instruction. The Disassembler prints out the address of each instruction before it prints out the instruction, so that you can interpret branching instructions when you see them (we haven't seen one of these yet, but we will later). The MOVE is an opcode: this is a MOVE instruction, which moves a datum from one place to another. The D-PDL is a destination specification. The D stands for "Destination", and so D-PDL means "Destination-PDL": the destination of the datum being moved is the PDL. This means that the result will be pushed onto the PDL, rather than just moved to the top; this instruction is pushing a datum onto the stack. The next field of the instruction is FEF|6. This is an *address*, and it specifies where the datum is coming from. The vertical bar serves to separate the two parts of the address. The part before the vertical bar can be thought of as a *base register*, and the part after the bar can be thought of as being an offset from that register. FEF as a base register means the address of the FEF that we are disassembling, and so this address means the location six words into the FEF. So what this instruction does is to take the datum located six words into the FEF, and push it onto the PDL. The instruction is followed by a "comment" field, which looks like ;'KEY. This is not a comment that any person wrote; the disassembler produces these to explain what is going on. The semicolon just serves to start the comment, the way semicolons in Lisp code do. In this case, the body of the comment, 'KEY, is telling us that the address field (FEF|6) is addressing a constant (that is what the single-quote in 'KEY means), and that the printed representation of that constant is KEY. With the help of this "comment" we finally get the real story about what this instruction is doing: it is pushing (a pointer to) the symbol key onto the stack.

The next instruction looks like this:

```
23 MOVE D-PDL ARG|0            ;X
```

This is a lot like the previous instruction; the only difference is that a different "base register" is being used in the address. The ARG "base register" is used for addressing your arguments: ARG|0 means that the datum being addressed is the zeroth argument. Again, the "comment" field explains what that means: the value of X (which was the zeroth argument) is being pushed onto the stack.

The third instruction is just like the first one; it pushes the symbol propname onto the stack.

The fourth instruction is something new:

```
25 (MISC) GET D-PDL
```

The first thing we see here is (MISC). This means that this is one of the so-called *miscellaneous* instructions. There are quite a few of these instructions. With some exceptions, each miscellaneous instruction corresponds to a Lisp function and has the same name as that Lisp function. If a Lisp function has a corresponding miscellaneous instruction, then that function is hand-coded in Lisp Machine microcode.

Miscellaneous instructions only have a destination field; they don't have any address field. The inputs to the instruction come from the stack: the top *n* elements on the stack are used as inputs to the instruction and popped off the stack, where *n* is the number of arguments taken by the function. The result of the function is stored wherever the destination field says. In our case, the function being executed is get, a Lisp function of two arguments. The top two values will be popped off the stack and used as the arguments to get (the value pushed first is the first argument, the value pushed second is the second argument, and so on). The result of the call to get will be sent to the destination D-PDL; that is, it will be pushed onto the stack. (In case you were wondering about how we handle optional arguments and multiple-value returns, the answer is very simple: functions that use either of those features cannot be miscellaneous instructions!) (If you are curious as to what functions are hand-microcoded and thus available as miscellaneous instructions, you can look at the defmic forms in the file "AI: LISPM; DEFMIC >".)

The fifth and last instruction is similar to the fourth:

```
26 (MISC) ASSQ D-RETURN
```

What is new here is the new value of the destination field. This one is called D-RETURN, and it can be used anywhere destination fields in general can be used (like in MOVE instructions). Sending something to "Destination-Return" means that this value should be the returned value of the function, and that we should return from this function. This is a bit unusual in instruction sets; rather than having a "return" instruction, we have a destination which, when stored into, returns from the function. What this instruction does, then, is to invoke the Lisp function assq on the top two elements of the stack, and return the result of assq as the result of this function.

Now, let's look at the program as a whole and see what it did:

```
22 MOVE D-PDL FEF|6              ;'KEY
23 MOVE D-PDL ARG|0              ;X
24 MOVE D-PDL FEF|7              ;'PROPNAME
25 (MISC) GET D-PDL
26 (MISC) ASSQ D-RETURN
```

First it pushes the symbol key. Then it pushes the value of x. Then it pushes the symbol propname. Then it invokes get, which pops the value of x and the symbol propname off the stack and uses them as arguments, thus doing the equivalent of evaluating the form (get x 'propname). The result is left on the stack; the stack now contains the result of the get on top, and the symbol key underneath that. Next, it invokes assq on these two values, thus doing the equivalent of evaluating (assq 'key (get x 'propname)). Finally, it returns the value produced by assq. Now, the original Lisp program we compiled was:

```
(defun foo (x)
  (assq 'key (get x 'propname)))
```

We can see that the code produced by the compiler is correct: it will do the same thing as the function we defined will do.

In summary, we have seen two kinds of instructions so far: the MOVE instruction, which takes a destination and an address, and two of the large set of miscellaneous instructions, which take only a destination, and implicitly get their inputs from the stack. We have seen two destinations (D-PDL and D-RETURN), and two forms of address (FEF addressing and ARG addressing).

27.2 A More Advanced Example

Here is a more complex Lisp function, demonstrating local variables, function calling, conditional branching, and some other new instructions.

```
(defun bar (y)
  (let ((z (car y)))
    (cond ((atom z)
           (setq z (cdr y))
           (foo y))
          (t
           nil))))
```

The disassembled code looks like this:

```
20 CAR D-PDL ARG|0          ;Y
21 POP LOCAL|0              ;Z
22 MOVE D-IGNORE LOCAL|0    ;Z
23 BR-NOT-ATOM 30
24 CDR D-PDL ARG|0          ;Y
25 POP LOCAL|0              ;Z
26 CALL D-RETURN FEF|6      ;#'FOO
27 MOVE D-LAST ARG|0        ;Y
30 MOVE D-RETURN 'NIL
```

The first instruction here is a CAR instruction. It has the same format as MOVE: there is a destination and an address. The CAR instruction reads the datum addressed by the address, takes the car of it, and stores the result into the destination. In our example, the first instruction addresses the zeroth argument, and so it computes (car y); then it pushes the result onto the stack.

The next instruction is something new: the POP instruction. It has an address field, but it uses it as a destination rather than as a source. The POP instruction pops the top value off the stack, and stores that value into the address specified by the address field. In our example, the value on the top of the stack is popped off and stored into address LOCAL|0. This is a new form of address; it means the zeroth local variable. The ordering of the local variables is chosen by the compiler, and so it is not fully predictable, although it tends to be by order of appearance in the code; fortunately you never have to look at these numbers, because the "comment" field explains what is going on. In this case, the variable being addressed is z. So this instruction pops the top value on the stack into the variable z. The first two instructions work together to take the car of y and store it into z, which is indeed the first thing the function bar ought to do. (If you have two local variables with the same name, then the "comment" field won't tell you which of the two you're talking about; you'll have to figure that out yourself. You can tell two local variables with the same name apart by looking at the number in the address.)

The next instruction is a familiar MOVE instruction, but it uses a new destination: D-IGNORE. This means that the datum being addressed isn't moved anywhere. If so, then why bother doing this instruction? The reason is that there is conceptually a set of *indicator* bits, as in the PDP-11. Every instruction that moves or produces a datum sets the "indicator" bits from that datum so that following instructions can test them. So the reason that the MOVE instruction is being done is so that someone can test the "indicators" set up by the value that was moved, namely the value of z.

All instructions except the branch instructions set the "indicator" bits from the result produced and/or stored by that instruction. (In fact, the POP in instruction 21 set the "indicators" properly, and so the MOVE at instruction 22 is superfluous. However, the compiler is not clever enough to notice that.)

The next instruction is a conditional branch; it changes the flow of control, based on the values in the "indicator" bits. The instruction is BR-NOT-ATOM 30, which means "Branch, if the quantity was not an atom, to location 30; otherwise proceed with execution". If z was an atom, the Lisp Machine branches to location 30, and execution proceeds there. (As you can see by skipping ahead, location 30 just contains a MOVE instruction, which will cause the function to return nil.)

If z is not an atom, the program keeps going, and the CDR instruction is next. This is just like the CAR instruction except that it takes the cdr; this instruction pushes the value of (cdr y) onto the stack. The next one pops that value off into the variable z.

There are just two more instructions left. These two instructions will be our first example of how function calling is compiled. It is the only really tricky thing in the instruction set. Here is how it works in our example:

```
        26 CALL D-RETURN FEF|6          ;#'FOO
        27 MOVE D-LAST ARG|0            ;Y
```

The form being compiled here is (foo y). This means we are applying the function which is in the function cell of the symbol foo, and passing it one argument, the value of y. The way function calling works is in the following three steps. First of all, there is a CALL instruction that specifies the function object being applied to arguments. This creates a new stack frame on the stack, and stores the function object there. Secondly, all the arguments being passed except the last one are pushed onto the stack. Thirdly and lastly, the last argument is sent to a special destination, called D-LAST, meaning "this is the last argument". Storing to this destination is what actually calls the function, *not* the CALL instruction itself.

There are two things you might wonder about this. First of all, when the function returns, what happens to the returned value? Well, this is what we use the destination field of the CALL instruction for. The destination of the CALL is not stored into at the time the CALL instruction is executed; instead, it is saved on the stack (into the stack frame created by the CALL instruction, along with the function object). Then, when the function actually returns, its result is stored into that destination.

The other question is what happens when there isn't any last argument; that is, when there is a call with no arguments at all? This is handled by a special instruction called CALL0. The address of CALL0 addresses the function object to be called; the call takes place immediately, and the result is stored into the destination specified by the destination field of the CALL0 instruction.

So, let's look at the two-instruction sequence above. The first instruction is a CALL; the function object it specifies is at FEF|6, which the comment tells us is the contents of the function cell of foo (the FEF contains an invisible pointer to that function cell). The destination field of the CALL is D-RETURN, but we aren't going to store into it yet; we will save it away in the stack frame and use it later. So the function doesn't return at this point, even though it says D-RETURN in the instruction; this is the tricky part.

Next we have to push all the arguments except the last one. Well, there's only one argument, so nothing needs to be done here. Finally, we move the last argument (that is, the only argument: the value of y) to D-LAST, using the MOVE instruction. Moving to D-LAST is what actually invokes the function, so at this point the function foo is invoked. When it returns, its result is sent to the destination stored in the stack frame: D-RETURN. Therefore, the value returned by the call to foo will be returned as the value of the function bar. Sure enough, this is what the original Lisp code says to do.

When the compiler pushes arguments to a function call, it sometimes does it by sending the values to a destination called D-NEXT (meaning the "next" argument). This is exactly the same as D-PDL; the only reasons for the difference are very historical. They mean the same thing.

Here is another example to illustrate function calling. This Lisp function calls one function on the results of another function.

```
(defun a (x y)
  (b (c x y) y))
```

The disassembled code looks like this:

```
22 CALL D-RETURN FEF|6          ;#'B
23 CALL D-PDL FEF|7             ;#'C
24 MOVE D-PDL ARG|0             ;X
25 MOVE D-LAST ARG|1            ;Y
26 MOVE D-LAST ARG|1            ;Y
```

The first instruction starts off the call to the function b. The destination field is saved for later: when this function returns, we will return its result as a's result. Next, the call to c is started. Its destination field, too, is saved for later; when c returns, its result should be pushed onto the stack, so that it will be the next argument to b. Next, the first and second arguments to c are passed; the second one is sent to D-LAST and so the function c is called. Its result, as we said, will be pushed onto the stack, and thus become the first argument to b. Finally, the second argument to b is passed, by storing in D-LAST; b gets called, and its result is sent to D-RETURN and is returned from a.

27.3 The Rest of the Instructions

Now that we've gotten some of the feel for what is going on, I will start enumerating the instructions in the instruction set. The instructions fall into four classes. Class I instructions have both a destination and an address. Class II instructions have an address, but no destination. Class III instructions are the branch instructions, which contain a branch address rather than a general base-and-offset address. Class IV instructions have a destination, but no address; these are the miscellaneous instructions.

We have already seen just about all the Class I instructions. There are nine of them in all: MOVE, CALL, CALL0, CAR, CDR, CAAR, CADR, CDAR, and CDDR. MOVE just moves a datum from an address to a destination; the CxR and CxxR instructions are the same but perform the function on the value before sending it to the destination; CALL starts off a call to a function with some arguments; CALL0 performs a call to a function with no arguments.

We've seen most of the possible forms of address. So far we have seen the FEF, ARG, and LOCAL base registers. There are two other kinds of addresses. One uses a "constant" base register, which addresses a set of standard constants: NIL, T, 0, 1, and 2. The disassembler doesn't even bother to print out CONSTANT|*n*, since the number *n* would not be even slightly interesting; it just prints out 'NIL or '1 or whatever. The other kind of address is a special one printed as PDL-POP, which means that to read the value at this address, an object should be popped off the top of the stack.

There is a higher number of Class II instructions. The only one we've seen so far is POP, which pops a value off the stack and stores it into the specified address. There is another instruction called MOVEM (from the PDP-10 opcode name, meaning MOVE to Memory), which stores the top element of the stack into the specified address, but doesn't pop it off the stack.

Then there are seven Class II instructions to implement heavily-used two-argument functions: +, -, *, /, LOGAND, LOGXOR, and LOGIOR. These instructions take their first argument from the top of the stack (popping them off) and their second argument from the specified address, and they push their result on the stack. Thus the stack level does not change due to these instructions.

Here is a small function that shows some of these new things:

```
(defun foo (x y)
  (setq x (logxor y (- x 2))))
```

The disassembled code looks like this:

```
16 MOVE D-PDL ARG|1           ;Y
17 MOVE D-PDL ARG|0           ;X
20 - '2
21 LOGXOR PDL-POP
22 MOVEM ARG|0                ;X
23 MOVE D-RETURN PDL-POP
```

Instructions 20 and 21 use two of the new Class II instructions: the - and LOGXOR instructions. Instructions 21 and 23 use the PDL-POP address type, and instruction 20 uses the "constant" base register to get to a fixnum 2. Finally, instruction 22 uses the MOVEM instruction; the compiler wants to use the top value of the stack to store it into the value of x, but it doesn't want to pop it off the stack because it has another use for it: to return it from the function.

Another four Class II instructions implement some commonly used predicates: =, >, <, and EQ. The two arguments come from the top of the stack and the specified address; the stack is popped, the predicate is applied to the two objects, and the result is left in the "indicators" so that a branch instruction can test it and branch based on the result of the comparison. These instructions remove the top item on the stack and don't put anything back, unlike the previous set which put their results back on the stack.

Next, there are four Class II instructions to read, modify, and write a quantity in ways that are common in Lisp code. These instructions are called SETE-CDR, SETE-CDDR, SETE-1+, and SETE-1-. The SETE- means to set the addressed value to the result of applying the specified one-argument function to the present value. For example, SETE-CDR means to read the value addressed, apply cdr to it, and store the result back in the specified address. This is used when compiling (setq x (cdr x)), which commonly occurs in loops; the other functions are used frequently in loops, too.

There are two instructions used to bind special variables. The first is BIND-NIL, which binds the cell addressed by the address field to nil; the second is BIND-POP, which binds the cell to an object popped off the stack rather than nil. The latter instruction pops a value off the stack; the former does not use the stack at all.

There are two instructions to store common values into addressed cells. SET-NIL stores nil into the cell specified by the address field; SET-ZERO stores 0. Neither instruction uses the stack at all.

Finally, the PUSH-E instruction creates a locative pointer to the cell addressed by the specified address, and pushes it onto the stack. This is used in compiling (value-cell-location 'z) where z is an argument or a local variable, rather than a symbol (special variable).

Those are all of the Class II instructions. Here is a contrived example that uses some of the ones we haven't seen, just to show you what they look like:

```
(declare (special *foo* *bar*))

(defun weird (x y)
  (cond ((= x y)
         (let ((*foo* nil) (*bar* 5))
           (setq x (cdr x)))
         nil)
        (t
         (setq x nil)
         (caar (value-cell-location 'y)))))
```

The disassembled code looks like this:

```
24 MOVE D-PDL ARG|0          ;X
25 = ARG|1                   ;Y
26 BR-NIL 35
27 BIND-NIL FEF|6            ;*FOO*
30 MOVE D-PDL FEF|8          ;'5
31 BIND-POP FEF|7            ;*BAR*
32 SETE-CDR ARG|0            ;X
33 (MISC) UNBIND 2 bindings
34 MOVE D-RETURN 'NIL
35 SET-NIL ARG|0             ;X
36 PUSH-E ARG|1              ;Y
37 CAAR D-RETURN PDL-POP
```

Instruction 25 is an = instruction; it numerically compares the top of the stack, x, with the addressed quantity, y. The x is popped off the stack, and the indicators are set to the result of the equality test. Instruction 26 checks the indicators, branching to 35 if the result of the call to = was NIL; that is, the machine will branch to 35 if the two values were not equal. Instruction 27 binds *foo* to nil; instructions 30 and 31 bind *bar* to 5. Instruction 32 demonstrates the use of SETE-CDR to compile (setq x (cdr x)), and instruction 35 demonstrates the use of SET-NIL to compile (setq x nil). Instruction 36 demonstrates the use of PUSH-E to compile (value-cell-location 'y).

The next class of instructions, Class III, are the branching instructions. These have neither addresses nor destinations of the usual sort; instead, they have branch-addresses: they say where to branch, if the branch is going to happen. There are several instructions, differing in the conditions under which they will branch, and whether they pop the stack. Branch-addresses are stored internally as self-relative addresses, to make Lisp Machine code relocatable, but the disassembler does the addition for you and prints out FEF-relative addresses so that you can easily see where the branch is going to.

The branch instructions we have seen so far decide whether to branch on the basis of the "nil indicator"; that is, whether the last value dealt with was nil or non-nil. BR-NIL branches if it was nil, and BR-NOT-NIL branches if it was not nil. There are two more instructions that test the result of the atom predicate on the last value dealt with. BR-ATOM branches if the value was an atom (that is, if it was anything besides a cons). and BR-NOT-ATOM branches if the value was not an atom (that is, if it was a cons). The BR instruction is an unconditional branch (it always branches).

None of the above branching instructions deal with the stack. There are two more instructions called BR-NIL-POP and BR-NOT-NIL-POP, which are the same as BR-NIL and BR-NOT-NIL except that if the branch is not done, the top value on the stack is popped off the stack. These are used for compiling and and or special forms.

Finally, there are the Class IV instructions, most of which are miscellaneous hand-microcoded Lisp functions. The file "AI: LISPM; DEFMIC >" has a list of all the miscellaneous instructions. Most correspond to Lisp functions, including the subprimitives, although some of these functions are very low level internals that may not be documented anywhere (don't be disappointed if you don't understand all of them). Please do not look at this file in hopes of finding obscure functions that you think you can use to speed up your programs; in fact, the compiler automatically uses these things when it can, and directly calling weird internal functions will only serve to make your code hard to read, without making it any faster. In fact, we don't guarantee that calling undocumented functions will continue to work in the future.

The DEFMIC file can be useful for determining if a given function is in microcode, although the only definitive way to tell is to compile some code that uses it and look at the results, since sometimes the compiler converts a documented function with one name into an undocumented one with another name.

27.4 Function Entry

When a function is first entered in the Lisp Machine, interesting things can happen because of the features that are invoked by use of the various "&" keywords. The microcode performs various services when a function is entered, even before the first instruction of the function is executed. These services are called for by various fields of the header portion of the FEF, including a list called the *Argument Descriptor List*, or *ADL*. We won't go into the detailed format of any of this, as it is complex and the details are not too interesting. The describe function will disassemble it, but not necessarily into a readily-comprehensible form.

The function-entry services include the initialization of unsupplied optional arguments and of &AUX variables. The ADL has a little instruction set of its own, and if the form that computes the initial value is something simple, such as a constant, or just a variable, then the ADL can handle things itself. However, if things get too complicated, instructions are needed, and the compiler generates some instructions at the front of the function to initialize the unsupplied variables. In this case, the ADL specifies several different starting addresses for the function, depending on which optional arguments have been supplied and which have been omitted. If all the optional arguments are supplied, then the ADL starts the function off after all the instructions that would have initialized the optional arguments; since the arguments were supplied, their values should not be set, and so all these instructions are skipped over. Here's an example:

```
(declare (special *y*))

(defun foo (&optional (x (car *y*)) (z (* x 3)))
  (cons x z))
```

The disassembled code looks like this:

```
32 CAR D-PDL FEF|6              ;*Y*
33 POP ARG|0                    ;X
34 MOVE D-PDL ARG|0             ;X
35 * FEF|11                     ;'3
36 POP ARG|1                    ;Z
37 MOVE D-PDL ARG|0             ;X
40 MOVE D-PDL ARG|1             ;Z
41 (MISC) CONS D-RETURN
```

If no arguments are supplied, the function will be started at instruction 32; if only one argument is supplied, it will be started at instruction 34; if both arguments are supplied, it will be started at instruction 37. (If you do (describe 'foo) and look at the incomprehensible stuff that gets printed out, you can see the numbers 34 and 37 in lines that correspond to elements of the ADL.)

The thing to keep in mind here is that when there is initialization of variables, you may see it as code at the beginning of the function, or you may not, depending upon whether it is too complex for the ADL to handle. This is true of &aux variables as well as unsupplied &optional arguments.

When there is an &rest argument, it is passed to the function as the zeroth local variable, rather than as any of the arguments. This is not really so confusing as it might seem, since an &rest argument is not an argument passed by the caller, rather it is a list of some of the arguments, created by the function-entry microcode services. In any case the "comment" tells you what is going on. In fact, one hardly ever looks much at the address fields in disassembled code, since the "comment" tells you the right thing anyway. Here is a silly example of the use of an &rest argument:

```
(defun prod (&rest values)
  (apply #'* values))
```

The disassembled code looks like this:

```
20 MOVE D-PDL FEF|6           ;#'*
21 MOVE D-PDL LOCAL|0         ;VALUES
22 (MISC) APPLY D-RETURN
```

As can be seen, values is referred to as LOCAL|0.

Another thing the microcode does at function entry is to bind the values of any arguments or &aux variables that are special. Thus, you won't see BIND instructions doing this, but it is still being done.

27.5 Special Class IV Instructions

We said earlier that most of the Class IV instructions are miscellaneous hand-microcoded Lisp functions. However, a few of them are not Lisp functions at all. There are two instructions that are printed as UNBIND 3 bindings or POP 7 values (except that the number can be anything up to 16 (these numbers are printed in decimal)). These instructions just do what they say, unbinding the last *n* values that were bound, or popping the top *n* values off the stack.

There are also special instructions to implement the Lisp list function, which is special because it is a primitive which takes a variable number of arguments. Let's take a look at how the compiler handles list.

```
(defun test-list (x y)
  (list 2 x y x))
```

The disassembled code looks like this:

```
16 (MISC) LIST 4 long D-RETURN
17 MOVE D-NEXT-LIST '2
20 MOVE D-NEXT-LIST ARG|0     ;X
21 MOVE D-NEXT-LIST ARG|1     ;Y
22 MOVE D-NEXT-LIST ARG|0     ;X
```

The instruction LIST 4 long prepares for the creation of a list of four elements; it allocates the storage, but doesn't put anything into it. The destination is not used immediately, but is saved for later, just as it is with CALL. Then the objects to be passed as arguments are successively generated and sent to a special destination, D-NEXT-LIST. This causes them to be put into the storage allocated by the LIST instruction. Once the fourth such sending is done, all the elements of the list are filled in, and the result (i.e. the list itself) is sent to whatever destination was specified in the LIST instruction.

By the way, that is the last of the destination codes; now you have seen all of them. In summary, they are D-IGNORE, D-PDL (and D-NEXT, which is the same thing), D-LAST, D-RETURN, and D-NEXT-LIST.

The array referencing functions—aref, aset, and aloc—also take a variable number of arguments, but they are handled differently. For one, two, and three dimensional arrays, these functions are turned into internal functions with names ar-1, as-1, and ap-1 (with the number of dimensions substituted for 1). Again, there is no point in using these functions yourself; it would only make your code harder to understand but not any faster at all. When there are more than three dimensions, the old Maclisp way is used: arrays are referenced by applying them as functions, using their dimensions as arguments, and they are stored into using xstore, which is like the Maclisp store but with its arguments in the other order. You can try compiling and disassembling some simple functions yourself if you want to see this in action.

When you call a function and expect to get more than one value back, a slightly different kind of function calling is used. Here is an example that uses multiple-value to get two values back from a function call:

```
(defun foo (x)
  (let (y z)
    (multiple-value (y z)
      (bar 3))
    (+ x y z)))
```

The disassembled code looks like this:

```
22 MOVE D-PDL FEF|6              ;#'BAR
23 MOVE D-PDL '2
24 (MISC) %CALL-MULT-VALUE D-IGNORE
25 MOVE D-LAST FEF|7             ;'3
26 POP LOCAL|1                   ;Z
27 POP LOCAL|0                   ;Y
30 MOVE D-PDL ARG|0              ;X
31 + LOCAL|0                     ;Y
32 + LOCAL|1                     ;Z
33 MOVE D-RETURN PDL-POP
```

A %CALL-MULT-VALUE instruction is used instead of a CALL instruction. The destination field of %CALL-MULT-VALUE is unused and will always be D-IGNORE. %CALL-MULT-VALUE takes two "arguments", which it finds on the stack; it pops both of them. The first one is the function object to be applied; the second is the number of return values that are expected. The rest of the call proceeds as usual, but when the call returns, the returned values are left on the stack. The number of objects left on the stack is always the same as the second "argument" to %CALL-MULT-VALUE. In our example, the two values returned are left on the stack, and they are immediately popped off into z and y. There is also a %CALL0-MULT-VALUE instruction, for the same reason CALL0 exists.

The multiple-value-bind form works similarly; here is an example:

```
(defun foo (x)
  (multiple-value-bind (y *foo* z)
      (bar 3)
    (+ x y z)))
```

The disassembled code looks like this:

```
24 MOVE D-PDL FEF|8           ;#'BAR
25 MOVE D-PDL FEF|7           ;'3
26 (MISC) %CALL-MULT-VALUE D-IGNORE
27 MOVE D-LAST FEF|7          ;'3
30 POP LOCAL|1                ;Z
31 BIND-POP FEF|6             ;*FOO*
32 POP LOCAL|0                ;Y
33 MOVE D-PDL ARG|0           ;X
34 + LOCAL|0                  ;Y
35 + LOCAL|1                  ;Z
36 MOVE D-RETURN PDL-POP
```

The %CALL-MULT-VALUE instruction is still used, leaving the results on the stack; these results are used to bind the variables.

Calls done with multiple-value-list work with the %CALL-MULT-VALUE-LIST instruction. It takes one "argument" on the stack: the function object to apply. When the function returns, the list of values is left on the top of the stack. Here is an example:

```
(defun foo (x y)
  (multiple-value-list (foo 3 y x)))
```

The disassembled code looks like this:

```
22 MOVE D-PDL FEF|6           ;#'FOO
23 (MISC) %CALL-MULT-VALUE-LIST D-IGNORE
24 MOVE D-PDL FEF|7           ;'3
25 MOVE D-PDL ARG|1           ;Y
26 MOVE D-LAST ARG|0          ;X
27 MOVE D-RETURN PDL-POP
```

Returning of more than one value from a function is handled by special miscellaneous instructions. %RETURN-2 and %RETURN-3 are used to return two or three values; these instructions take two and three arguments, respectively, on the stack, and return from the current function just as storing to D-RETURN would. If there are more than three return values, they are all pushed, then the number that there were is pushed, and then the %RETURN-N instruction is executed. None of these instructions use their destination field. Note: the **return-list** function is just an ordinary miscellaneous instruction; it takes the list of values to return as an argument on the stack, and it returns those values from the current function.

The function **lexpr-funcall** is compiled using a special instruction called %SPREAD to iterate over the elements of its last argument, which should be a list. %SPREAD takes one argument (on the stack), which is a list of values to be passed as arguments (pushed on the stack). If the destination of %SPREAD is D-PDL (or D-NEXT), then the values are just pushed; if it is D-LAST, then after the values are pushed, the function is invoked. **lexpr-funcall** will always compile using a %SPREAD whose destination is D-LAST. Here is an example:

```
(defun foo (a b &rest c)
  (lexpr-funcall #'format t a c)
  b)
```

The disassembled code looks like this:

```
20 CALL D-IGNORE FEF|6          ;#'FORMAT
21 MOVE D-PDL 'T
22 MOVE D-PDL ARG|0             ;A
23 MOVE D-PDL LOCAL|0           ;C
24 (MISC) %SPREAD D-LAST
25 MOVE D-RETURN ARG|1          ;B
```

Note that in instruction 23, the address LOCAL|0 is used to access the **&rest** argument.

The ***catch** special form is also handled specially by the compiler. Here is a simple example of ***catch**:

```
(defun a ()
  (*catch 'foo (bar)))
```

The disassembled code looks like this:

```
22 MOVE D-PDL FEF|6             ;'26
23 (MISC) %CATCH-OPEN D-PDL
24 MOVE D-PDL FEF|7             ;'FOO
25 CALL0 D-LAST FEF|8           ;#'BAR
26 MOVE D-RETURN PDL-POP
```

The %CATCH-OPEN instruction is like the CALL instruction; it starts a call to the ***catch** function. It takes one "argument" on the stack, which is the location in the program that should be branched to if this ***catch** is ***thrown** to. In addition to saving that program location, the instruction saves the state of the stack and of special-variable binding so that they can be restored in the event of a ***throw**. So instructions 22 and 23 start a ***catch** block, and the rest of the function passes its two arguments. The ***catch** function itself simply returns its second argument; but if a ***throw** happens during the evaluation of the (bar) form, then the stack will be unwound and execution will resume at instruction 26. The destination field of %CATCH-OPEN is like that of CALL; it is saved on the stack, and controls what will be done with the result of the call to the ***catch**. Note that even though ***catch** is really a Lisp special form, it is compiled more or less as if it were a function of two arguments.

To allow compilation of (multiple-value (...) (*catch ...)), there is a special instruction called %CATCH-OPEN-MULT-VALUE, which is a cross between %CATCH-OPEN and %CALL-MULT-VALUE. multiple-value-list with *catch is not supported.

27.6 Estimating Run Time

You may sometimes want to estimate the speed at which a function will execute by examination of the compiled code. This section gives some rough guidelines to the relative cost of various instructions; the actual speed may vary from these estimates by as much as a factor of two. Some of these speeds vary with time; they speed up as work is done to improve system efficiency and slow down sometimes when sweeping changes are made (for instance, when garbage collection was introduced it slowed down some operations even when garbage collection is not turned on.) However these changes are usually much less than a factor of two.

It is also important to realize that in many programs the execution time is determined by paging rather than by CPU run time. The cost of paging is unfortunately harder to estimate than run time, because it depends on dynamic program behavior and locality of data structure.

On a conventional computer such as the pdp-10, rough estimates of the run time of compiled code are fairly easy to make. It is a reasonable approximation to assume that all machine instructions take about the same amount of time to execute. When the compiler generates a call to a runtime support routine, the user can estimate the speed of that routine since it is implemented in the same instructions as the user's compiled program. Actual speeds can vary widely because of data dependencies; for example, when using the plus function the operation will be much slower if an argument is a bignum than if the arguments are all fixnums. However, in Maclisp most performance-critical functions use declarations to remove such data dependencies, because generic, data-dependent operations are so much slower than type-specific operations.

Things are different in the Lisp Machine. The instruction set we have just seen is a high-level instruction set. Rather than specifying each individual machine operation, the compiler calls for higher-level Lisp operations such as cdr or memq. This means that some instructions take many times longer to execute than others. Furthermore, in the Lisp machine we do not use data-type declarations. Instead the machine is designed so that all instructions can be *generic*; that is, they determine the types of their operands at run time. This means that there are data dependencies that can have major effects on the speed of execution of an instruction. For instance, the + instruction is quite fast if both operands turn out to be fixnums, but much slower if they are bignums.

The Lisp machine also has a large amount of microcode, both to implement certain Lisp functions and to assist with common operations such as function calling. It is not as easy for a user to read microcode and estimate its speed as with compiled code, although it is a much more readable microcode than on most computers.

In this section we give some estimates of the speed of various operations. There are also facilities for measuring the actual achieved speed of a program. These will not be documented here as they are currently being changed.

We will express all times in terms of the time to execute the simplest instruction, MOVE D-PDL ARG|0. This time is about two microseconds and will be called a "unit".

MOVE takes the same amount of time if the destination is D-IGNORE or D-NEXT, or if the address is a LOCAL or PDL-POP rather than an ARG. A MOVE that references a constant, via either the FEF base register or the CONSTANT base register, takes about two units. A MOVE that references a special variable by means of the FEF base register and an invisible pointer takes closer to three units.

Use of a complex destination (D-LAST, D-RETURN, or D-NEXT-LIST) takes extra time because of the extra work it has to do calling a function, returning from a function, or the bookkeeping associated with forming a list. These costs will be discussed a bit later.

The other Class I instructions take longer than MOVE. Each memory reference required by car/cdr operations costs about one unit. Note that cdr requires one memory cycle if the list is compactly cdr-coded and two cycles if it is not. The CALL instruction takes three units. The CALL0 instruction takes more, of course, since it actually calls the function.

The Class II (no destination) instructions vary. The MOVEM and POP operations take about one unit. (Of course they take more if FEF or CONSTANT addressing is used.) The arithmetic and logical operations and the predicates take two units when applied to fixnums, except for multiplication and division which take five. The SETE-1+ and SETE-1- instructions take two units, the same time as a push followed by a pop; i.e. (setq x (1+ x)) takes the same amount of time as (setq x y). The SET-NIL and SET-ZERO instructions take one unit. The special-variable binding instructions take several units.

A branch takes between one and two units.

The cost of calling a function with no arguments and no local variables that doesn't do anything but return nil is about 15 units (7 cdrs or additions). This is the cost of a CALL FEF|*n* instruction, a MOVE to D-LAST, the simplest form of function-entry services, and a MOVE to D-RETURN. If the function takes arguments the cost of calling the function includes the cost of the instructions in the caller that compute the arguments. If the function has local variables initialized to nil or optional arguments defaulted to nil there is a negligible additional cost. The cost of having an &rest argument is less than one additional unit. But if the function binds special variables there is an additional cost of 8 units per variable (this includes both binding the variables on entry and unbinding them on return).

If the function needs an ADL, typically because of complex optional-argument initializations, the cost goes up substantially. It's hard to characterize just how much it goes up by, since this depends on what you do. Also calling for multiple values is more expensive than simple calling.

We consider the cost of calling functions to be somewhat higher than it should be, and would like to improve it. But this might require incompatible system architecture changes and probably will not happen, at least not soon.

Flonum and bignum arithmetic are naturally slower than fixnum arithmetic. For instance, flonum addition takes 8 units more than fixnum addition, and addition of 60-bit bignums takes 15 units more. Note that these times include some garbage-collection overhead for the intermediate results which have to be created in memory. Fixnums and small flonums do not take up any memory and avoid this overhead. Thus small-flonum addition takes only about 2 units more than fixnum addition. This garbage-collection overhead is of the "extra-pdl-area" sort rather than the full Baker garbage collector sort; if you don't understand this don't worry about it for now.

Floating-point subtraction, multiplication, and division take just about the same time as floating-point addition. Floating-point execution times can be as many as 3 units longer depending on the arguments.

The run time of a Class IV (or miscellaneous) instruction depends on the instruction and its arguments. The simplest instructions, predicates such as atom and numberp, take 2 units. This is basically the overhead for doing a Class IV instruction. The cost of a more complex instruction can be estimated by looking at what it has to do. You can get a reasonable estimate by charging one unit per memory reference, car operation, or cdr-coded cdr operation. A non-cdr-coded cdr operation takes two units. For instance, (memq 'nil '(a b c)) takes 13 units, of which 4 are pushing the arguments on the stack, 2 are Class IV instruction overhead, 6 are accounted for by cars and cdrs, and 1 is "left over".

The cost of array accessing depends on the type of array and the number of dimensions. aref of a 1-dimensional non-indirect art-q array takes 6 units and aset takes 5 units, not counting pushing the arguments onto the stack. (These are the costs of the AR-1 and AS-1 instructions.) A 2-dimensional array takes 6 units more. aref of a numeric array takes the same amount of time as aref of an art-q array. aset takes 1 unit longer. aref of an art-float array takes 5 units longer than aref of an art-q array. aset takes 3 units longer.

The functions copy-array-contents and copy-array-portion optimize their array accessing to remove overhead from the inner loop. copy-array-contents of an art-q array has a startup overhead of 8 units, not including pushing the arguments, then costs just over 2 units per array element.

The cons function takes 7 units if garbage collection is turned off. (list a b c d) takes 24 units, which includes 4 units for getting the local variables a, b, c, and d.

28. Querying the User

The following functions provide a convenient and consistent interface for asking questions of the user. Questions are printed and the answers are read on the stream query-io, which normally is synonymous with terminal-io but can be rebound to another stream for special applications.

We will first describe two simple functions for yes-or-no questions, then the more general function on which all querying is built.

y-or-n-p &optional *message stream*

This is used for asking the user a question whose answer is either "yes" or "no". It types out *message* (if any), reads a one-character answer, echoes it as "Yes" or "No", and returns t if the answer is "yes" or nil if the answer is "no". The characters which mean "yes" are Y, T, space, and hand-up. The characters which mean "no" are N, rubout, and hand-down. If any other character is typed, the function will beep and demand a "Y or N" answer.

If the *message* argument is supplied, it will be printed on a fresh line (using the :fresh-line stream operation). Otherwise the caller is assumed to have printed the message already. If you want a question mark and/or a space at the end of the message, you must put it there yourself; y-or-n-p will not add it. *stream* defaults to the value of query-io.

y-or-n-p should only be used for questions which the user knows are coming. If the user is not going to be anticipating the question (e.g. if the question is "Do you really want to delete all of your files?" out of the blue) then y-or-n-p should not be used, because the user might type ahead a T, Y, N, space, or rubout, and therefore accidentally answer the question. In such cases, use yes-or-no-p.

yes-or-no-p &optional *message stream*

This is used for asking the user a question whose answer is either "Yes" or "No". It types out *message* (if any), beeps, and reads in a line from the keyboard. If the line is the string "Yes", it returns t. If the line is "No", it returns nil. (Case is ignored, as are leading and trailing spaces and tabs.) If the input line is anything else, yes-or-no-p beeps and demands a "yes" or "no" answer.

If the *message* argument is supplied, it will be printed on a fresh line (using the :fresh-line stream operation). Otherwise the caller is assumed to have printed the message already. If you want a question mark and/or a space at the end of the message, you must put it there yourself; yes-or-no-p will not add it. *stream* defaults to the value of query-io.

To allow the user to answer a yes-or-no question with a single character, use y-or-n-p. yes-or-no-p should be used for unanticipated or momentous questions; this is why it beeps and why it requires several keystrokes to answer it.

fquery *options format-string* &rest *format-args*

Asks a question, printed by (format query-io *format-string format-args...*), and returns the answer. fquery takes care of checking for valid answers, reprinting the question when the user clears the screen, giving help, and so forth.

options is a list of alternating keywords and values, used to select among a variety of features. Most callers will have a constant list which they pass as *options* (rather than consing up a list whose contents varies). The keywords allowed are:

:type	What type of answer is expected. The currently-defined types are :tyi (a single character) and :readline (a line terminated by a carriage return). :tyi is the default.
:choices	Defines the allowed answers. The allowed forms of choices are complicated and explained below. The default is the same set of choices as the y-or-n-p function (see above). Note that the :type and :choices options should be consistent with each other.
:list-choices	If t, the allowed choices are listed (in parentheses) after the question. The default is t; supplying nil causes the choices not to be listed unless the user tries to give an answer which is not one of the allowed choices.
:help-function	Specifies a function to be called if the user hits the HELP key. The default help-function simply lists the available choices. Specifying nil disables special treatment of HELP. Specifying a function of three arguments—the stream, the list of choices, and the type-function—allows smarter help processing. The type-function is the internal form of the :type option and can usually be ignored.
:condition	If non-nil, a condition to be signalled before asking the question. The handler of this condition may supply an answer, in which case the user is not asked. The details are given below. The default condition is :fquery.
:fresh-line	If t, query-io is advanced to a fresh line before asking the question. If nil, the question is printed wherever the cursor was left by previous typeout. The default is t.
:beep	If t, fquery beeps to attract the user's attention to the question. The default is nil, which means not to beep unless the user tries to give an answer which is not one of the allowed choices.
:clear-input	If t, fquery throws away type-ahead before reading the user's response to the question. Use this for unexpected questions. The default is nil, which means not to throw away typeahead unless the user tries to give an answer which is not one of the allowed choices. In that case, typeahead is discarded since the user probably wasn't expecting the question.
:select	If t and query-io is a visible window, that window is temporarily selected while the question is being asked. The default is nil.
:make-complete	If t and query-io is a typeout-window, the window is "made complete" after the question has been answered. This tells the system that the contents of the window are no longer useful. Refer to the window system

documentation for further explanation. The default is **t**.

The argument to the **:choices** option is a list each of whose elements is a *choice*. The cdr of a choice is a list of the user inputs which correspond to that choice. These should be characters for **:type :tyi** or strings for **:type :readline**. The car of a choice is either a symbol which **fquery** should return if the user answers with that choice, or a list whose first element is such a symbol and whose second element is the string to be echoed when the user selects the choice. In the former case nothing is echoed. In most cases **:type :readline** would use the first format, since the user's input has already been echoed, and **:type :tyi** would use the second format, since the input has not been echoed and furthermore is a single character, which would not be mnemonic to see on the display.

Perhaps this can be clarified by example. The **yes-or-no-p** function uses this list of choices:

```
((t "Yes") (nil "No"))
```

and the y-or-n-p function uses this list:

```
(((t "Yes.") #/y #/t #\sp #\hand-up)
 ((nil "No.") #/n #\rubout #\hand-down))
```

If a condition is specified (or allowed to default to **:fquery**), before asking the question **fquery** will signal the condition. (See section 26.1.1, page 440 for information about conditions.) The handler will receive four arguments: the condition name, the *options* argument to **fquery**, the *format-string* argument to **fquery**, and the list of *format-args* arguments to **fquery**. As usual with conditions, if the handler returns **nil** the operation proceeds as if there had been no handler. If the handler returns two values, **t** and *ans*, **fquery** will immediately return *ans*. No conventions have yet been defined for standard condition names for use with **fquery**.

If you want to use the formatted output functions instead of **format** to produce the promting message, write

```
(fquery options (format:outfmt exp-or-string exp-or-string ...))
```

format:outfmt puts the output into a list of a string, which makes **format** print it exactly as is. There is no need to supply additional arguments to the **fquery** unless it signals a condition. In that case the arguments might be passed so that the condition handler can see them. The condition handler will receive a list containing one string, the message, as its third argument instead of just a string. If this argument is passed along to **format**, all the right things happen.

29. Initializations

There are a number of programs and facilities in the Lisp Machine which require that "initialization routines" be run either when the facility is first loaded, or when the system is booted, or both. These initialization routines may set up data structures, start processes running, open network connections, and so on.

An initialization that needs to be done once, when a file is loaded, can be done simply by putting the Lisp forms to do it in that file; when the file is loaded the forms will be evaluated. However, some initializations need to be done each time the system is booted, and some initializations depend on several files having been loaded before they can work.

The system provides a consistent scheme for managing these initializations. Rather than having a magic function which runs when the system is started and knows everything that needs to be initialized, each thing that needs initialization contains its own initialization routine. The system keeps track of all the initializations through a set of functions and conventions, and executes all the initialization routines when necessary. The system also avoids re-executing initializations if a program file is loaded again after it has already been loaded and initialized.

There is something called an *initialization list*. This is an ordered list of *initializations*. Each initialization has a name, a form to be evaluated, and a flag saying whether the form has yet been evaluated or not. When the time comes, initializations are evaluated in the order that they were added to the list. The name is a string and lies in the **car** of an initialization; thus **assoc** may be used on initialization lists.

add-initialization *name form* &optional *list-of-keywords initialization-list-name*

Adds an initialization called *name* with the form *form* to the initialization list specified either by *initialization-list-name* or by keyword. If the initialization list already contains an initialization called *name*, change its form to *form*.

initialization-list-name, if specified, is a symbol that has as its value the initialization list. If it is unbound, it is initialized (!) to **nil**. If a keyword specifies an initialization list, *initialization-list-name* is ignored and should not be specified.

The keywords allowed in *list-of-keywords* are of two kinds. These specify what initialization list to use:

:cold	Use the standard cold-boot list (see below).
:warm	Use the standard warm-boot list (see below). This is the default.
:before-cold	Use the standard before-disk-save list (see below).
:once	Use the once-only list (see below).
:system	Use the system list (see below).
:login	Use the login list (see below).
:logout	Use the logout list (see below).

These specify when to evaluate *form*:

:normal	Only place the form on the list. Do not evaluate it until the time comes to do this kind of initialization. This is the default unless :system or :once is specified.
:now	Evaluate the form now as well as adding it to the list.
:first	Evaluate the form now if it is not flagged as having been evaluated before. This is the default if :system or :once is specified.
:redo	Do not evaluate the form now, but set the flag to nil even if the initialization is already in the list and flagged t.

Actually, the keywords are compared with string-equal and may be in any package. If both kinds of keywords are used, the list keyword should come *before* the when keyword in *list-of-keywords*; otherwise the list keyword may override the when keyword.

The add-initialization function keeps each list ordered so that initializations added first are at the front of the list. Therefore, by controlling the order of execution of the additions, explicit dependencies on order of initialization can be controlled. Typically, the order of additions is controlled by the loading order of files. The system list (see below) is the most critically ordered of the pre-defined lists.

delete-initialization *name* &optional *keywords initialization-list-name*
Removes the specified initialization from the specified initialization list. Keywords may be any of the list options allowed by add-initialization.

initializations *initialization-list-name* &optional *redo-flag flag-value*
Perform the initializations in the specified list. *redo-flag* controls whether initializations that have already been performed are re-performed; nil means no, non-nil is yes, and the default is nil. *flag-value* is the value to be bashed into the flag slot of an entry. If it is unspecified, it defaults to *t*, meaning that the system should remember that the initialization has been done. The reason that there is no convenient way for you to specify one of the specially-known-about lists is that you shouldn't be calling initializations on them.

reset-initializations *initialization-list-name*
Bashes the flag of all entries in the specified list to nil, thereby causing them to get rerun the next time the function initializations is called on the initialization list.

29.1 System Initialization Lists

The special initialization lists that are known about by the above functions allow you to have your subsystems initialized at various critical times without modifying any system code to know about your particular subsystems. This also allows only a subset of all possible subsystems to be loaded without necessitating either modifying system code (such as lisp-reinitialize) or such kludgy methods as using fboundp to check whether or not something is loaded.

The :once initialization list is used for initializations that need to be done only once when the subsystem is loaded and must never be done again. For example, there are some databases that need to be initialized the first time the subsystem is loaded, but should not be reinitialized every time a new version of the software is loaded into a currently running system. This list is for that purpose. The initializations function is never run over it; its "when" keyword defaults to :first and so the form is normally only evaluated at load-time, and only if it has not been evaluated before. The :once initialization list serves a similar purpose to the defvar special form (see page 19), which sets a variable only if it is unbound.

The :system initialization list is for things that need to be done before other initializations stand any chance of working. Initializing the process and window systems, the file system, and the ChaosNet NCP falls in this category. The initializations on this list are run every time the machine is cold or warm booted, as well as when the subsystem is loaded unless explicitly overridden by a :normal option in the keywords list. In general, the system list should not be touched by user subsystems, though there may be cases when it is necessary to do so.

The :cold initialization list is used for things which must be run once at cold-boot time. The initializations on this list are run after the ones on :system but before the ones on the :warm list. They are run only once, but are reset by disk-save thus giving the appearance of being run only at cold-boot time.

The :warm initialization list is used for things which must be run every time the machine is booted, including warm boots. The function that prints the greeting, for example, is on this list. Unlike the :cold list, the :warm list initializations are run regardless of their flags.

The :before-cold initialization list is a variant of the :cold list. These initializations are run before the world is saved out by disk-save. Thus they happen essentially at cold boot time, but only once when the world is saved, not each time it is started up.

The :login and :logout lists are run by the login and logout functions (see page 506) respectively. Note that disk-save calls logout. Also note that often people don't call logout; they often just cold-boot the machine.

User programs are free to create their own initialization lists to be run at their own times. Some system programs, such as the editor, have their own initialization list for their own purposes.

30. Dates and Times

The **time:** package contains a set of functions for manipulating dates and times: finding the current time, reading and printing dates and times, converting between formats, and other miscellany regarding peculiarities of the calendar system. It also includes functions for accessing the Lisp Machine's microsecond timer.

Times are represented in two different formats by the functions in the **time** package. One way is to represent a time by many numbers, indicating a year, a month, a date, an hour, a minute, and a second (plus, sometimes, a day of the week and timezone). The year is relative to 1900 (that is, if it is 1981, the *year* value would be **81**); however, the functions that take a year as an argument will accept either form. The month is 1 for January, 2 for February, etc. The date is 1 for the first day of a month. The hour is a number from 0 to 23. The minute and second are numbers from 0 to 59. Days of the week are fixnums, where 0 means Monday, 1 means Tuesday, and so on. A timezone is specified as the number of hours west of GMT; thus in Massachusetts the timezone is 5. Any adjustment for daylight savings time is separate from this.

This "decoded" format is convenient for printing out times into a readable notation, but it is inconvenient for programs to make sense of these numbers, and pass them around as arguments (since there are so many of them). So there is a second representation, called Universal Time, which measures a time as the number of seconds since January 1, 1900, at midnight GMT. This "encoded" format is easy to deal with inside programs, although it doesn't make much sense to look at (it looks like a huge integer). So both formats are provided; there are functions to convert between the two formats; and many functions exist in two forms, one for each format.

The Lisp Machine hardware includes a timer that counts once every microsecond. It is controlled by a crystal and so is fairly accurate. The absolute value of this timer doesn't mean anything useful, since it is initialized randomly; what you do with the timer is to read it at the beginning and end of an interval, and subtract the two values to get the length of the interval in microseconds. These relative times allow you to time intervals of up to an hour (32 bits) with microsecond accuracy.

The Lisp Machine keeps track of the time of day by maintaining a "timebase", using the microsecond clock to count off the seconds. When the machine first comes up, the timebase is initialized by querying hosts on the Chaos net to find out the current time.

There is a similar timer which counts in 60ths of a second rather than microseconds; it is useful for measuring intervals of a few seconds or minutes with less accuracy. Periodic housekeeping functions of the system are scheduled based on this timer.

30.1 Getting the Time

time:get-time

Get the current time, in decoded form. Return seconds, minutes, hours, date, month, year, day-of-the-week, and daylight-savings-time-p, with the same meanings as time:decode-universal-time (see page 497).

time:get-universal-time

Returns the current time, in Universal Time form.

30.1.1 Elapsed Time in 60ths of a Second

The following functions deal with a different kind of time. These are not calendrical date/times, but simply elapsed time in 60ths of a second. These times are used for many internal purposes where the idea is to measure a small interval accurately, not to depend on the time of day or day of month.

time

Returns a number which increases by 1 every 1/60 of a second, and wraps around roughly once a day. Use the time-lessp and time-difference functions to avoid getting in trouble due to the wrap-around. time is completely incompatible with the Maclisp function of the same name.

time-lessp *time1 time2*

t if *time1* is earlier than *time2*, compensating for wrap-around, otherwise nil.

time-difference *time1 time2*

Assuming *time1* is later than *time2*, returns the number of 60ths of a second difference between them, compensating for wrap-around.

30.1.2 Elapsed Time in Microseconds

time:microsecond-time

Return the value of the microsecond timer, as a bignum.

time:fixnum-microsecond-time

Return the value of the low 23 bits of the microsecond timer, as a fixnum. This is like time:microsecond-time, with the advantage that it returns a value in the same format as the time function, except in microseconds rather than 60ths of a second. This means that you can compare fixnum-microsecond-times with time-lessp and time-difference. time:fixnum-microsecond-time is also a bit faster, but has the disadvantage that since you only see the low bits of the clock, the value can "wrap around" more quickly (every few seconds). Note that the Lisp Machine garbage collector is so designed that the bignums produced by time:microsecond-time are garbage-collected quickly and efficiently, so the overhead for creating the bignums is really not high.

30.2 Printing Dates and Times

The functions in this section create printed representations of times and dates in various formats, and send the characters to a stream. To any of these functions, you may pass **nil** as the *stream* parameter, and the function will return a string containing the printed representation of the time, instead of printing the characters to any stream.

time:print-current-time &optional (*stream* standard-output)
Print the current time, formatted as in 11/25/80 14:50:02, to the specified stream.

time:print-time *seconds minutes hours date month year* &optional (*stream* standard-output)
Print the specified time, formatted as in 11/25/80 14:50:02, to the specified stream.

time:print-universal-time *universal-time* &optional (*stream* standard-output) (*timezone* time:*timezone*)
Print the specified time, formatted as in 11/25/80 14:50:02, to the specified stream.

time:print-current-date &optional (*stream* standard-output)
Print the current time, formatted as in Tuesday the twenty-fifth of November, 1980; 3:50:41 pm, to the specified stream.

time:print-date *seconds minutes hours date month year day-of-the-week* &optional (*stream* standard-output)
Print the specified time, formatted as in Tuesday the twenty-fifth of November, 1980; 3:50:41 pm, to the specified stream.

time:print-universal-date *universal-time* &optional (*stream* standard-output) (*timezone* time:*timezone*)
Print the specified time, formatted as in Tuesday the twenty-fifth of November, 1980; 3:50:41 pm, to the specified stream.

time:print-brief-universal-time *universal-time* &optional (*stream* standard-output) *reference-time*
This is like time:print-universal-time except that it omits seconds and only prints those parts of *universal-time* that differ from *reference-time*, a universal time that defaults to the current time. Thus the output will be in one of the following three forms:

```
02:59           ;the same day
3/4 14:01       ;a different day in the same year
8/17/74 15:30   ;a different year
```

30.3 Reading Dates and Times

These functions will accept most reasonable printed representations of date and time and convert them to the standard internal forms. The following are representative formats that are accepted by the parser.

```
"March 15, 1960" "15 March 1960" "3//15//60"
"15//3//60" "3//15//1960" "3-15-60" "15-3-1960"
"3-15" "15-March-60" "15-Mar-60" "March-15-60"
"1130." "11:30" "11:30:17" "11:30 pm" "11:30 AM" "1130" "113000"
"11.30" "11.30.00" "11.3" "11 pm" "12 noon"
"midnight" "m" "Friday, March 15, 1980" "6:00 gmt" "3:00 pdt"
"15 March 60" "15 march 60 seconds"
"Fifteen March 60" "The Fifteenth of March, 1960;"
"One minute after March 3, 1960"
"Two days after March 3, 1960"
"Three minutes after 23:59:59 Dec 31, 1959"
"Now" "Today" "Yesterday" "two days after tomorrow"
"one day before yesterday" "the day after tomorrow"
"five days ago"
```

time:parse *string* &optional (*start* 0) (*end* nil) (*futurep* t) *base-time must-have-time date-must-have-year time-must-have-second* (*day-must-be-valid* t)

Interpret *string* as a date and/or time, and return seconds, minutes, hours, date, month, year, day-of-the-week, daylight-savings-time-p, and relative-p. *start* and *end* delimit a substring of the string; if *end* is nil, the end of the string is used. *must-have-time* means that *string* must not be empty. *date-must-have-year* means that a year must be explicitly specified. *time-must-have-second* means that the second must be specified. *day-must-be-valid* means that if a day of the week is given, then it must actually be the day that corresponds to the date. *base-time* provides the defaults for unspecified components; if it is nil, the current time is used. *futurep* means that the time should be interpreted as being in the future; for example, if the base time is 5:00 and the string refers to the time 3:00, that means the next day if *futurep* is non-nil, but it means two hours ago if *futurep* is nil. The *relative-p* returned value is t if the string included a relative part, such as "one minute after" or "two days before" or "tomorrow" or "now"; otherwise, it is nil. If the parse encounters an error, the first returned value is a string giving an error message.

time:parse-universal-time *string* &optional (*start* 0) (*end* nil) (*futurep* t) *base-time must-have-time date-must-have-year time-must-have-second* (*day-must-be-valid* t)

This is the same as time:parse except that it returns one integer, representing the time in Universal Time, and the *relative-p* value. If the parse encounters an error, the first returned value is a string giving an error message.

30.4 Time Conversions

time:decode-universal-time *universal-time* &optional (*timezone* time:*timezone*)

Convert *universal-time* into its decoded representation. The following values are returned: seconds, minutes, hours, date, month, year, day-of-the-week, daylight-savings-time-p. *daylight-savings-time-p* tells you whether or not daylight savings time is in effect; if so, the value of *hour* has been adjusted accordingly. You can specify *timezone* explicitly if you want to know the equivalent representation for this time in other parts of the world.

time:encode-universal-time *seconds minutes hours date month year* &optional *timezone*

Convert the decoded time into Universal Time format, and return the Universal Time as an integer. If you don't specify *timezone*, it defaults to the current timezone adjusted for daylight savings time; if you provide it explicitly, it is not adjusted for daylight savings time. *year* may be absolute, or relative to 1900 (that is, 81 and 1981 both work).

time:*timezone* *Variable*

The value of time:*timezone* is the time zone in which this Lisp Machine resides, expressed in terms of the number of hours west of GMT this time zone is. This value does not change to reflect daylight savings time; it tells you about standard time in your part of the world.

30.5 Internal Functions

These functions provide support for those listed above. Some user programs may need to call them directly, so they are documented here.

time:initialize-timebase

Initialize the timebase by querying Chaos net hosts to find out the current time. This is called automatically during system initialization. You may want to call it yourself to correct the time if it appears to be inaccurate or downright wrong.

time:daylight-savings-time-p *hours date month year*

Return t if daylight savings time is in effect for the specified hour; otherwise, return nil. *year* may be absolute, or relative to 1900 (that is, 81 and 1981 both work).

time:daylight-savings-p

Return t if daylight savings time is currently in effect; otherwise, return nil.

time:month-length *month year*

Return the number of days in the specified *month*; you must supply a *year* in case the month is February (which has a different length during leap years). *year* may be absolute, or relative to 1900 (that is, 81 and 1981 both work).

time:leap-year-p *year*

Return t if *year* is a leap year; otherwise return nil. *year* may be absolute, or relative to 1900 (that is, 81 and 1981 both work).

time:verify-date *date month year day-of-the-week*

If the day of the week of the date specified by *date*, *month*, and *year* is the same as *day-of-the-week*, return nil; otherwise, return a string which contains a suitable error message. *year* may be absolute, or relative to 1900 (that is, 81 and 1981 both work).

time:day-of-the-week-string *day-of-the-week* &optional (*mode* ':long)

Return a string representing the day of the week. As usual, 0 means Monday, 1 means Tuesday, and so on. Possible values of *mode* are:

:long	Return the full English name, such as "Monday", "Tuesday", etc. This is the default.
:short	Return a three-letter abbreviation, such as "Mon", "Tue", etc.
:medium	Same as :short, but use "Tues" and "Thurs".
:french	Return the French name, such as "Lundi", "Mardi", etc.
:german	Return the German name, such as "Montag", "Dienstag", etc.

time:month-string *month* &optional (*mode* ':long)

Return a string representing the month of the year. As usual, 1 means January, 2 means February, etc. Possible values of *mode* are:

:long	Return the full English name, such as "January", "February", etc. This is the default.
:short	Return a three-letter abbreviation, such as "Jan", "Feb", etc.
:medium	Same as :short, but use "Sept", "Novem", and "Decem".
:roman	Return the Roman numeral for *month* (this convention is used in Europe).
:french	Return the French name, such as "Janvier", "Fevrier", etc.
:german	Return the German name, such as "Januar", "Februar", etc.

time:timezone-string &optional (*timezone* time:*timezone*)
(*daylight-savings-p* (time:daylight-savings-p))

Return the three-letter abbreviation for this time zone. For example, if *timezone* is 5, then either "EST" (Eastern Standard Time) or "CDT" (Central Daylight Time) will be used, depending on *daylight-savings-p*.

31. Miscellaneous Useful Functions

This chapter describes a number of functions which don't logically fit in anywhere else. Most of these functions are not normally used in programs, but are "commands", i.e. things that you type directly at Lisp.

31.1 Poking Around in the Lisp World

who-calls *x* &optional *package*
who-uses *x* &optional *package*

x must be a symbol or a list of symbols. who-calls tries to find all of the functions in the Lisp world which call *x* as a function, use *x* as a variable, or use *x* as a constant. (It won't find things that use constants which contain *x*, such as a list one of whose elements is *x*; it will only find it if *x* itself is used as a constant.) It tries to find all of the functions by searching all of the function cells of all of the symbols on *package* and *package*'s descendants. *package* defaults to the global package, and so normally all packages are checked.

If who-calls encounters an interpreted function definition, it simply tells you if *x* appears anywhere in the interpreted code. who-calls is smarter about compiled code, since it has been nicely predigested by the compiler.

If *x* is a list of symbols, who-calls does them all simultaneously, which is faster than doing them one at a time.

who-uses is an obsolete name for who-calls.

The editor has a command, META/X List Callers, which is similar to who-calls.

The symbol unbound-function is treated specially by who-calls. (who-calls 'unbound-function) will search the compiled code for any calls through a symbol which is not currently defined as a function. This is useful for finding errors such as functions you misspelled the names of or forgot to write.

who-calls prints one line of information for each caller it finds. It also returns a list of the names of all the callers.

what-files-call *x* &optional *package*

Similar to who-calls but returns a list of the pathnames of all the files which contain functions that who-calls would have printed out. This is useful if you need to recompile and/or edit all of those files.

apropos *string* &optional *package*

(apropos *string*) tries to find all symbols whose print-names contain *string* as a substring. Whenever it finds a symbol, it prints out the symbol's name; if the symbol is defined as a function and/or bound to a value, it tells you so, and prints the names of the arguments (if any) to the function. It finds the symbols on *package* and *package*'s decendants. *package* defaults to the global package, so normally all packages are searched. apropos returns a list of all the symbols it finds.

where-is *pname* &optional *package*

Prints the names of all packages which contain a symbol with the print-name *pname*. If *pname* is a string it gets upper-cased. The package *package* and all its sub-packages are searched; *package* defaults to the global package, which causes all packages to be searched. where-is returns a list of all the symbols it finds.

describe *x*

describe tries to tell you all of the interesting information about any object *x* (except for array contents). describe knows about arrays, symbols, flonums, packages, stack groups, closures, and FEFs, and prints out the attributes of each in human-readable form. Sometimes it will describe something which it finds inside something else; such recursive descriptions are indented appropriately. For instance, describe of a symbol will tell you about the symbol's value, its definition, and each of its properties. describe of a flonum (regular or small) will show you its internal representation in a way which is useful for tracking down roundoff errors and the like.

If *x* is a named-structure, describe handles it specially. To understand this, you should read the section on named structures (see page 271). First it gets the named-structure symbol, and sees whether its function knows about the :describe operation. If the operation is known, it applies the function to two arguments: the symbol :describe, and the named-structure itself. Otherwise, it looks on the named-structure symbol for information which might have been left by defstruct; this information would tell it what the symbolic names for the entries in the structure are, and describe knows how to use the names to print out what each field's name and contents is.

describe always returns its argument, in case you want to do something else to it.

inspect *x*

A window-oriented version of describe. See the window system documentation for details, or try it.

disassemble *function*

function should be a FEF, or a symbol which is defined as a FEF. This prints out a human-readable version of the macro-instructions in *function*. The macro-code instruction set is explained in chapter 27, page 469.

The grindef function (see page 360) may be used to display the definition of a non-compiled function.

room &rest *areas*

room tells you the amount of physical memory on the machine, the amount of available virtual memory not yet filled with data (that is, the portion of the available virtual memory that has not yet been allocated to any region of any area), and the amount of "wired" physical memory (i.e. memory not available for paging). Then it tells you how much room is left in some areas. For each area it tells you about, it prints out the name of the area, the number of regions which currently make up the area, the current size of the area in kilowords, and the amount of the area which has been allocated, also in kilowords. If the area cannot grow, the percentage which is free is displayed.

(room) tells you about those areas which are in the list which is the value of the variable room. These are the most interesting ones.

(room *area1 area2*...) tells you about those areas, which can be either the names or the numbers.

(room t) tells you about all the areas.

(room nil) does not tell you about any areas; it only prints the header. This is useful if you just want to know how much memory is on the machine or how much virtual memory is available.

room *Variable*

The value of room is a list of area names and/or area numbers, denoting the areas which the function room will describe if given no arguments. Its initial value is:

```
        (working-storage-area macro-compiled-program)
```

set-memory-size *n-words*

set-memory-size tells the virtual memory system to use only *n-words* words of main memory for paging. Of course, *n-words* may not exceed the amount of main memory on the machine.

31.2 Utility Programs

ed &optional *x*

ed is the main function for getting into the editor, Zwei. Zwei is not yet documented in this manual, but the commands are very similar to Emacs.

(ed) or (ed nil) simply enters the editor, leaving you in the same buffer as the last time you were in the editor.

(ed t) puts you in a fresh buffer with a generated name (like BUFFER-4).

(ed *pathname*) edits that file. *pathname* may be an actual pathname or a string.

(ed 'foo) tries hard to edit the definition of the foo function. It will find a buffer or file containing the source code for foo and position the cursor at the beginning of the code. In general, foo can be any function-spec (see section 10.2, page 136).

(ed 'zwei:reload) reinitializes the editor. It will forget about all existing buffers, so use this only as a last resort.

zwei:save-all-files

This function is useful in emergencies in which you have modified material in Zmacs buffers that needs to be saved, but the editor is partially broken. This function does what the editor's Save All Files command does, but it stays away from redisplay and other advanced facilities so that it might work if other things are broken.

dired &optional *pathname*

Puts up a window and edits the directory named by *pathname*, which defaults to the last file opened. While editing a directory you may view, edit, compare, hardcopy, and delete the files it contains. While in the directory editor type the HELP key for further information.

mail &optional *who what*

Sends mail by putting up a window in which you may compose the mail. *who* is a symbol or a string which is who to send it to. *what* is a string which is the initial contents of the mail. If these are unspecified they can be typed in during composition of the mail. Type the END key to send the mail and return from the mail function.

bug &optional *who what*

Reports a bug. *who* is the name of the faulty program (a symbol or a string). It defaults to lispm (the Lisp Machine system itself). *what* is a string which is the initial contents of the mail. bug is like mail but includes information about the system version and what machine you are on in the text of the message. This information is important to the maintainers of the faulty program; it aids them in reproducing the bug and in determining whether it is one which is already being worked on or has already been fixed.

qsend *who* &optional *what*

Sends a message to another user. qsend is different from mail because it sends the message immediately; it will appear within seconds on the other user's screen, rather than being saved in her mail file.

who is a string of the form "*user@host*"; *host* is the name of the Lisp Machine or timesharing system the user is currently logged-in to. *what* is a string which is the message. If *what* is not specified, you will be prompted to type in a message. Unlike mail and bug, qsend does not put up a window to allow you to edit the message; it just sends it.

[qsend currently does not evaluate its arguments, and is implemented as a macro, but this should probably be changed.]

print-sends

Reprints any messages that have been received. This is useful if you want to see a message again.

print-notifications

Reprints any notifications that have been received. The difference between notifications and sends is that sends come from other users, while notifications are asynchronous messages from the Lisp Machine system itself.

si:print-disk-error-log
Prints information about the half dozen most recent disk errors (since the last cold boot).

peek &optional *character*
peek is similar to the ITS program of the same name. It displays various information about the system, periodically updating it. Like ITS PEEK, it has several modes, which are entered by typing a single key which is the name of the mode. The initial mode is selected by the argument, *character*. If no argument is given, peek starts out by explaining what its modes are.

hostat &rest *hosts*
Asks each of the *hosts* for its status, and prints the results. If no hosts are specified, all hosts on the Chaosnet are asked. Hosts can be specified either by name or by number.

For each host, a line is output which either says that the host is not responding or gives metering information for the host's network attachments. If a host is not responding, that usually means that it is down or there is no such host at that address. A Lisp Machine can fail to respond if it is looping inside without-interrupts or paging extremely heavily, such that it is simply unable to respond within a reasonable amount of time.

supdup &optional *host*
host may be a string or symbol, which will be taken as a host name, or a number, which will be taken as a host number. If no *host* is given, the machine you are logged-in to is assumed. This function opens a connection to the host over the Chaosnet using the Supdup protocol, and allows the Lisp Machine to be used as a terminal for any ITS or TOPS-20 system.

To give commands to supdup, type the NETWORK key followed by one character. Type NETWORK followed by HELP for documentation.

telnet &optional *host simulate-imlac*
telnet is similar to supdup but uses the Arpanet-standard Telnet protocol, simulating a printing terminal rather than a display terminal.

31.3 The Lisp Top Level

These functions constitute the Lisp top level, and its associated functions.

si:lisp-top-level
This is the first function called in the initial Lisp environment. It calls lisp-reinitialize, clears the screen, and calls si:lisp-top-level1.

lisp-reinitialize
This function does a wide variety of things, such as resetting the values of various global constants and initializing the error system.

si:lisp-top-level1

This is the actual top level loop. It reads a form from standard-input, evaluates it, prints the result (with slashification) to standard-output, and repeats indefinitely. If several values are returned by the form all of them will be printed. Also the values of *, +, -, //, ++, **, +++, and *** are maintained (see below).

break *tag* [*conditional-form*] *Special Form*

break is used to enter a breakpoint loop, which is similar to a Lisp top level loop. (break *tag*) will always enter the loop; (break *tag conditional-form*) will evaluate *conditional-form* and only enter the break loop if it returns non-nil. If the break loop is entered, break prints out

```
        ;Breakpoint tag; Resume to continue, Abort to quit.
```

and then enters a loop reading, evaluating, and printing forms. A difference between a break loop and the top level loop is that when reading a form, break checks for the following special cases: If the Abort key is typed, control is returned to the previous break or error-handler, or to top-level if there is none. If the Resume key is typed, break returns nil. If the symbol ◊p is typed, break returns nil. If the list (return *form*) is typed, break evaluates *form* and returns the result.

Inside the break loop, the streams standard-output, standard-input, and query-io are bound to be synonymous to terminal-io; terminal-io itself is not rebound. Several other internal system variables are bound, and you can add your own symbols to be bound by pushing elements onto the value of the variable sys:*break-bindings* (see page 505).

prin1 *Variable*

The value of this variable is normally nil. If it is non-nil, then the read-eval-print loop will use its value instead of the definition of prin1 to print the values returned by functions. This hook lets you control how things are printed by all read-eval-print loops—the Lisp top level, the break function, and any utility programs that include a read-eval-print loop. It does not affect output from programs that call the prin1 function or any of its relatives such as print and format; if you want to do that, read about customizing the printer, on section 21.2.1, page 321. If you set prin1 to a new function, remember that the read-eval-print loop expects the function to print the value but not to output a return character or any other delimiters.

- *Variable*

While a form is being evaluated by a read-eval-print loop, - is bound to the form itself.

+ *Variable*

While a form is being evaluated by a read-eval-print loop, + is bound to the previous form that was read by the loop.

***** *Variable*

While a form is being evaluated by a read-eval-print loop, * is bound to the result printed the last time through the loop. If there were several values printed (because of a multiple-value return), * is bound to the first value.

// *Variable*

While a form is being evaluated by a read-eval-print loop, // is bound to a list of the results printed the last time through the loop.

++ *Variable*

+ + holds the previous value of +, that is, the form evaluated two interactions ago.

+++ *Variable*

+ + + holds the previous value of + +.

** *Variable*

** holds the previous value of *, that is, the result of the form evaluated two interactions ago.

*** *Variable*

*** holds the previous value of **.

sys:*break-bindings* *Variable*

When **break** is called, it binds some special variables under control of the list which is the value of sys:*break-bindings*. Each element of the list is a list of two elements: a variable and a form which is evaluated to produce the value to bind it to. The bindings happen sequentially. Users may **push** things on this list (adding to the front of it), but should not replace the list wholesale since several of the variable bindings on this list are essential to the operation of **break**.

lisp-crash-list *Variable*

The value of lisp-crash-list is a list of forms. lisp-reinitialize sequentially evaluates these forms, and then sets lisp-crash-list to nil.

In most cases, the *initialization* facility should be used rather than lisp-crash-list. Refer to chapter 29, page 490. .

31.4 The Garbage Collector

gc-on

Turns garbage collection on. It is off by default, currently.

gc-off

Turns garbage collection off.

number-gc-on &optional (*on-p* t)

Turns the special bignum/flonum garbage collector on, or off if *on-p* is nil. This garbage collector is on by default, since it has negligible overhead and significantly improves the performance of computational programs.

31.5 Logging In

Logging in tells the Lisp Machine who you are, so that other users can see who is logged in, you can receive messages, and your INIT file can be run. An INIT file is a Lisp program which gets loaded when you log in; it can be used to set up a personalized environment.

When you log out, it should be possible to undo any personalizations you have made so that they do not affect the next user of the machine. Therefore, anything done by an INIT file should be undoable. In order to do this, for every form in the INIT file, a Lisp form to undo its effects should be added to the list which is the value of logout-list. The functions login-setq and login-eval help make this easy; see below.

user-id *Variable*

The value of user-id is either the name of the logged in user, as a string, or else an empty string if there is no user logged in. It appears in the who-line.

logout-list *Variable*

The value of logout-list is a list of forms which are evaluated when a user logs out.

login *name* &optional *host load-init-file*

Sets your name (the variable user-id) to *name* and logs in a file server on *host*. *host* also becomes your default file host. If *host* requires passwords for logging in you will be asked for a password. When logging in to a TOPS-20 host, typing an asterisk before your password will enable any special capabilities you may be authorized to use. The default value of *host* depends on which Lisp Machine you use using. It is found from the value of chaos:machine-location-alist, which is a list that has one element for every known individual Lisp Machine. login also runs the :login initialization list (see page 492).

Unless *load-init-file* is specified as nil, login will load your init file if it exists. On ITS, your init file is *name* LISPM on your home directory. On TOPS-20 your init file is LISPM.INIT on your directory.

If anyone is logged into the machine already, login logs him out before logging in *name*. (See logout.) Init files should be written using the login-setq and login-eval functions below so that logout can undo them. Usually, however, you cold-boot the machine before logging in, to remove any traces of the previous user. login returns t.

logout

First, logout evaluates the forms on logout-list. Then it sets user-id to an empty string and logout-list to nil. Then it runs the :logout initialization list (see page 492), and returns t.

login-setq {*variable value*}... *Special Form*

login-setq is like setq except that it puts a setq form on logout-list to set the variables to their previous values.

login-eval *x*

login-eval is used for functions which are "meant to be called" from INIT files, such as zwei:set-comtab-return-undo, which conveniently return a form to undo what they did. login-eval adds the result of the form *x* to the logout-list.

31.6 Dribble Files

dribble-start *filename* &optional *editor-p*

dribble-start opens *filename* as a "dribble file" (also known as a "wallpaper file"). It rebinds standard-input and standard-output so that all of the terminal interaction is directed to the file as well as the terminal. If *editor-p* is non-nil, then instead of opening *filename* on the file computer, dribble-start dribbles into a Zmacs buffer whose name is *filename*, creating it if it doesn't exist.

dribble-end

This closes the file opened by dribble-start and resets the I/O streams.

31.7 Status and SStatus

The status and sstatus special forms exist for compatibility with Maclisp. Programs that wish to run in both Maclisp and Zetalisp can use status to determine which of these they are running in. Also, (sstatus feature ...) can be used as it is in Maclisp.

status *Special Form*

(status features) returns a list of symbols indicating features of the Lisp environment. The complete list of all symbols which may appear on this list, and their meanings, is given in the Maclisp manual. The default list for the Lisp Machine is:

```
(loop defstruct site sort fasload string
 newio roman trace grindef grind lispm)
```

The value of this list will be kept up to date as features are added or removed from the Lisp Machine system. Most important is the symbol lispm, which is the last element of the list; this indicates that the program is executing on the Lisp Machine. *site* is a symbol indicating where the machine is located, such as :mit or :xerox. The order of this list should not be depended on, and may not be the same as shown above.

This features list is used by the *# +* read-time conditionalization syntax. See page 327.

(status feature *symbol*) returns t if *symbol* is on the (status features) list, otherwise nil.

(status nofeature *symbol*) returns t if *symbol* is not on the (status features) list, otherwise nil.

(status status) returns a list of all status operations.

(status sstatus) returns a list of all sstatus operations.

sstatus *Special Form*

(sstatus feature *symbol*) adds *symbol* to the list of features.

(sstatus nofeature *symbol*) removes *symbol* from the list of features.

Concept Index

"&" keywords, 148

reader macros, 325

%%kbd fields, 315

60ths of a second, 494

accessor function, 257
active process, 428
advice to functions, 460
alist, 65
allocation of storage, 81
alphabetic case, 127
alterant macro, 269
applicable function, 9
area, 54, 192
argument checking, 446
array, 9, 107
array initialization, 111
array leader, 110, 113
array-elements loop iteration path, 251
arrest reason, 428
association list, 65, 69
atom, 4, 9
attribute, 71

backquote, 211
band, 422
base-flavor, 304
basic definition, 137, 153
binding, 7
blocks, 72
byte, 102
byte specifiers, 102

catch, 47
cell, 170
character object, 333
character set, 314
circular list, 52
cleanup handler, 48
closure, 10, 158
combined-method, 291
compiled code, 469
compiler, 197
compiler warnings, 203
completion (of file names), 374
condition handler, 441
conditional, 33
conditionalization, read-time, 327
conditionalizing clause(s), in loop, 243
conditions, 440
cons, 9, 52
constructor macro, 258, 267
control structure, 33
coroutine, 158, 163

data type, 8
data type keywords, in loop, 245
dates and times, 493
debugger, 450
declaration, 201
declaring packages, 395
default (pathname), 380
definition, 7, 136
defmacro, 210
defstruct, 257
dependency, 409
device (pathname), 376
directory (of files), 373
directory (pathname), 376
disembodied property list, 72
displaced array, 110, 112
displacing macros, 226
dot, in symbols, 218
dotted list, 52, 324
dotted pair, 4

elements (of a list), 52
encapsulation, 137, 153
entity, 10, 162
eq versus equal, 11, 12
equal versus =, 12
error system, 440
eval-when, 199
evalhook, 24
evaluation, 13
exit, 33
external value cell, 159

fasdump, 206
FEF, 7, 144
file, 365
file directory, 373
file property list, 369
filename, 376
filename completion, 374
fill pointer, 110
fish, 35
flavor, 279
flavor-method-symbol, 294
flow of control, 33
formatted output, 346

formatting lisp code, 360
function, 9, 136
function cell, 87
function entry frame, 144
function renaming, 156
function spec, 136

generic pathname, 381
grinding, 360
grouped array, 271

handling conditions, 441
handling errors, 440
hash table, 74, 78
home directory, 384
host (pathname), 376

I/O stream, 338
ignored arguments, 204
index offset, 111, 112
indicator, 71
indirect array, 111, 112
init file, 384
initialization, 490
input and output, 314
input to the compiler, 198
instance, 279
internal value cell, 159
interned-symbols loop iteration path, 250
invisible-pointer, 174
iteration, 33, 38, 233
ITS pathnames, 386

keyboard character, 315
kitty, 37
kitty, yu-shiang, 71, 350

lambda list, 20
lambda-list keywords, 148
lexpr, 28
list, 52
loading, 368
local variable, 14
locative, 10, 170
lock, 432
logical pathnames, 388
loop, 233
lower case letter, 127

machine code, 469
Maclisp file manipulation, 390
macro character, 325
macro defining macros, 257
macro-defining macros, 210
macros, 208
mapping, 50
MAR, 467
merge (pathname), 377, 380
message, 279
method, 279
microseconds, 494
mixin, 304
module, 408
monitoring the value of a variable, 468
multiple accumulations, in loop, 241
multiple values, 29
multiprocessing, 428

name (pathname), 376
named structure, 271
named-structure, 320
named-subst, 143
namelists (Maclisp compatibility), 390
naming convention, 8
nil, used as a condition name, 444
non-local exit, 33, 47
number, 9, 92

object, 279
optimizer, compiler, 204
order of evaluation in iteration clauses, in loop, 236

package, 392
package declarations, 395
parse (pathname), 377
partition, 422
patch, 416
pathname, 376
period, in symbols, 218
plane, 122
plist, 71
ppss, 102
predicate, 8
pretty-printing, 360
print name, 7, 89
printer, 319
process, 428
process wait function, 428
program source file, 368
property list, 7, 71
property list, file, 369

querying the user, 487
quote, 25

reader, 322
readtable, 328
record (structure), 257
recursion, 33
rename-within, 156
resource, 81
resumer, 164
returning multiple values, 29
rubout handler, 361

run reason, 428

S-expression, 4
saving the Lisp world, 422
scheduler, 429
sequential vs parallel binding and initialization, in loop, 235
set, 65
setf, 229
sharp sign reader macros, 325
signaller, 441
signalling conditions, 441
signalling errors, 444
simple process, 438
single-character symbol, 330
slashification, 319
slot, 259
sorting, 79
special character, 328
special forms, 141
special variable, 14, 200
stack group, 109, 163
status, of a patchable system, 421
stepping through evaluation, 464
storage allocation, 81
stream, 338
string, 9, 126
structured pathname components, 379
structures, 257
subprimitive, 172
subscript, 107
subst, 143, 215
substitutable function, 215
substitution, 62
supplied-p variable, 22
symbol, 7, 8, 86
system, 406
system maintenance, 416

Tenex pathnames, 387
terminating the iteration, in loop, 242
throw, 47
times and dates, 493
TOPS-20 pathnames, 387
tracing function execution, 456
transformation, 409
tree, 52
type (pathname), 376
types of arrays, 107

universal time, 493
unspecific pathname components, 378
unwind protection, 48
unwinding a stack, 48
upper case letter, 127

value cell, 86
variable bindings, in loop, 237
version (pathname), 376

wait, 428
warnings from the compiler, 203

yes-or-no, 487

Flavor Index

si:process, 438
si:property-list-mixin, 312
si:simple-process, 438
si:vanilla-flavor, 305

Message Index

:active-p (to process), 437
:advance-input-buffer, 373
:arrest-reason (to process), 437
:arrest-reasons (to process), 437

:back-translated-pathname (to fs:logical-pathname), 389
:beep, 341

:clear-input, 340
:clear-output, 341
:clear-screen, 342
:close, 341
:creation-date, 372

:delete, 372
:describe, 305
:device (to pathname), 384
:directory (to pathname), 384

:eval-inside-yourself, 306

:finish, 341
:flush (to process), 438
:fn1 (to its-pathname), 387
:fn2 (to its-pathname), 387
:force-output, 341
:fresh-line, 339
:funcall-inside-yourself, 306

:generic-pathname (to pathname), 385
:get (to pathname), 386
:get (to si:property-list-mixin), 313
:get-handler-for, 306
:get-input-buffer, 373
:getl (to pathname), 386
:getl (to si:property-list-mixin), 313

:host (to pathname), 384

:info, 372
:initial-form (to process), 435
:initial-stack-group (to process), 435
:interrupt (to process), 438

:kill (to process), 438

:length, 372
:line-in, 340
:line-out, 340
:listen, 339

:name (to pathname), 384
:name (to process), 435
:new-device (to pathname), 385
:new-directory (to pathname), 385
:new-name (to pathname), 385
:new-pathname (to pathname), 385
:new-structured-device (to pathname), 385
:new-structured-directory (to pathname), 385
:new-structured-name (to pathname), 385
:new-type (to pathname), 385
:new-version (to pathname), 385

:operation-handled-p, 306

:pathname, 371
:plist (to pathname), 386
:preset (to process), 437
:print-self, 305
:priority (to process), 436
:property-list (to si:property-list-mixin), 313
:push-property (to si:property-list-mixin), 313
:putprop (to pathname), 386
:putprop (to si:property-list-mixin), 313

:qfaslp, 372
:quantum (to process), 436
:quantum-remaining (to process), 436

:read-cursorpos, 342
:read-pointer, 372
:remprop (to pathname), 386
:remprop (to si:property-list-mixin), 313
:rename, 372
:reset (to process), 437
:revoke-arrest-reason (to process), 437
:revoke-run-reason (to process), 436
:rewind, 373
:rubout-handler, 341
:run-reason (to process), 436
:run-reasons (to process), 436
:runnable-p (to process), 437

:send-if-handles, 306
:set-byte-size, 372
:set-cursorpos, 342
:set-pointer, 372
:set-priority (to process), 436
:set-property-list (to si:property-list-mixin), 313
:set-quantum (to process), 436
:set-warm-boot-action (to process), 436
:simple-p (to process), 436
:stack-group (to process), 435
:string-for-dired (to pathname), 385

:string-for-editor (to pathname), 385
:string-for-host (to pathname), 386
:string-for-printing (to pathname), 385
:string-for-wholine (to pathname), 385
:string-out, 340

:translated-pathname (to fs:logical-pathname), 389
:truename, 372
:tyi, 338
:tyi-no-hang, 342
:tyo, 338
:type (to pathname), 384

:untyi, 339
:untyo, 342
:untyo-mark, 342

:version (to pathname), 384

:wait-argument-list (to process), 435
:wait-function (to process), 435
:warm-boot-action (to process), 436
:which-operations, 306, 339
:whostate (to process), 436

Keyword Index

sys:%%region-map-bits (for make-area), 194
sys:%%region-scavenge-enable (for make-area), 194
sys:%%region-space-type (for make-area), 194

:accessor-prefix (for defflavor), 303
:alterant (for defstruct), 261
:area (for make-array), 111
:area (for make-equal-hash-table), 77
:area (for make-hash-table), 75
:area (for make-list), 57
:arg (for trace), 458
:argpdl (for trace), 458
:array (for defstruct), 261
:array-leader (for defstruct), 261
:arrest-reasons (for make-process), 434
:ascii (for open), 366

:batch (for make-system), 412
:beep (for fquery), 488
:before-cold (for add-initialization), 490
:block (for open), 366
:both (for trace), 458
:break (for trace), 457
:but-first (for defstruct), 265
:byte-size (for open), 366

:callable-accessors (for defstruct), 266
:chaos (for si:sb-on), 432
:checker (for defresource), 83
:choices (for fquery), 488
:clear-input (for fquery), 488
:clock (for si:sb-on), 432
:cold (for add-initialization), 490
:combination (for "defflavor), 304
:compile (for make-system), 412
:component-systems (for defsystem), 407
:conc-name (for defstruct), 262
:cond (for trace), 457
:condition (for fquery), 488
:cons (for defstruct-define-type), 276
:constructor (for defresource), 82
:constructor (for defstruct), 261

:default-handler (for defflavor), 302
:default-init-plist (for defflavor), 300
:default-pointer (for defstruct), 262
:default-value (for make-plane), 123
:defstruct (for defstruct-define-type), 277
:deleted (for fs:complete-pathname), 375
:deleted (for fs:directory-list), 374
:deleted (for open), 366
:displace (for defstruct), 265

:displaced-index-offset (for make-array), 112
:displaced-to (for make-array), 112
:documentation (for defflavor), 304

:entry (for trace), 458
:entrycond (for trace), 457
:entryprint (for trace), 458
:error (for trace), 457
:essential-mixin (for "defflavor), 304
:eval-when (for defstruct), 266
:exit (for trace), 458
:exitbreak (for trace), 457
:exitcond (for trace), 457
:exitprint (for trace), 458
:extension (for make-plane), 123

:finder (for defresource), 83
:first (for add-initialization), 491
:fixnum (for defstruct), 261
:fixnum (for open), 366
:fixnum-array (for defstruct), 261
:flavor (for make-process), 433
:flonum-array (for defstruct), 261
:french (for time:day-of-the-week-string), 498
:french (for time:month-string), 498
:fresh-line (for fquery), 488

:gc (for make-area), 194
:german (for time:day-of-the-week-string), 498
:german (for time:month-string), 498
:gettable-instance-variables (for defflavor), 300
:grouped-array (for defstruct), 261
:growth-factor (for make-equal-hash-table), 77

:help-function (for fquery), 488

:in (for fs:complete-pathname), 375
:in (for open), 366
:include (for defstruct), 262
:included-flavors (for defflavor), 302
:increment-patch (for make-system), 412
:init-keywords (for defflavor), 300
:initable-instance-variables (for defflavor), 300
:initial-copies (for defresource), 83
:initial-offset (for defstruct), 265
:initial-status (for defsystem), 408
:initial-value (for make-list), 57

:keyboard (for si:sb-on), 432
:keywords (for defstruct-define-type), 277

:leader-length (for make-array), 112

:leader-list (for make-array), 112
:list (for defstruct), 261
:list-choices (for fquery), 488
:login (for add-initialization), 490
:logout (for add-initialization), 490
:long (for time:day-of-the-week-string), 498
:long (for time:month-string), 498
:lowlevel-mixin (for "defflavor), 304

:macro (for setsyntax), 330
:make-array (for defstruct), 264
:make-complete (for fquery), 488
:matcher (for defresource), 83
:medium (for time:day-of-the-week-string), 498
:medium (for time:month-string), 498
:method-combination (for defflavor), 304
:mixin (for "defflavor), 304
:module (for defsystem), 408

:name (for defsystem), 407
:name (for make-area), 193
:named (for defstruct), 264
:named (for defstruct-define-type), 277
:named-array (for defstruct), 261
:named-array-leader (for defstruct), 261
:named-list (for defstruct), 261
:named-structure-symbol (for make-array), 112
:new-ok (for fs:complete-pathname), 375
:no-increment-patch (for make-system), 412
:no-vanilla-flavor (for defflavor), 302
:noconfirm (for make-system), 412
:noerror (for fs:directory-list), 373
:noerror (for open), 366
:noload (for make-system), 412
:noop (for make-system), 412
:normal (for add-initialization), 491
:noselective (for load-patches), 420
:not-in-disk-label (for defsystem), 408
:now (for add-initialization), 491

:old (for fs:complete-pathname), 375
:once (for add-initialization), 490
:ordered-instance-variables (for defflavor), 302
:out (for fs:complete-pathname), 375
:out (for open), 366
:outside-accessible-instance-variables (for defflavor), 303
:overhead (for defstruct-define-type), 277

:package (for defsystem), 407
:patchable (for defsystem), 408
:pathname-default (for defsystem), 407
:pdl (for make-area), 194
:print (for fs:complete-pathname), 375
:print (for open), 366
:print (for trace), 458
:print-only (for make-system), 412
:priority (for make-process), 433

:probe (for open), 366
:property (for defstruct), 266

:quantum (for make-process), 433

:raw (for open), 366
:read (for fs:complete-pathname), 375
:read (for open), 366
:read-only (for make-area), 194
:redo (for add-initialization), 491
:ref (for defstruct-define-type), 276
:region-size (for make-area), 193
:regular-pdl-area (for make-process), 434
:regular-pdl-area (for make-stack-group), 166
:regular-pdl-size (for make-process), 434
:regular-pdl-size (for make-stack-group), 166
:rehash-function (for make-hash-table), 75
:rehash-size (for make-hash-table), 76
:rehash-threshold (for make-equal-hash-table), 77
:reload (for make-system), 412
:representation (for make-area), 193
:required-flavors (for defflavor), 301
:required-instance-variables (for defflavor), 300
:required-methods (for defflavor), 301
:roman (for time:month-string), 498
:room (for make-area), 194
:run-reasons (for make-process), 433

:safe (for make-stack-group), 167
:select (for fquery), 488
:select-method-order (for defflavor), 303
:selective (for load-patches), 420
:selective (for make-system), 412
:settable-instance-variables (for defflavor), 300
:sg-area (for make-process), 434
:sg-area (for make-stack-group), 166
:short (for time:day-of-the-week-string), 498
:short (for time:month-string), 498
:short-name (for defsystem), 407
:silent (for load-patches), 420
:silent (for make-system), 412
:simple-p (for make-process), 433
:single (for open), 366
:single (for setsyntax), 331
:size (for make-area), 193
:size (for make-equal-hash-table), 76
:size (for make-hash-table), 75
:size-macro (for defstruct), 265
:size-symbol (for defstruct), 264
:special-pdl-area (for make-process), 434
:special-pdl-area (for make-stack-group), 166
:special-pdl-size (for make-process), 434
:special-pdl-size (for make-stack-group), 166
:special-purpose (for "defflavor), 304
:splicing (for setsyntax), 331
:stack-group (for make-process), 433
:step (for trace), 457

:super-image (for open), 366
:swap-sv-of-sg-that-calls-me (for make-process), 434
:swap-sv-of-sg-that-calls-me (for make-stack-group), 166
:swap-sv-on-call-out (for make-process), 434
:swap-sv-on-call-out (for make-stack-group), 166
:system (for add-initialization), 490
:systems (for load-patches), 420

:temporary (for open), 366
:times (for defstruct), 264
:trap-enable (for make-process), 434
:trap-enable (for make-stack-group), 166
:tree (for defstruct), 261
:type (for defstruct), 260
:type (for fquery), 488
:type (for make-array), 111
:type (for make-plane), 123

:value (for trace), 458
:verbose (for load-patches), 420

:warm (for add-initialization), 490
:warm-boot-action (for make-process), 433
:wherein (for trace), 458
:write (for fs:complete-pathname), 375
:write (for open), 366

Object Creation Options

:property-list (for si:property-list-mixin), 313

Meter Index

sys:%aging-depth, 191
sys:%count-age-flushed-pages, 191
sys:%count-aged-pages, 191
sys:%count-chaos-transmit-aborts, 189
sys:%count-cons-work, 189
sys:%count-disk-ecc-corrected-errors, 190
sys:%count-disk-errors, 190
sys:%count-disk-page-read-operations, 190
sys:%count-disk-page-reads, 189
sys:%count-disk-page-write-busys, 190
sys:%count-disk-page-write-operations, 190
sys:%count-disk-page-write-waits, 190
sys:%count-disk-page-writes, 189
sys:%count-disk-prepages-not-used, 190
sys:%count-disk-prepages-used, 190
sys:%count-disk-read-compare-differences, 190
sys:%count-disk-read-compare-rereads, 190
sys:%count-disk-read-compare-rewrites, 191
sys:%count-disk-recalibrates, 190
sys:%count-findcore-emergencies, 191
sys:%count-findcore-steps, 191
sys:%count-first-level-map-reloads, 189
sys:%count-fresh-pages, 189
sys:%count-meta-bits-map-reloads, 189
sys:%count-pdl-buffer-memory-faults, 189
sys:%count-pdl-buffer-read-faults, 189
sys:%count-pdl-buffer-write-faults, 189
sys:%count-scavenger-work, 189
sys:%count-second-level-map-reloads, 189
sys:%disk-error-log-pointer, 191
sys:%disk-wait-time, 190
sys:%tv-clock-rate, 189

Variable Index

%%arg-desc-max-args, 152
%%arg-desc-min-args, 152
%%ch-char, 315
%%ch-font, 315
%%kbd-char, 315
%%kbd-control, 315
%%kbd-control-meta, 316
%%kbd-hyper, 315
%%kbd-meta, 315
%%kbd-mouse, 316
%%kbd-mouse-button, 316
%%kbd-mouse-n-clicks, 316
%%kbd-super, 315
%%q-all-but-cdr-code, 182
%%q-all-but-pointer, 182
%%q-all-but-typed-pointer, 182
%%q-cdr-code, 181
%%q-data-type, 181
%%q-flag-bit, 181
%%q-high-half, 182
%%q-low-half, 182
%%q-pointer, 182
%%q-pointer-within-page, 182
%%q-typed-pointer, 182
%arg-desc-evaled-rest, 152
%arg-desc-fef-bind-hair, 152
%arg-desc-fef-quote-hair, 152
%arg-desc-interpreted, 152
%arg-desc-quoted-rest, 152
sys:%chaos-csr-address, 187
sys:%current-stack-group, 165
sys:%current-stack-group-calling-args-number, 186
sys:%current-stack-group-calling-args-pointer, 186
sys:%current-stack-group-previous-stack-group, 165
sys:%current-stack-group-state, 186
sys:%disk-blocks-per-cylinder, 188
sys:%disk-blocks-per-track, 188
sys:%disk-run-light, 187
sys:%error-handler-stack-group, 186
sys:%gc-flip-ready, 187
sys:%gc-generation-number, 187
sys:%inhibit-read-only, 187
sys:%initial-fef, 186
sys:%initial-stack-group, 186
sys:%loaded-band, 188
%mar-high, 187
%mar-low, 187
sys:%meter-buffer-pointer, 188
sys:%meter-disk-address, 188
sys:%meter-disk-count, 188
sys:%meter-global-enable, 188
%microcode-version-number, 186
sys:%number-of-micro-entries, 186
sys:%page-cons-alarm, 187
sys:%read-compare-enables, 188
sys:%region-cons-alarm, 187
sys:%scheduler-stack-group, 187
sys:%trap-micro-pc, 186

*, 504
**, 505
***, 505
all-flavor-names, 293
si:*batch-mode-p*, 413
sys:*break-bindings*, 505
fs:*default-pathname-defaults*, 381
fs:*defaults-are-per-host*, 381
si:*file-transformation-function*, 413
si:*flavor-compilations*, 299
fs:*its-uninteresting-types*, 387
fs:*known-types*, 382
si:*make-system-forms-to-be-evaled-after*, 413
si:*make-system-forms-to-be-evaled-before*, 413
si:*make-system-forms-to-be-evaled-finally*, 413
*nopoint, 319
fs:*pathname-hash-table*, 384
si:*query-type*, 413
si:*redo-all*, 413
si:*silent-p*, 413
si:*system-being-defined*, 414
si:*system-being-made*, 412
time:*timezone*, 497
si:*top-level-transformations*, 413

+, 504
++, 505
+++, 505

-, 504

//, 505

sys:a-memory-counter-block-names, 191
sys:a-memory-location-names, 188
sys:active-processes, 431
si:advised-functions, 462
sys:all-processes, 432
all-special-switch, 203
allow-variables-in-function-position-switch, 202
alphabetic-case-affects-string-comparison, 127
area-list, 194
arglist, 459
array-bits-per-element, 109
array-elements-per-q, 109

array-types, 109
art-16b, 107
art-1b, 107
art-2b, 107
art-32b, 108
art-4b, 107
art-8b, 107
art-fat-string, 108
art-float, 108
art-fps-float, 108
art-half-fix, 108
art-q, 107
art-q-list, 107
art-reg-pdl, 109
art-special-pdl, 109
art-stack-group-head, 109
art-string, 108

base, 319

cdr-error, 182
cdr-next, 182
cdr-nil, 182
cdr-normal, 182
sys:clock-function-list, 431
current-process, 430
sys:currently-prepared-sheet, 188

default-cons-area, 193

eh:error-handler-io, 343
si:encapsulation-standard-order, 155
eh:error-handler-io, 343
error-output, 343
errset, 450
evalhook, 466

sys:fasl-constants-area, 196
sys:fdefine-file-pathname, 149
fs:*default-pathname-defaults*, 381
fs:*defaults-are-per-host*, 381
fs:*its-uninteresting-types*, 387
fs:*known-types*, 382
fs:*pathname-hash-table*, 384
fs:last-file-opened, 381
fs:load-pathname-defaults, 381

ibase, 322
ignore, 204
inhibit-fdefine-warnings, 149
inhibit-scavenging-flag, 187
inhibit-scheduling-flag, 430
inhibit-style-warnings-switch, 203
sys:init-list-area, 196
si:initial-process, 432
si:initial-readtable, 329

lambda-list-keywords, 148
fs:last-file-opened, 381
lisp-crash-list, 505
fs:load-pathname-defaults, 381
si:login-history, 425
logout-list, 506
si:loop-use-system-destructuring?, 247

macro-compiled-program, 195

sys:nr-sym, 195
sys:number-cons-area, 186

obsolete-function-warning-switch, 202
open-code-map-switch, 202

sys:p-n-string, 195
package, 398
permanent-storage-area, 195
sys:pkg-area, 195
prin1, 504
prinlength, 321
prinlevel, 321
sys:property-list-area, 195

q-data-types, 174
query-io, 343

read-preserve-delimiters, 332
readtable, 328
room, 501
rubout-handler, 362
run-in-maclisp-switch, 202

sys:scheduler-stack-group, 431
self, 297
si:*batch-mode-p*, 413
si:*file-transformation-function*, 413
si:*flavor-compilations*, 299
si:*make-system-forms-to-be-evaled-after*, 413
si:*make-system-forms-to-be-evaled-before*, 413
si:*make-system-forms-to-be-evaled-finally*, 413
si:*query-type*, 413
si:*redo-all*, 413
si:*silent-p*, 413
si:*system-being-defined*, 414
si:*system-being-made*, 412
si:*top-level-transformations*, 413
si:advised-functions, 462
si:encapsulation-standard-order, 155
si:initial-process, 432
si:initial-readtable, 329
si:login-history, 425
si:loop-use-system-destructuring?, 247
standard-input, 343
standard-output, 343
sys:%chaos-csr-address, 187

sys:%current-stack-group, 165
sys:%current-stack-group-calling-args-number, 186
sys:%current-stack-group-calling-args-pointer, 186
sys:%current-stack-group-previous-stack-group, 165
sys:%current-stack-group-state, 186
sys:%disk-blocks-per-cylinder, 188
sys:%disk-blocks-per-track, 188
sys:%disk-run-light, 187
sys:%error-handler-stack-group, 186
sys:%gc-flip-ready, 187
sys:%gc-generation-number, 187
sys:%inhibit-read-only, 187
sys:%initial-fef, 186
sys:%initial-stack-group, 186
sys:%loaded-band, 188
sys:%meter-buffer-pointer, 188
sys:%meter-disk-address, 188
sys:%meter-disk-count, 188
sys:%meter-global-enable, 188
sys:%number-of-micro-entries, 186
sys:%page-cons-alarm, 187
sys:%read-compare-enables, 188
sys:%region-cons-alarm, 187
sys:%scheduler-stack-group, 187
sys:%trap-micro-pc, 186
sys:*break-bindings*, 505
sys:a-memory-counter-block-names, 191
sys:a-memory-location-names, 188
sys:active-processes, 431
sys:all-processes, 432
sys:clock-function-list, 431
sys:currently-prepared-sheet, 188
sys:fasl-constants-area, 196
sys:fdefine-file-pathname, 149
sys:init-list-area, 196
sys:nr-sym, 195
sys:number-cons-area, 186
sys:p-n-string, 195
sys:pkg-area, 195
sys:property-list-area, 195
sys:scheduler-stack-group, 431

terminal-io, 343
time:*timezone*, 497
trace-compile-flag, 460
trace-output, 343

user-id, 506

values, 459

working-storage-area, 195

Function Index

≠, 96
≤, 96
≥, 95

%24-bit-difference, 105
%24-bit-plus, 105
%24-bit-times, 105
%activate-open-call-block, 183
%allocate-and-initialize, 177
%allocate-and-initialize-array, 177
%area-number, 195
%args-info, 152
%assure-pdl-room, 183
sys:%binding-instances, 185
sys:%change-page-status, 184
sys:%compute-page-hash, 184
sys:%create-physical-page, 184
%data-type, 176
sys:%delete-physical-page, 184
sys:%disk-restore, 185
sys:%disk-save, 185
%divide-double, 106
%find-structure-header, 176
%find-structure-leader, 177
%float-double, 106
sys:%halt, 178
sys:%internal-value-cell, 185
%logdpb, 104
%logldb, 104
%make-pointer, 176
%make-pointer-offset, 176
%multiply-fractions, 105
%open-call-block, 182
%p-cdr-code, 180
%p-contents-as-locative, 179
%p-contents-as-locative-offset, 179
%p-contents-offset, 179
%p-data-type, 180
%p-deposit-field, 180
%p-deposit-field-offset, 180
%p-dpb, 180
%p-dpb-offset, 180
%p-flag-bit, 181
%p-ldb, 179
%p-ldb-offset, 180
%p-mask-field, 180
%p-mask-field-offset, 180
%p-pointer, 180
%p-store-cdr-code, 181
%p-store-contents, 179
%p-store-contents-offset, 179
%p-store-data-type, 181
%p-store-flag-bit, 181
%p-store-pointer, 181
%p-store-tag-and-pointer, 179
%pointer, 176
%pointer-difference, 176
%pop, 183
%push, 183
%region-number, 195
%remainder-double, 106
%stack-frame-pointer, 181
%store-conditional, 178
%string-equal, 128
%string-search-char, 131
%structure-boxed-size, 177
%structure-total-size, 177
%unibus-read, 178
%unibus-write, 178
sys:%using-binding-instances, 185
%xbus-read, 178
%xbus-write, 178
sys:%xbus-write-sync, 178

*, 97
*$, 97
*array, 124
*catch, 47
*dif, 98
*expr, 202
*fexpr, 202
*lexpr, 202
*plus, 98
*quo, 98
*throw, 48
*times, 98
*unwind-stack, 48

+, 96
+$, 96

-, 96
-$, 96

//, 97
//$, 97

1+, 98
1+$, 98
1-, 98
1-$, 98

<, 95
<=, 96

=, 95

>, 95
>=, 95

@define, 146

abs, 97
add-initialization, 490
fs:add-logical-pathname-host, 388
compiler:add-optimizer, 205
add1, 98
adjust-array-size, 116
advise, 460
si:advise-1, 461
advise-within, 463
allocate-resource, 84
aloc, 114
alphalessp, 135
and, 35
ap-1, 114
ap-2, 114
ap-3, 114
ap-leader, 114
append, 59
apply, 24
apropos, 499
ar-1, 113
ar-2, 113
ar-3, 113
area-name, 195
aref, 113
arg, 29
arglist, 150
args-info, 151
array, 124
array-#-dims, 115
array-active-length, 115
array-bits-per-element, 109
array-dimension-n, 115
array-dimensions, 115
array-displaced-p, 116
array-element-size, 109
array-elements-per-q, 109
array-grow, 117
array-has-leader-p, 116
array-in-bounds-p, 116
array-indexed-p, 116
array-indirect-p, 116
array-leader, 114
array-leader-length, 116
array-length, 114
array-pop, 118
array-push, 118
array-push-extend, 118
array-type, 114
array-types, 109
arraycall, 125
arraydims, 115
arrayp, 9
as-1, 113
as-2, 113
as-3, 113
ascii, 135
aset, 113
ash, 101
ass, 70
assoc, 70
assq, 69
atan, 99
atan2, 99
atom, 9

beep, 335
bigp, 9
bind, 183
bit-test, 101
bitblt, 120
boole, 100
boundp, 86
break, 504
format:breakline, 359
bug, 502
butlast, 60

c...r, 53
caaaar, 53
caaadr, 53
caaar, 53
caadar, 53
caaddr, 53
caadr, 53
caar, 53
cadaar, 53
cadadr, 53
cadar, 53
caddar, 53
cadddr, 53
caddr, 53
cadr, 53
call, 25
car, 53
car-location, 54
caseq, 38
catch, 48
catch-all, 49
catch-error, 450
cdaaar, 53
cdaadr, 53
cdaar, 53
cdadar, 53
cdaddr, 53
cdadr, 53
cdar, 53

cddaar, 53
cddadr, 53
cddar, 53
cdddar, 53
cddddr, 53
cdddr, 53
cddr, 53
cdr, 53
cerror, 445
fs:change-file-properties, 374
char-downcase, 127
char-equal, 127
char-lessp, 127
char-upcase, 127
character, 127
check-arg, 446
check-arg-type, 448
circular-list, 57
clear-mar, 467
close, 367
fs:close-all-files, 367
closure, 161
closure-alist, 161
closure-function, 161
closurep, 10
clrhash, 76
clrhash-equal, 77
comment, 27
si:compare-band, 426
compile, 197
compile-flavor-methods, 298
compiler-let, 203
compiler:add-optimizer, 205
compiler:fasd-file-symbols-properties, 207
compiler:fasd-font, 207
compiler:fasd-symbol-value, 207
compiler:function-referenced, 204
compiler:make-obsolete, 204
fs:complete-pathname, 375
cond, 34
cond-every, 34
condition-bind, 441
cons, 53
cons-in-area, 54
copy-array-contents, 119
copy-array-contents-and-leader, 120
copy-array-portion, 120
copy-readtable, 329
copyalist, 58
copylist, 58
copylist*, 58
copysymbol, 90
copytree, 58
cos, 99
cosd, 99
cursorpos, 335
data-type, 173
time:day-of-the-week-string, 498
time:daylight-savings-p, 497
time:daylight-savings-time-p, 497
deallocate-resource, 84
debugging-info, 150
decf, 231
declare, 200
declare-flavor-instance-variables, 297
time:decode-universal-time, 497
math:decompose, 122
def, 146
fs:default-pathname, 383
defconst, 19
deff, 146
defflavor, 292
si:define-defsystem-special-variable, 414
define-loop-macro, 245
define-loop-path, 252
define-loop-sequence-path, 251
si:define-make-system-special-variable, 413
si:define-simple-transformation, 414
defmacro, 210
defmacro-displace, 227
defmethod, 293
defprop, 73
defresource, 82
defselect, 147
defstruct, 259
defstruct-define-type, 275
defsubst, 215
defsystem, 406
defun, 139
defun-compatibility, 146
defunp, 140
defvar, 19
defwrapper, 295
del, 68
del-if, 69
del-if-not, 69
delete, 68
delete-initialization, 491
deletef, 367
delq, 67
deposit-byte, 103
deposit-field, 103
describe, 500
describe-area, 194
describe-defstruct, 260
describe-flavor, 299
describe-package, 396
fs:describe-pathname, 384
math:determinant, 121
difference, 96
fs:directory-list, 373
dired, 502
disassemble, 500

disk-restore, 423
disk-save, 424
dispatch, 37
displace, 226
do, 38
do-named, 41
documentation, 150
dolist, 42
dotimes, 42
dpb, 103
dribble-end, 507
dribble-start, 507

ed, 501
si:edit-disk-label, 423
eh, 451
si:encapsulate, 153
time:encode-universal-time, 497
entity, 162
entityp, 10
eq, 11
equal, 12
err, 450
error, 445
error-restart, 446
errset, 449
eval, 23
eval-when, 200
evalhook, 466
evenp, 94
every, 69
exp, 99
explode, 336
explodec, 336
exploden, 336
expt, 98

false, 27
compiler:fasd-file-symbols-properties, 207
compiler:fasd-font, 207
compiler:fasd-symbol-value, 207
fasload, 368
fboundp, 87
fdefine, 149
fdefinedp, 149
fdefinition, 150
si:fdefinition-location, 150
si:fdefinition-symbol-or-location, 150
ferror, 444
fifth, 55
fs:file-properties, 374
fs:file-property-bindings, 371
fs:file-read-property-list, 370
math:fill-2d-array, 122
fillarray, 119
find-position-in-list, 67
find-position-in-list-equal, 67
first, 55
firstn, 61
fix, 99
time:fixnum-microsecond-time, 494
fixnump, 9
fixp, 9
fixr, 99
flatc, 337
flatsize, 337
flavor-allows-init-keyword-p, 299
float, 100
floatp, 9
flonump, 9
fmakunbound, 87
follow-cell-forwarding, 175
follow-structure-forwarding, 175
format, 346
format:breakline, 359
format:ochar, 357
format:ofloat, 357
format:onum, 357
format:oprint, 357
format:ostring, 357
format:outfmt, 356
format:output, 356
format:pad, 358
format:plural, 359
format:print-list, 355
format:tab, 358
forward-value-cell, 175
fourth, 55
fquery, 488
freturn, 48
fs:add-logical-pathname-host, 388
fs:change-file-properties, 374
fs:close-all-files, 367
fs:complete-pathname, 375
fs:default-pathname, 383
fs:describe-pathname, 384
fs:directory-list, 373
fs:file-properties, 374
fs:file-property-bindings, 371
fs:file-read-property-list, 370
fs:init-file-pathname, 384
fs:make-pathname, 383
fs:make-pathname-defaults, 383
fs:merge-and-set-pathname-defaults, 383
fs:merge-pathname-defaults, 382
fs:parse-pathname, 382
fs:pathname-plist, 384
fs:set-default-pathname, 383
fs:user-homedir, 384
fset, 87
fset-carefully, 149
fsymeval, 87
funcall, 24
funcall-self, 297

function, 26
function-cell-location, 88
compiler:function-referenced, 204
functionp, 9

g-l-p, 117
gc-off, 505
gc-on, 505
gcd, 98
gensym, 91
get, 72
get-handler-for, 298
get-list-pointer-into-array, 118
get-locative-pointer-into-array, 118
get-pname, 89
si:get-system-version, 418
time:get-time, 494
time:get-universal-time, 494
getchar, 135
getcharn, 135
gethash, 76
gethash-equal, 77
getl, 72
globalize, 404
go, 45
greaterp, 95
grind-top-level, 360
grindef, 360

haipart, 102
haulong, 102
hostat, 503

if, 33
if-for-lispm, 205
if-for-maclisp, 206
if-for-maclisp-else-lispm, 206
if-in-lispm, 206
if-in-maclisp, 206
ignore, 27
implode, 135
incf, 231
inhibit-style-warnings, 203
fs:init-file-pathname, 384
initializations, 491
time:initialize-timebase, 497
inspect, 500
instantiate-flavor, 294
intern, 399
intern-local, 399
intern-local-soft, 399
intern-soft, 399
math:invert-matrix, 121
isqrt, 98

keyword-extract, 42

last, 56
ldb, 102
ldb-test, 103
ldiff, 61
time:leap-year-p, 497
length, 54
lessp, 95
let, 16
let*, 17
let-closed, 161
let-globally, 17
let-if, 17
lexpr-funcall, 24
lexpr-funcall-self, 297
lisp-reinitialize, 503
si:lisp-top-level, 503
si:lisp-top-level1, 504
list, 56
list*, 57
list*-in-area, 57
math:list-2d-array, 122
list-array-leader, 119
list-in-area, 57
listarray, 119
listify, 29
listp, 9
load, 368
load-byte, 103
si:load-mcr-file, 425
load-patches, 420
local-declare, 201
locate-in-closure, 161
locate-in-instance, 299
locativep, 10
locf, 230
log, 99
logand, 100
login, 506
login-eval, 507
login-setq, 506
logior, 100
lognot, 100
logout, 506
logxor, 100
loop, 233
loop-finish, 242
si:loop-named-variable, 254
si:loop-tassoc, 253
si:loop-tequal, 253
si:loop-tmember, 253
lsh, 101
lsubrcall, 25

macro, 208
macroexpand, 229
macroexpand-1, 228
mail, 502

make-area, 193
make-array, 111
make-array-into-named-structure, 273
make-broadcast-stream, 344
make-equal-hash-table, 76
make-hash-table, 75
make-instance, 294
make-list, 57
compiler:make-obsolete, 204
fs:make-pathname, 383
fs:make-pathname-defaults, 383
make-plane, 123
make-process, 433
make-stack-group, 166
make-symbol, 90
make-syn-stream, 344
make-system, 411
maknam, 135
makunbound, 86
map, 50
mapatoms, 400
mapatoms-all, 400
mapc, 50
mapcan, 50
mapcar, 50
mapcon, 50
maphash, 76
maphash-equal, 77
maplist, 50
mar-mode, 467
mask-field, 103
math:decompose, 122
math:determinant, 121
math:fill-2d-array, 122
math:invert-matrix, 121
math:list-2d-array, 122
math:multiply-matrices, 121
math:solve, 122
math:transpose-matrix, 121
max, 96
mem, 67
memass, 70
member, 66
memq, 66
fs:merge-and-set-pathname-defaults, 383
fs:merge-pathname-defaults, 382
mexp, 226
time:microsecond-time, 494
min, 96
minus, 96
minusp, 94
monitor-variable, 468
time:month-length, 497
time:month-string, 498
multiple-value, 30
multiple-value-bind, 30
multiple-value-list, 31
math:multiply-matrices, 121

named-structure-invoke, 273
named-structure-p, 273
named-structure-symbol, 273
nbutlast, 60
nconc, 59
ncons, 54
ncons-in-area, 54
neq, 11
nleft, 61
nlistp, 9
not, 12
nreconc, 60
nreverse, 58
nsublis, 63
nsubst, 62
nsubstring, 129
nsymbolp, 8
nth, 55
nthcdr, 56
null, 12
number-gc-on, 505
numberp, 9

format:ochar, 357
oddp, 94
format:ofloat, 357
once-only, 223
format:onum, 357
open, 367
format:oprint, 357
or, 35
format:ostring, 357
format:outfmt, 356
format:output, 356

package-cell-location, 400
package-declare, 396
format:pad, 358
sys:page-in-area, 184
sys:page-in-array, 184
sys:page-in-region, 184
sys:page-in-structure, 183
sys:page-in-words, 184
sys:page-out-area, 184
sys:page-out-array, 184
sys:page-out-region, 184
sys:page-out-structure, 184
sys:page-out-words, 184
pairlis, 71
time:parse, 496
fs:parse-pathname, 382
time:parse-universal-time, 496
fs:pathname-plist, 384
peek, 503
pkg-bind, 398

pkg-create-package, 400
pkg-find-package, 401
pkg-goto, 398
pkg-kill, 401
pkg-name, 401
pkg-refname-alist, 401
pkg-super-package, 401
plane-aref, 123
plane-aset, 123
plane-default, 123
plane-extension, 123
plane-origin, 123
plane-ref, 123
plane-store, 123
plist, 88
format:plural, 359
plus, 96
plusp, 94
pop, 231
prin1, 334
prin1-then-space, 335
princ, 335
print, 335
time:print-brief-universal-time, 495
time:print-current-date, 495
time:print-current-time, 495
time:print-date, 495
si:print-disk-error-log, 503
print-disk-label, 422
format:print-list, 355
si:print-list, 365
print-loaded-band, 423
print-notifications, 502
si:print-object, 365
print-sends, 502
print-system-modifications, 418
time:print-time, 495
time:print-universal-date, 495
time:print-universal-time, 495
probef, 367
process-allow-schedule, 431
process-disable, 439
process-enable, 439
process-initial-form, 439
process-initial-stack-group, 439
process-lock, 432
process-name, 439
process-preset, 439
process-reset, 439
process-reset-and-enable, 439
process-run-function, 434
process-run-restartable-function, 434
process-run-temporary-function, 434
process-sleep, 431
process-stack-group, 439
process-unlock, 433
process-wait, 431
process-wait-argument-list, 439
process-wait-function, 439
process-wait-with-timeout, 431
process-whostate, 439
prog, 43
prog*, 45
prog1, 28
prog2, 28
progn, 28
progv, 18
progw, 18
property-cell-location, 88
psetq, 16
push, 231
puthash, 76
puthash-equal, 77
putprop, 73

q-data-types, 174
qc-file, 197
qc-file-load, 198
qsend, 502
quote, 25
quotient, 97

random, 104
si:random-create-array, 105
si:random-initialize, 105
rass, 70
rassoc, 70
rassq, 70
read, 332
read-for-top-level, 333
read-from-string, 334
read-meter, 188
readch, 333
readfile, 368
readline, 333
readlist, 334
si:receive-band, 425
recompile-flavor, 297
rem, 68
rem-if, 68
rem-if-not, 68
remainder, 97
remhash, 76
remhash-equal, 77
remob, 400
remove, 68
remprop, 73
remq, 68
si:rename-within-new-definition-maybe, 157
renamef, 367
reset-initializations, 491
rest1, 55
rest2, 55
rest3, 55

rest4, 55
return, 45
return-array, 117
return-from, 46
return-list, 46
reverse, 58
room, 500
rot, 101
rplaca, 61
rplacd, 62

samepnamep, 89
sassoc, 71
sassq, 71
zwei:save-all-files, 502
si:sb-on, 432
second, 55
select, 36
selector, 37
selectq, 35
selectq-every, 37
set, 86
set-character-translation, 329
set-current-band, 423
set-current-microload, 423
fs:set-default-pathname, 383
set-in-closure, 161
set-in-instance, 299
set-mar, 467
set-memory-size, 501
set-syntax-#-macro-char, 330
set-syntax-from-char, 329
set-syntax-from-description, 330
set-syntax-macro-char, 329
si:set-system-status, 421
setarg, 29
setf, 229
setplist, 88
setq, 16
setsyntax, 330
setsyntax-sharp-macro, 331
seventh, 55
si:advise-1, 461
si:compare-band, 426
si:define-defsystem-special-variable, 414
si:define-make-system-special-variable, 413
si:define-simple-transformation, 414
si:edit-disk-label, 423
si:encapsulate, 153
si:fdefinition-location, 150
si:fdefinition-symbol-or-location, 150
si:get-system-version, 418
si:lisp-top-level, 503
si:lisp-top-level1, 504
si:load-mcr-file, 425
si:loop-named-variable, 254
si:loop-tassoc, 253
si:loop-tequal, 253
si:loop-tmember, 253
si:print-disk-error-log, 503
si:print-list, 365
si:print-object, 365
si:random-create-array, 105
si:random-initialize, 105
si:receive-band, 425
si:rename-within-new-definition-maybe, 157
si:sb-on, 432
si:set-system-status, 421
si:system-version-info, 418
si:transmit-band, 426
si:unadvise-1, 461
si:unencapsulate-function-spec, 155
si:unwire-page, 183
si:wire-page, 183
signal, 441
signp, 94
sin, 99
sind, 99
sixth, 55
small-float, 100
small-floatp, 9
math:solve, 122
some, 69
sort, 80
sort-grouped-array, 81
sort-grouped-array-group-key, 81
sortcar, 81
special, 201
sqrt, 98
sstatus, 508
stable-sort, 81
stable-sortcar, 81
stack-group-preset, 167
stack-group-resume, 167
stack-group-return, 167
status, 507
step, 464
store, 125
store-array-leader, 114
stream-copy-until-eof, 335
stream-default-handler, 345
string, 128
string-append, 129
string-downcase, 127
string-equal, 128
string-left-trim, 130
string-length, 128
string-lessp, 129
string-nconc, 129
string-nreverse, 130
string-pluralize, 130
string-reverse, 130
string-reverse-search, 132
string-reverse-search-char, 132

string-reverse-search-not-char, 132
string-reverse-search-not-set, 133
string-reverse-search-set, 132
string-right-trim, 130
string-search, 131
string-search-char, 131
string-search-not-char, 131
string-search-not-set, 132
string-search-set, 131
string-trim, 130
string-upcase, 127
stringp, 9
structure-forward, 175
subl, 98
sublis, 63
subrcall, 25
subrp, 10
subset, 68
subset-not, 68
subst, 62
substring, 129
supdup, 503
swaphash, 76
swaphash-equal, 77
sxhash, 78
symbol-package, 400
symbolp, 8
symeval, 86
symeval-in-closure, 161
symeval-in-instance, 299
symeval-in-stack-group, 167
sys:%binding-instances, 185
sys:%change-page-status, 184
sys:%compute-page-hash, 184
sys:%create-physical-page, 184
sys:%delete-physical-page, 184
sys:%disk-restore, 185
sys:%disk-save, 185
sys:%halt, 178
sys:%internal-value-cell, 185
sys:%using-binding-instances, 185
sys:%xbus-write-sync, 178
sys:page-in-area, 184
sys:page-in-array, 184
sys:page-in-region, 184
sys:page-in-structure, 183
sys:page-in-words, 184
sys:page-out-area, 184
sys:page-out-array, 184
sys:page-out-region, 184
sys:page-out-structure, 184
sys:page-out-words, 184
si:system-version-info, 418

format:tab, 358
tailp, 67
telnet, 503
terpri, 335
third, 55
throw, 48
time, 494
time-difference, 494
time-lessp, 494
time:day-of-the-week-string, 498
time:daylight-savings-p, 497
time:daylight-savings-time-p, 497
time:decode-universal-time, 497
time:encode-universal-time, 497
time:fixnum-microsecond-time, 494
time:get-time, 494
time:get-universal-time, 494
time:initialize-timebase, 497
time:leap-year-p, 497
time:microsecond-time, 494
time:month-length, 497
time:month-string, 498
time:parse, 496
time:parse-universal-time, 496
time:print-brief-universal-time, 495
time:print-current-date, 495
time:print-current-time, 495
time:print-date, 495
time:print-time, 495
time:print-universal-date, 495
time:print-universal-time, 495
time:timezone-string, 498
time:verify-date, 498
times, 97
time:timezone-string, 498
trace, 457
si:transmit-band, 426
math:transpose-matrix, 121
true, 27
tyi, 332
tyipeek, 333
tyo, 335
typep, 10

unadvise, 461
si:unadvise-1, 461
unadvise-within, 463
uncompile, 197
undefmethod, 297
undefun, 150
si:unencapsulate-function-spec, 155
unmonitor-variable, 468
unspecial, 201
untrace, 459
unwind-protect, 49
si:unwire-page, 183
fs:user-homedir, 384
using-resource, 84

value-cell-location, 87

values, 30
values-list, 30
time:verify-date, 498

what-files-call, 499
where-is, 500
who-calls, 499
who-uses, 499
si:wire-page, 183
with-input-from-string, 133
with-open-file, 365
with-output-to-string, 134
without-interrupts, 430
write-meter, 188

xcons, 54
xcons-in-area, 54
xstore, 125

y-or-n-p, 487
yes-or-no-p, 487

zerop, 94
zwei:save-all-files, 502

\, 97
\\, 98

^, 98
^$, 98